1973 SAGAMORE COMPUTER CONFERENCE

PROCEEDINGS

OF THE

1973 SAGAMORE COMPUTER CONFERENCE

ON

PARALLEL PROCESSING

Syracuse University

in cooperation with

IEEE ACM

PROCEEDINGS

OF THE

1973 SAGAMORE COMPUTER CONFERENCE

ON

PARALLEL PROCESSING

Papers presented on

August 22-24, 1973

Department of Electrical & Computer Engineering

SYRACUSE UNIVERSITY

in Cooperation with

IEEE ACM

IEEE Catalog Number 73 CH0812-8 C

Additional copies available from IEEE, 345 East 47th Street, New York, N.Y. 10017.

Price: $12.00 per copy

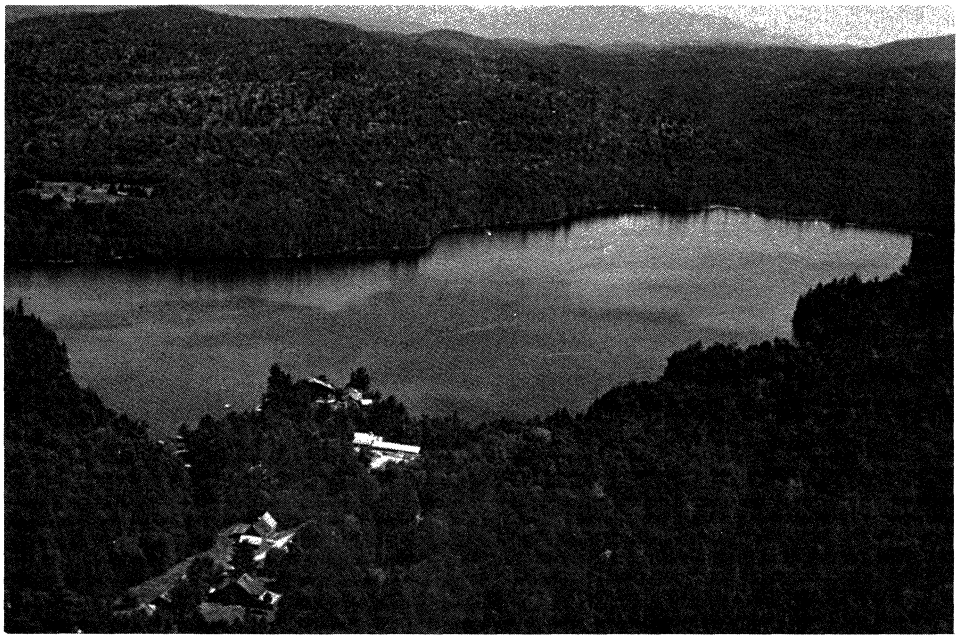
PREFACE

The 1973 Sagamore Computer Conference on Parallel Processing was held on August 22-24, 1973 at the former Vanderbilt summer estate in the Central Adirondack Mountains. The Conference was conceived to provide a secluded environment, a 1300-acre preserve surrounding the private Sagamore Lake, for the participants with excellent opportunities for exchanging ideas and learning each others research activities. Thus, informative discussions may be made not only during the technical sessions but also throughout the various sports and social gatherings provided by the Conference.

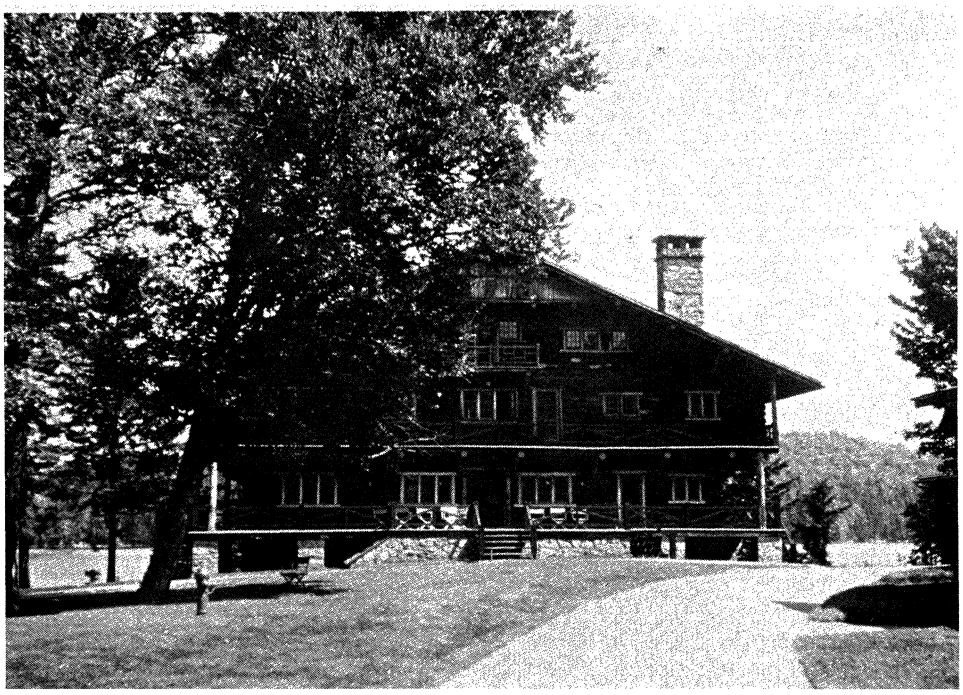
The enthusiastic cooperation and response that we received throughout the Conference and during its preparation was indeed most heartening. We not only received many more papers than we could possibly schedule, but also the number of requests to attend exceeded the Sagamore accomodations. Thus, there seems to be a popular demand for such a conference in parallel processing. Another conference is being scheduled for the next year — August 21-23, 1974.

The success of such a conference requires the vigorous support of many individuals. In this respect, we are most grateful to all the authors who submitted their papers for consideration. It is our deep regret that not all qualified papers could be scheduled for the Conference. We are also much indebted to all the reviewers who, in order to meet the stringent review deadlines, put aside their own busy work schedule to carefully evaluate the papers sent for their judgement. Their valuable comments not only resulted a set of high-quality papers for the Conference, but also were sincerely appreciated by many authors. The generous help we received from the session chairmen also contributes much to the success of the Conference. In addition, we wish to acknowledge the excellent cooperation provided to us by IEEE, IEEE Computer Society, ACM, their local chapter chairmen, as well as the staff of various technical magazines. In particular, we are indebted to Mr. James J. Andover, Mr. Charles Casale, Dr. W. Smith Dorsey, Prof. Michael J, Flynn, Prof. Caxton C. Foster, Mrs. Irene Hollister, Mr. David Jacobsohn, Mr. John L. Kirkley, Mr. E. D. MacDonald, Prof. Harold S. Stone, and many others for their assistance in achieving such a cooperation. Special thanks are also due to members of various committees. Their time and effort devoted to the Conference are indeed invaluable.

Tse-yun Feng
Department of Electrical & Computer Engineering
Syracuse University



The Secluded Environment of Sagamore



The Main Lodge

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D.)

TABLE OF CONTENTS (CONT'D.)

THE COORDINATE METHOD FOR THE PARALLEL EXECUTION OF DO LOOPS

Leslie Lamport
Massachusetts Computer Associates
Wakefield, Massachusetts 01880

Abstract -- An algorithm is presented which translates a program with nested sequential DO loops into one suitable for execution on a parallel array or vector computer. If necessary, extensive rearrangement of the program's structure is made.

Introduction

We consider the problem of compiling ordinary sequential programs for execution on a parallel array or vector computer such as the Illiac IV or the CDC Star-100. This problem is of practical importance for the following reasons:

(1) There exist sequential programs which one would like to run on these parallel computers.

(2) If a program is to be run on two different machines, it might be best to write it in sequential form and let each compiler find the most efficient parallel execution for its computer.

(3) A compiler may be able to find more parallelism in a program than the programmer can. (See [1].)

The methods which we introduce should also be useful in other areas of program optimization.

We consider a FORTRAN program containing DO loops, and describe a method of translating it into an extended FORTRAN program in which one or more of the DO loops is executed in parallel. This is an obvious approach, and has been used in [2] - [4]. The method presented here generalizes the coordinate method of [4], and is more general than the analogous methods of [2] and [3]. Although our exposition is self-contained, it is best to read [4] first.

We specify parallel execution with a DO SIM statement of the following form:

DO 99 SIM FOR ALL I $\in$ $\mathcal{S}$,

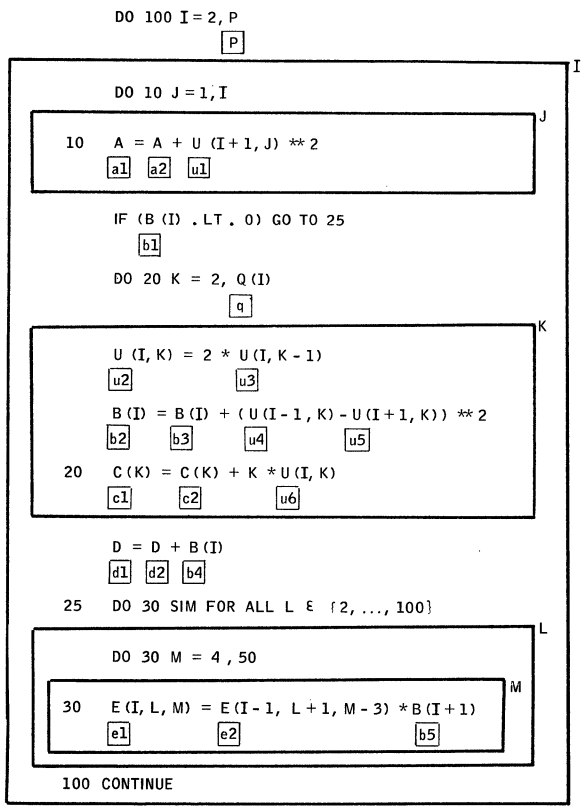
where $\mathcal{S}$ is a set of integers. The statements in its range are executed one after another as usual. However, each statement is executed simultaneously for all of the indicated values of I. An assignment statement is executed by first computing the right-hand side for each value of I, then simultaneously performing the assignments. Thus, the statement

A(I) = A(I - 1) + B(I)

would simultaneously set A(i) equal to the original value of A(i - 1) plus B(i) , for each value of i in $\mathcal{S}$.

The coordinate method tries to change DO loops to DO SIM loops. We show that it suffices to consider one DO loop at a time. The basic method is the coordinate algorithm, which we illustrate by an example. Suppose we are given the following program.

Program 1:



This is a nonsensical program, but it will serve to illustrate most details of the algorithm. Each occurrence of a non-index variable is given a name, which appears in a box beneath it. Loop bodies are boxed and labeled for legibility. The L loop might have been changed from a DO to a

DO SIM loop by a previous application of the algorithm.

The coordinate algorithm can translate Program 1 into the following equivalent program.

Program 2:

```
TMP1 = P
       [P]

DO 901 I1 = 2, TMP1                                          I1
DO 10 J = 1, I1
10  A = A + U(I1+1, J) ** 2                                  J
    [a1] [a2] [u1]
901 CONTINUE

DO 902 SIM FOR ALL I2 ε {i : 2 ≤ i ≤ TMP1}                   I2
TMP2 (I2) = .NOT. (B(I2) .LT. 0)
                  [b1]
IF (TMP2(I2)) TMP3(I2) = Q(I2)
                         [q]
DO 30 SIM FOR ALL L ε {2, ..., 100}                          L
DO 30 M = 4, 50                                              M
30  E(I2, L, M) = E(I2 - 1, L + 1, M - 3) * B(I2 + 1)
    [e1]          [e2]                  [b5]
902 CONTINUE

DO 920 K = 2, MAXIMUM ({TMP3(i):                             K
          2 ≤ i ≤ TMP1 .AND. TMP2(i)})
DO 904 SIM FOR ALL I4 ε {i : K ≤ TMP3(i)
          .AND. 2 ≤ i ≤ TMP1 .AND. TMP2(i)}
TMP4 (I4) = U (I4 + 1, K)                                    I4
            [u5]
U(I4, K) = 2 * U(I4, K - 1)
[u2]       [u3]
B(I4) = B(I4) + (U(I4 - 1, K) - TMP4(I4)) ** 2
[b2]    [b3]    [u4]
TMP5 (I4) = K * U(I4, K)
                [u6]
904 CONTINUE
DO 905 I5 = 2, TMP1                                          I5
IF (TMP2(I5) .AND. K ≤ TMP3(I5))
                C(K) = C(K) + TMP5(I5)
                [c1]   [c2]
905 CONTINUE
920 CONTINUE
```

(a) We need only assume that we know which data can be modified by a subroutine or function call, but this would complicate matters.

```
DO 903 I3 = 2, TMP1                                          I3
IF (TMP2(I3)) D = D + B(I3)
                [d1] [d2] [b4]
903 CONTINUE
```

Observe that the I loop of Program 1 has been split into the five $\bar{I}_1, \ldots, \bar{I}_5$ loops. Two of these are DO SIM loops, so Program 2 has more parallel execution than Program 1. Note the extensive rearrangement of Program 1 needed to achieve this parallelism. The L/M loop has been moved before the K loop; statement 20 has been split into two parts which appear inside different loops; the u5 occurrence has been moved; etc. Of course, this example is contrived to demonstrate the power of the algorithm.

In general, we consider an extended FORTRAN program containing DO and DO SIM loops, with the following restrictions.

1. There is no backward transfer of control other than that implied by the DO loops. Thus, if all DO and DO SIM statements were removed, then the resulting program would have no loops. (Techniques for translating programmed loops into DO loops are described in [3].)
2. There is no I/O statement. We assume that input/output is done with the initial/final values of variables.
3. The increment of every DO loop is a constant which is known at compile time.
4. There is no transfer of control from inside the range of a DO or DO SIM loop to outside its range - i.e., no premature exits from loops.
5. There is no subroutine call, and no function call which can change the value of a variable. The value of a function must depend only on the values of its arguments.(a)

The program which we consider here may be any portion of an actual FORTRAN program having a single entry point. In particular, it may consist of a single DO loop. Hence, these restrictions are reasonable.

Space limitations require that we eliminate many details, including the proofs of theorems. They will appear in [5].

Representation of the Program

For our analysis, we need a way of representing a program which is more convenient than the original FORTRAN representation. To simplify the exposition, we assume that all DO loop increments equal 1. The generalization to arbitrary increments is described later.

The Program Tree

The first part of our representation is the program tree, which describes a program's nested loop structure. The terminal nodes of the tree represent occurrences of variables. (Occurrences of DO and DO SIM index variables are excluded.) The non-terminal nodes represent the DO and DO

SIM loop bodies. A dummy node, labeled o , is placed at the top of the tree.

The program trees of Programs 1 and 2 are shown in Figures 1 and 2, respectively. (For Program 2, we have excluded occurrences of TMP1, ..., TMP5 from the tree.) Occurrence nodes are denoted by boxes. Loop nodes are denoted by circles and labeled by the index variable name. (We assume that each loop has a unique index variable.) DO SIM nodes are distinguished by concentric circles.

We use paternity relations to describe tree structure. In Figure 1, the J node is the father of the u1 node and the son of the I node. The o node is an ancestor of all other nodes.

We let $\eta(\mathfrak{T})$ denote the set of all nodes of a tree $\mathfrak{T}$, and $\mathcal{O}(\mathfrak{T})$ denote the set of all terminal nodes. If α is any node of a tree, then $\mathfrak{T}(\alpha)$ denotes the subtree headed by α. We let $\eta(\alpha)$ and $\mathcal{O}(\alpha)$ denote $\eta[\mathfrak{T}(\alpha)]$ and $\mathcal{O}[\mathfrak{T}(\alpha)]$, respectively. In Figure 2, $\mathcal{O}(\overline{I}_2) = \{\overline{b1}, \overline{q}, \overline{e1}, \overline{e2}, \overline{b5}\}$.

A non-empty sequence of nodes $\alpha_1, \ldots, \alpha_n$ is called a branch of a tree if o is the father of α_1, and each α_k is the father of α_{k+1}. Three branches of Figure 1 are: (1) I, K; (2) I, J, a2; and (3) p .

If α and β are two nodes of a tree, we let $\alpha \cap \beta$ denote their most recent common ancestor. In Figure 1, we have a1 $\cap$ a2 = J , u3 $\cap$ J = I and p $\cap$ q = o . We define $\alpha \cap \alpha$ to be the father of α .

Let f and g be occurrences in a program. We say that f precedes g if there is a flow path from f to g in which each DO loop is executed at most once. In Program 1, u3 precedes u2, u5, e2, etc. By restriction 1, if f precedes g then g cannot precede f .

The motive for the following definition comes from considering $f \rightarrow g$ to mean that the occurrence f must precede the occurrence g. Then $\overset{*}{\rightarrow}$ contains precedence relations on the loop nodes implied by $\rightarrow$.

Definition 1: Let $\mathfrak{T}$ be a tree and let $\rightarrow$ be any relation on $\mathcal{O}(\mathfrak{T})$. The tree completion of $\rightarrow$ is the smallest relation $\overset{*}{\rightarrow}$ on $\eta(\mathfrak{T})$ which satisfies the following conditions:

(1) If $f \rightarrow g$ then $f \overset{*}{\rightarrow} g$.

(2) If $\alpha \overset{*}{\rightarrow} \beta$, $\beta \overset{*}{\rightarrow} \gamma$ and α is neither an ancestor nor a descendant of γ , then $\alpha \overset{*}{\rightarrow} \gamma$.

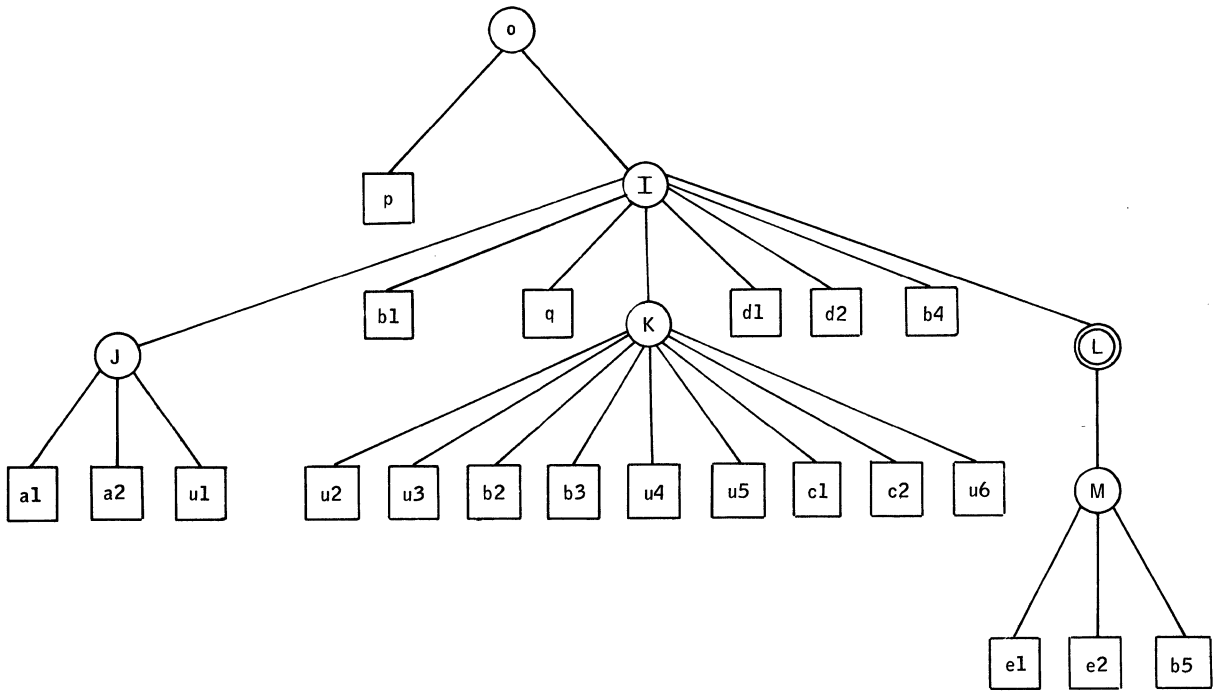


Figure 1

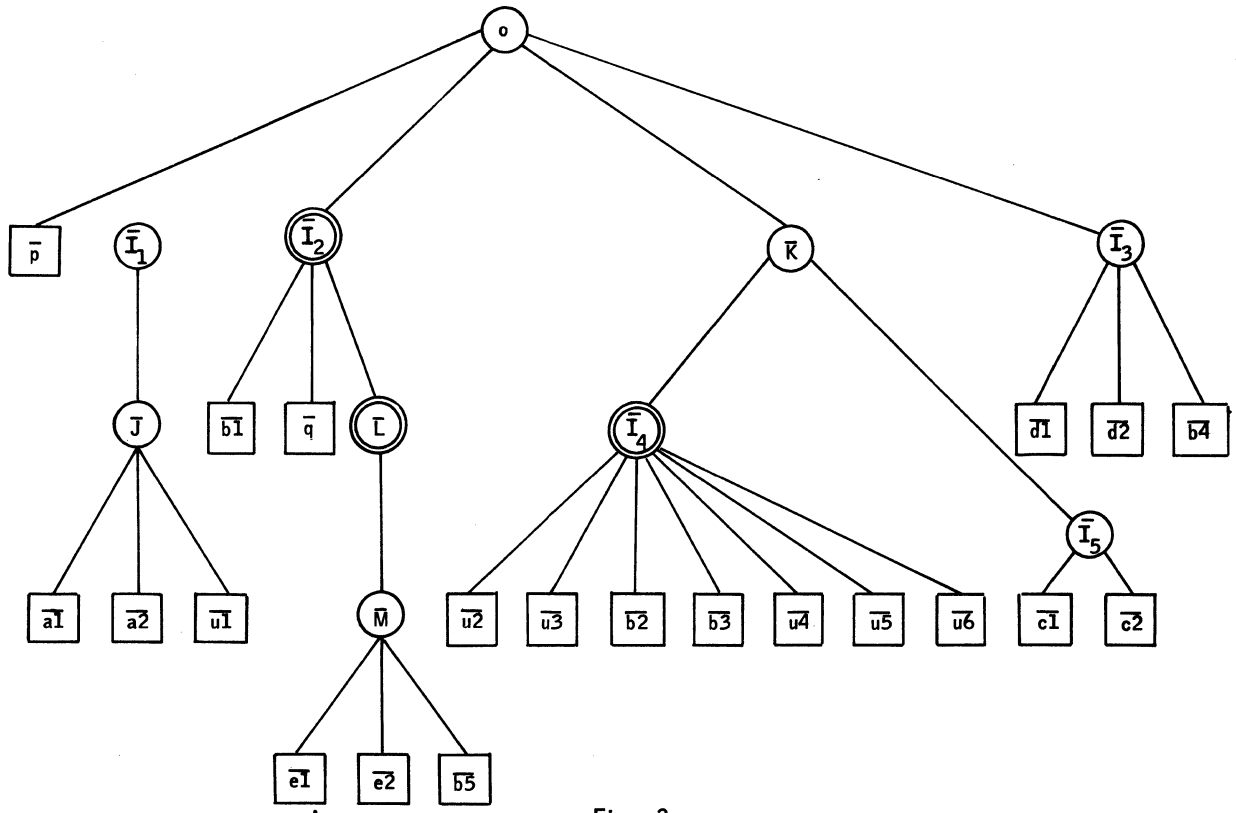


Figure 2

(3) If $\alpha \overset{*}{\rightarrow} \beta$, $\alpha \neq \beta$, α' is either the father of α or else $\alpha' = \alpha$, β' is either the father of β or else $\beta' = \beta$, and $\alpha' \neq \beta'$; then $\alpha' \overset{*}{\rightarrow} \beta'$.

The relation $\rightarrow$ is said to be tree inconsistent if $\alpha \overset{*}{\rightarrow} \alpha$ for some node α. Otherwise, $\rightarrow$ is said to be tree consistent.

A partition of a set is a collection of pairwise disjoint subsets whose union equals the whole set. Let $\mathcal{P} = \{ S_1, \ldots, S_n \}$ be a partition of a set S, and let $\rightarrow$ be a relation on S. The relation $\overline{\rightarrow}$ induced on $\mathcal{P}$ by $\rightarrow$ is defined by $S_i \overline{\rightarrow} S_j$ if and only if $S_i \neq S_j$ and there exist $s \in S_i$ and $t \in S_j$ such that $s \rightarrow t$.

A tree partition $\mathcal{P}$ of a tree $\mathcal{T}$ is a partition $\{ N_1, \ldots, N_r \}$ of $\eta(\mathcal{T})$ satisfying the following property: If $\alpha \in N_i$, β and $\gamma \in N_j$, $N_i \neq N_j$ and α is the father of β, then the father of γ is contained in either N_i or N_j. We give $\mathcal{P}$ a tree structure by letting the father/son relation be the one induced on $\mathcal{P}$ by the father/son relation of $\mathcal{T}$.

As an example, let $\mathcal{T}$ be the subtree $\mathcal{T}(\bar{I}_2)$ of Figure 2. Then $\{\bar{b1}\}$, $\{\bar{I}_2, \bar{q}, \bar{L}\}$, $\{\bar{M}, \bar{e1}, \bar{b5}\}$, $\{\bar{e2}\}$ is a tree partition of $\mathcal{T}$. Its tree structure is shown in Figure 3.

Index Sets

Let $\mathbb{Z}^n$ denote the set of all n-tuples of integers, with the usual operations of addition and subtraction, and let $\vec{0} = (0, 0, \ldots, 0)$. We define $\mathbb{Z}^0 = \{0\}$.

Let α be a loop node of a program tree, and let $I^1, \ldots, I^n$ be the branch with $I^n = \alpha$. Then $|\alpha|$ is defined to equal n. We define $\mathbb{Z}_\alpha$ to be the set $\mathbb{Z}^{|\alpha|}$.

The relations $\sim$ and $\prec$ on $\mathbb{Z}_\alpha$ are defined as follows. Let $I^{j_1}, \ldots, I^{j_k}$ be the DO

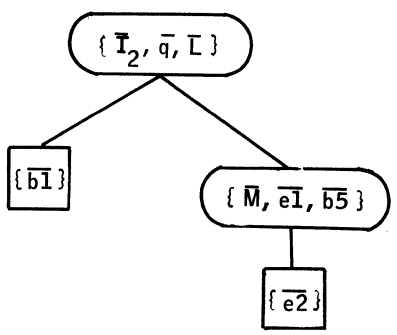


Figure 3

nodes among the I^j (the remaining I^j being DO SIM nodes), with $j_1 < \ldots < j_k$. For any elements $(a^1, \ldots, a^n)$ and $(b^1, \ldots, b^n)$ of $\mathbb{Z}_\alpha$, we let

(1) $(a^1, \ldots, a^n) \sim (b^1, \ldots, b^n)$ if $(a^{j_1}, \ldots, a^{j_k}) = (b^{j_1}, \ldots, b^{j_k})$.

(2) $(a^1, \ldots, a^n) < (b^1, \ldots, b^n)$ if $(a^{j_1}, \ldots, a^{j_k})$ is lexicographically smaller than $(b^{j_1}, \ldots, b^{j_k})$, reading the components from left to right.

In Figure 1, we have $|M| = 3$, $\mathbb{Z}_M = \mathbb{Z}^3$, and $(3, 7, 2) \sim (3, 10, 2) < (3, -1, 4)$.

The element $A = (a^1, \ldots, a^n)$ of $\mathbb{Z}_\alpha$ represents a possible execution of the body of the α loop for $I^1 = a^1, \ldots, I^n = a^n$. For any element B of $\mathbb{Z}_\alpha$, $A \sim B$ if the executions of the α loop body for A and B occur simultaneously, and $A < B$ if the execution for A precedes the execution for B. Note that this defines the meaning of a DO loop inside a DO SIM loop. (It is not the meaning one might expect if the lower DO limit depends upon the DO SIM index variable.

We define $|o| = 0$, and let the relation $\sim$ on $\mathbb{Z}_o = \{0\}$ be defined by $0 \sim 0$. If f is an occurrence node whose father is α, then we let $|f| = |\alpha|$ and $\mathbb{Z}_f = \mathbb{Z}_\alpha$. An element of $\mathbb{Z}_f$ represents a possible execution of the occurrence f.

Now let the I^j be as above and let $\beta = I^k$, $k \leq n$. We define the projection mapping $\Pi^\alpha_\beta : \mathbb{Z}_\alpha \to \mathbb{Z}_\beta$ by $\Pi^\alpha_\beta(a^1, \ldots, a^n) = (a^1, \ldots, a^k)$. If an occurrence node f is the son of α, then we let $\Pi^f_\beta = \Pi^\alpha_\beta$. In Figure 1, $\Pi^{el}_I : \mathbb{Z}^3 \to \mathbb{Z}^1$ is defined by $\Pi^{el}_I(i, \ell, m) = (i)$.

The reader can verify the following fact.

Proposition 2: Let f and g be occurrences, $P \in \mathbb{Z}_f$ and $Q \in \mathbb{Z}_g$. The execution of f for P precedes the execution of g for Q if either

(i) $\Pi^f_{f\cap g}(P) < \Pi^g_{f\cap g}(Q)$, or

(ii) $\Pi^f_{f\cap g}(P) \sim \Pi^g_{f\cap g}(Q)$ and f precedes g.

For any node α, we define the index set $\mathcal{I}_\alpha$ to be the subset of $\mathbb{Z}_\alpha$ consisting of those elements for which α is actually executed. In Program 1, we have:

$\mathcal{I}_{a1} = \mathcal{I}_J = \{(i,j) : 2 \leq i \leq P$ and $1 \leq j \leq i\}$

$\mathcal{I}_q = \{(i) : 2 \leq i \leq P$ and $B(i) \geq 0\}$

$\mathcal{I}_{u2} = \mathcal{I}_K = \{(i,k) : 2 \leq i \leq P, 2 \leq k \leq Q(i)$ and $B(i) \geq 0\}$.

Note that in general, $\mathcal{I}_\alpha$ may depend upon the initial values of variables, and often will not be known at compile time.

Occurrences

An occurrence of a variable is called a generation if it appears on the left-hand side of an assignment statement, otherwise it is called a use. A relevant occurrence pair is an ordered pair of occurrences of a single variable, at least one of which is a generation. In Program 1, there are three relevant pairs of occurrences of the variable E: (1) e1, e2; (2) e2, e1; and (3) e1, e1.

Execution of the occurrence b5 of Program 1 for an element (i, ℓ, m) in $\mathcal{I}_{b5}$ references the (i+1) element of the array B. This defines the occurrence mapping $T_{b5} : \mathcal{I}_{b5} \to \mathbb{Z}^1$ given by $T_{b5}(i, \ell, m) = (i+1)$. In general, let f be an occurrence of a k-dimensional array variable. (A scalar is considered to be a 0-dimensional array.) Then $T_f : \mathcal{I}_f \to \mathbb{Z}^k$. The mapping T_f may not be known at compile time.[b]

Definition 3: Let f, g be a relevant occurrence pair. We define $<< f, g >>$ to be the set $\{X \in \mathbb{Z}_{f\cap g} :$ there exist $P \in \mathcal{I}_f$ and $Q \in \mathcal{I}_g$ such that $T_f(P) = T_g(Q)$ and $X = \Pi^g_{f\cap g}(Q) - \Pi^f_{f\cap g}(P)\}$.

We define $< f, g >$ to be some fixed subset of $\mathbb{Z}_{f\cap g}$, known at compile time, which contains $<< f, g >>$.

An element X of $<< f, g >>$ implies the existence of elements $P \in \mathcal{I}_f$ and $Q \in \mathcal{I}_g$ such that the executions of f for P and g for Q reference the same array element. Since $A < B$ if $B - A > \vec{0}$, Proposition 2 implies that the reference by f precedes the reference by g if either (i) $X > \vec{0}$ or (ii) $X \sim \vec{0}$ and f precedes g.

Some $<< f, g >>$ sets for Program 1 are:

$<< e1, e2 >> = \{(1, -1, 3)\}$

$<< u1, u2 >> = \{(1)\}$ if $\mathcal{I}_K \neq \emptyset$ [c]

$<< b2, b3 >> = \{(0, k) : 2\text{-}Q(i) \leq k \leq Q(i) - 2$ for some $i \in \mathcal{I}_K\}$.

The set $< f, g >$ is the best "upper bound" on the set $<< f, g >>$ which the compiler can find. Computing these sets is a major implementation problem which we will not discuss. We assume that the compiler finds the following $< f, g >$ sets for Program 1.

---

[b] If f appears in a DO SIM set expression, then T_f could be a multi-valued mapping. To handle this case, replace any statement in this paper of the form "... $T_f(P)$ ..." by "there exists an $X \in T_f(P)$ such that ... X ...".

[c] We let $\emptyset$ denote the empty set.

$< a1, a1 > = < a1, a2 > = < a2, a1 > = \mathbb{Z}^2$
$< b1, b2 > = < b2, b1 > = < b2, b4 >$
$= < b4, b2 > = \{(0)\}$
$< b2, b2 > = < b2, b3 > = < b3, b2 >$
$= \{(0, k) : k \text{ any integer}\}$
$< b2, b5 > = \{(-1)\}$
$< b5, b2 > = \{(1)\}$
$< c1, c1 > = < c1, c2 > = < c2, c1 >$
$= \{(i, 0) : i \text{ any integer}\}$
$< d1, d1 > = < d1, d2 > = < d2, d1 > = \mathbb{Z}^1$
$< e1, e1 > = \{(0, 0, 0)\}$
$< e1, e2 > = \{(1, -1, 3)\}$
$< e2, e1 > = \{(-1, 1, -3)\}$
$< u2, u2 > = < u2, u6 > = < u6, u2 >$
$= \{(0, 0)\}$
$< u1, u2 > = \{(1)\}$
$< u2, u1 > = \{(-1)\}$
$< u2, u3 > = \{(0, 1)\}$
$< u3, u2 > = \{(0, -1)\}$
$< u2, u4 > = < u5, u2 > = \{(1, 0)\}$
$< u4, u2 > = < u2, u5 > = \{(-1, 0)\}$

Precedence Relations

The FORTRAN representation of a program usually specifies more precedence relations among the occurrences than are necessary. For example, b2 need not precede c1 in Program 1. We now describe all the precedence relations that are necessary in order to specify the correct execution of a program. These are of two types. The first, denoted by $\Rightarrow$, describes those precedence relations which are logically necessary for a meaningful execution of the program.

Definition 4: For occurrences f and g in a FORTRAN program, we write $f \Rightarrow g$ in any of the following cases:

1. (a) g is a generation and f appears on the right-hand side of the assignment statement of g.
   (b) f appears in a subscript expression of g.
2. (a) f appears in the conditional expression of a conditional branch, and g appears in a statement whose execution is conditional upon which branch is taken.
   (b) f appears in the limits of a DO statement, or in the index set expression of a DO SIM statement, whose range contains g.

In 2(a), we consider a conditional assignment statement to consist of a conditional branch and an assignment statement.

The relations $\Rightarrow$ for Program 1 are indicated in Figure 4. E.g., the $\Rightarrow$ in the a2 row, a1 column denotes the relation a2 $\Rightarrow$ a1.

The second form of precedence relation, denoted by $\rightarrow$, is necessitated by data conflicts. If a generation and any other occurrence reference the same array element, then the order of the references must be specified. Our previous remarks then lead to the following definition.

Definition 5: For each relevant pair of occurrences f, g with $f \neq g$, we let $f \rightarrow g$ if and only if f precedes g and there exists an element $X \in < f, g >$ with $X \sim \vec{0}$.

The relations $\rightarrow$ for Program 1 are shown in Figure 4.

We let $\Rightarrow\!\!\!\!\rightarrow$ denote the union of the relations $\rightarrow$ and $\Rightarrow$, so $f \Rightarrow\!\!\!\!\rightarrow g$ if $f \rightarrow g$ or $f \Rightarrow g$. Then $\Rightarrow\!\!\!\!\rightarrow$ gives all precedence relations necessary for the proper execution of the program. It can be used to determine, for example, that during an iteration of the I loop of Program 1, the J and K loops can be executed concurrently by two independent processors. This yields a generalization of the methods of [6]. However, this type of parallelism will not be discussed here.

The Complete Representation

We define a program specification $\mathcal{S}$ to consist of the following:

S1. A program tree, also denoted by $\mathcal{S}$.
S2. The precedence relations $\rightarrow$ and $\Rightarrow$.
S3. A specification of the occurrence mapping for each occurrence.
S4. A specification of the index set of each occurrence and of the assignment values for each generation, in terms of occurrences.

Part S4 is quite vague. For Program 1, it might include the following:

$\mathcal{I}_q = \{(i) : 2 \leq i \leq p \text{ and } b1 \geq 0\}$
a1 = a2 + u1**2 .

S3 is also vague if we consider occurrences like A(B(I), J). We will not need to define S3 and S4 any more precisely because our translation procedure will leave these parts of the specification essentially unchanged.

There are many criteria which must be met for S1-S4 to be a valid program specification. However, if S3 and S4 are assumed to be valid, then the following conditions are sufficient to insure that the entire specification is valid.

L1. (a) For each relevant pair of occurrences f, g with $f \neq g$: if there exists an $X \in << f, g >>$ with $X \sim \vec{0}$, then either $f \rightarrow g$ or $g \rightarrow f$.

(b) For each generation g: if $X \in << g, g >>$ and $X \sim \vec{0}$, then $X = \vec{0}$.

L2. The relation $\Rightarrow\!\!\!\!\rightarrow$ is tree consistent.

Note that to verify L1, it suffices to verify it with each set $<< f, g >>$ replaced by $< f, g >$.

Given a valid program specification, we can use it to write an extended FORTRAN program. For example, we define a program specification as follows:

S1. The program tree is given by Figure 2.

S2. We let $\overline{f} \rightarrow$ or $\Rightarrow \overline{g}$ if the relation $f \rightarrow$ or $\Rightarrow g$ appears in Figure 4. We also add the following relations: $\overline{u1} \rightarrow \overline{u2}$, $\overline{u2} \rightarrow \overline{u4}$, $\overline{u5} \rightarrow \overline{u2}$, and $\overline{b5} \rightarrow \overline{b2}$.

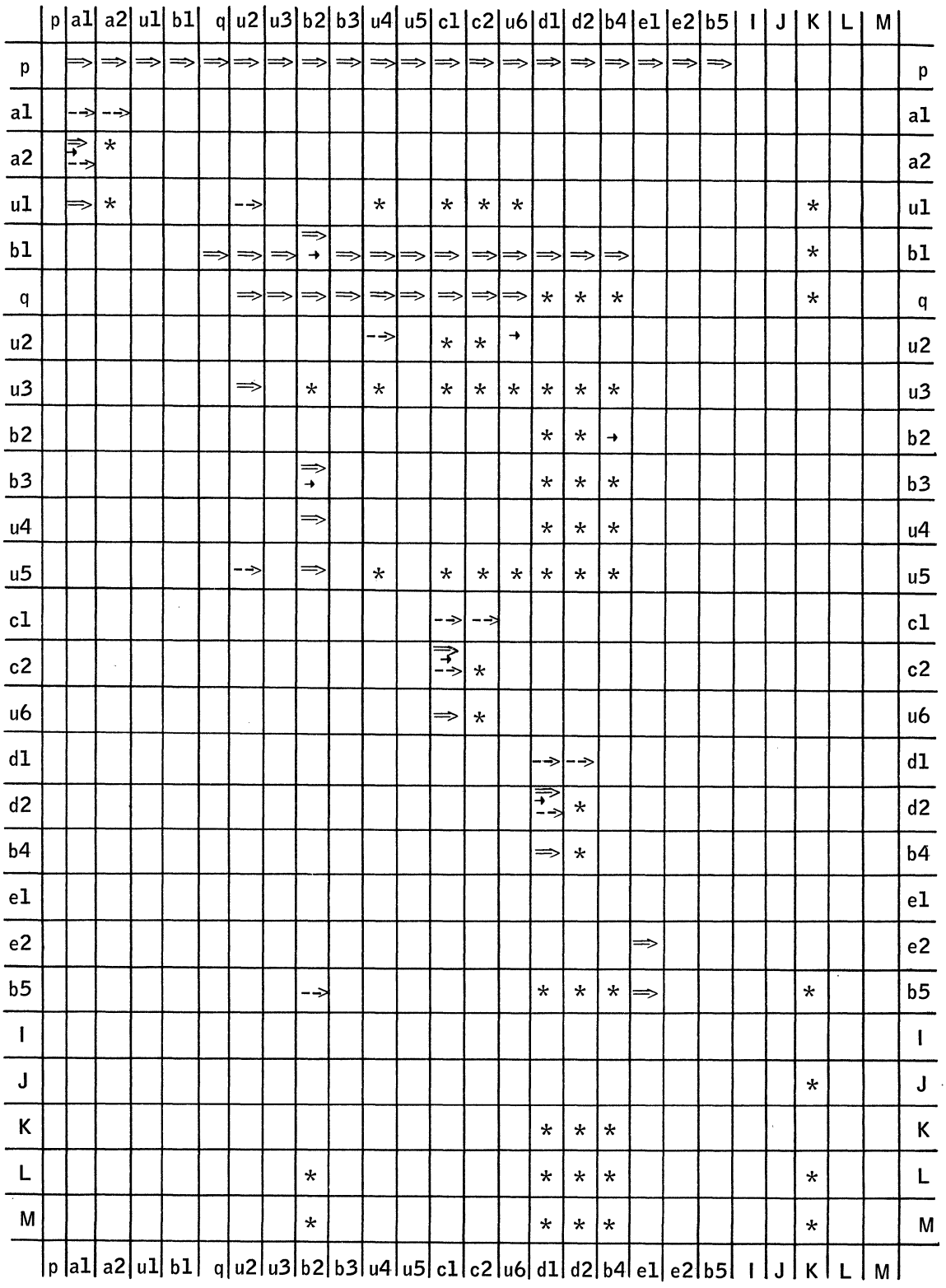


Figure 4

S3, S4. We obtain this from the specification of Program 1 in the obvious way. E.g., we have

$$\overline{a1} = \overline{a2} + \overline{u1} ** 2$$

$$T_{\overline{u3}}(k, i) = T_{\overline{u3}}(i, k) = (i, k-1) .$$

The reader can check that this specification satisfies L1 and L2. A simple-minded translation of this specification into an extended FORTRAN program gives Program 2. (A more efficient translation is possible.)

As this example shows, the problem of going from a valid program specification to a FORTRAN program can be difficult. However, it is always possible. A compiler would probably not do this, but would translate the program specification into an internal form suitable for generating code.

Program Mappings

Linear Program Mappings

Our basic idea is to transform a given program specification $\mathcal{S}$ into a new one $\overline{\mathcal{S}}$ which produces the same results, but has more parallel computation. The tree of $\overline{\mathcal{S}}$ will be obtained by splitting apart and rearranging loop nodes of $\mathcal{S}$. In the following definition, $\theta^{-1}(\alpha)$ is the set of nodes into which the loop node $\alpha \in \eta(\mathcal{S})$ is split.

Definition 6: Let $\mathcal{S}$ and $\overline{\mathcal{S}}$ be program trees. A linear tree mapping $\Omega : \mathcal{S} \to \overline{\mathcal{S}}$ consists of:

(1) A surjective mapping $\theta : \eta(\overline{\mathcal{S}}) \to \eta(\mathcal{S})$ such that:

(a) θ is a 1-1 correspondence between $\mathcal{O}(\overline{\mathcal{S}})$ and $\mathcal{O}(\mathcal{S})$.

(b) If $\overline{\alpha}$ is an ancestor of an occurrence node $\overline{f}$ of $\overline{\mathcal{S}}$, then $\theta(\overline{\alpha})$ is an ancestor of $\theta(\overline{f})$.

(c) For each $\overline{f} \in \mathcal{O}(\overline{\mathcal{S}}) : |\theta(\overline{f})| = |\overline{f}|$.

We denote $\theta(\overline{f})$ by f for each $\overline{f} \in \mathcal{O}(\overline{\mathcal{S}})$.

(2) For each $f \in \mathcal{O}(\mathcal{S})$, a linear 1-1 correspondence $\Omega_f : \mathbb{Z}_f \to \mathbb{Z}_{\overline{f}}$ satisfying the following condition: For any $f, g \in \mathcal{O}(\mathcal{S})$, the mapping $\Omega_{<f,g>} : \mathbb{Z}_{f \cap g} \to \mathbb{Z}_{\overline{f} \cap \overline{g}}$ defined by

$$\Omega_{<f, g>} = \Pi^{\overline{f}}_{\overline{f} \cap \overline{g}} \circ \Omega_f \circ \left(\Pi^{f}_{f \cap g}\right)^{-1}$$

is single-valued, and $\Omega_{<f, g>} = \Omega_{<g, f>}$.

As an example, let $\mathcal{S}$, $\overline{\mathcal{S}}$ be the trees of Figures 1 and 2, respectively. We define the linear tree mapping $\Omega : \mathcal{S} \to \overline{\mathcal{S}}$ as follows. Let $\theta(\overline{I}_1) = \ldots = \theta(\overline{I}_5) = I$, $\theta(\overline{J}) = J$, $\theta(\overline{a1}) = a1$, etc. Let Ω_f be the identity mapping unless f is a descendant of K, in which case let $\Omega_f(i, k) = (k, i)$. We then have

$$\Omega_{<u2, u3>}(i, k) = (k, i)$$
$$\Omega_{<u2, c1>}(i, k) = (k)$$
$$\Omega_{<a1, c1>}(i) = 0 .$$

Definition 7: Let $\mathcal{S}$, $\overline{\mathcal{S}}$ be program specifications. A linear program mapping $\Omega : \mathcal{S} \to \overline{\mathcal{S}}$ consists of a linear tree mapping Ω from the tree of $\mathcal{S}$ to that of $\overline{\mathcal{S}}$ such that:

(1) For each $f \in \mathcal{O}(\mathcal{S})$, f and $\overline{f}$ are occurrences of the same variable, and $T_f = T_{\overline{f}} \circ \Omega_f$.

(2) $f \Rightarrow g$ in $\mathcal{S}$ if and only if $\overline{f} \Rightarrow \overline{g}$ in $\overline{\mathcal{S}}$.

(3) Replacing each occurrence f by $\overline{f}$ in S4 of the specification $\mathcal{S}$ gives S4 of $\overline{\mathcal{S}}$.

Part 3 of the definition is as vague as our definition of S4 of the program specification. However, its meaning should be clear from our example. The mapping Ω defined above gives a linear program mapping from the specification of Program 1 to that of Program 2.

We say that two program specifications are equivalent if they produce the same output when run with the same legal input values. (Recall restriction 2.) Let $\Omega : \mathcal{S} \to \overline{\mathcal{S}}$ be a linear program mapping. To obtain the equivalence of $\mathcal{S}$ and $\overline{\mathcal{S}}$, we will assume that Ω satisfies the following condition.

EL. For each relevant pair of occurrences f, g in $\mathcal{S}$: if there exists an element $X \in < f, g >$ such that either (i) $X > \vec{0}$ or (ii) $X \sim \vec{0}$ and $f \to g$, then either (i) $\Omega_{<f, g>}(X) > \vec{0}$ or (ii) $\Omega_{<f, g>}(X) \sim \vec{0}$ and $\overline{f} \to \overline{g}$.

Theorem 8: Let $\mathcal{S}$ be a valid program specification, let $\overline{\mathcal{S}}$ be a specification satisfying L2, and let $\Omega : \mathcal{S} \to \overline{\mathcal{S}}$ be a linear program mapping satisfying EL. Then

(1) $\overline{\mathcal{S}}$ is a valid program specification.

(2) $\mathcal{S}$ and $\overline{\mathcal{S}}$ are equivalent.

(3) For each relevant occurrence pair f, g of $\mathcal{S}$: $<< \overline{f}, \overline{g} >> = \Omega_{<f, g>}(<< f, g >>)$.

Part 3 of the theorem allows us to choose $< \overline{f}, \overline{g} >$ to be $\Omega_{<f, g>}(< f, g >)$. The reader can check that Theorem 8 implies the equivalence of Programs 1 and 2.

Two Applications

We now describe two simple ways of obtaining a new program specification $\overline{\mathcal{S}}$ from a given specification $\mathcal{S}$. We leave it to the reader to verify that Theorem 8 implies the equivalence of $\mathcal{S}$ and $\overline{\mathcal{S}}$.

1. Interchange a tightly nested DO/DO SIM pair of nodes. E.g., let $\mathcal{S}$ be the specification of the L loop of Program 1. Then $\bar{\mathcal{S}}$ is the specification of the program:

```
   DO 30 M = 4, 50
   DO 30 SIM FOR ALL L ∈ {2, ..., 100}
30 E(I, L, M) = E(I-1, L+1, M-3) * B(I + 1) .
```

2. Split a single DO SIM node into several - one for each son. Applying this to the above specification $\bar{\mathcal{S}}$ then gives the specification $\bar{\bar{\mathcal{S}}}$ of the following program:

```
      DO 33 M = 4, 50
      DO 31 SIM FOR ALL L1 ∈ {2, ..., 100}
31    TMP1(L1) = E(I-1, L1+1, M-3)
      DO 32 SIM FOR ALL L2 ∈ {2, ..., 100}
32    TMP2(L2) = B(I+1)
      DO 33 SIM FOR ALL L3 ∈ {2, ..., 100}
33    E(I, L3, M) = TMP1(L3) * TMP2(L3) .
```

Note that this new version describes one way that the L loop of Program 1 might actually be executed by an array computer, TMP1 and TMP2 representing arithmetic registers.

In general, repeated application of these two rewriting procedures shows that DO SIM loops can always be rewritten in terms of vector assignment statements.

Coordinate Mappings

The mapping Ω_f for a linear program mapping Ω may be any linear 1-1 correspondence. This allows a generalization of the hyperplane method of [4], which will be done in a later paper. For the coordinate method, we restrict Ω_f to be a permutation of the coordinates.

To form the tree of $\bar{\mathcal{S}}$, we allow a DO node to be changed into one or more DO and/or DO SIM nodes, which may be moved lower in the tree. DO SIM nodes may not be changed, and no other rearrangement of nodes is allowed.

Definition 9: A linear program mapping $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ is a coordinate mapping if there is a subset $\mathcal{C}$ of the DO nodes of $\mathcal{S}$, called the set of changed nodes, satisfying the following conditions (where θ is as in Definition 6):

(1) For each $f \in \mathcal{O}(\bar{\mathcal{S}})$, let $\bar{I}^1, \ldots, \bar{I}^n, \bar{f}$ be a branch of $\bar{\mathcal{S}}$, let $I^j = \theta(\bar{I}^j)$, and let π be the permutation such that $I^{\pi(1)}, \ldots, I^{\pi(n)}, f$ is a branch of $\mathcal{S}$. Then

(a) If $j < k$ and $\pi(j) > \pi(k)$, then $I^{\pi(j)} \in \mathcal{C}$ and $I^{\pi(k)} \notin \mathcal{C}$.

(b) $\Omega_f(x^{\pi(1)}, \ldots, x^{\pi(n)}) = (x^1, \ldots, x^n)$.

(2) For each loop node α of $\mathcal{S}$ with $\alpha \notin \mathcal{C}$: $\theta^{-1}(\alpha)$ consists of a single node of the same type (DO or DO SIM) as α.

The mapping Ω defined above from the specification of Program 1 to that of Program 2 is a coordinate mapping with $\mathcal{C} = \{I\}$. Thus, only the I node of Program 1 is changed by Ω.

For a coordinate mapping $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$, we introduce the following condition.

EC. For each relevant pair of occurrences f, g in $\bar{\mathcal{S}}$:

(1) If $f \to g$, then $\bar{f} \to \bar{g}$.

(2) For each $X \in \langle f, g \rangle$ with $X > \vec{0}$, either

(a) $\Omega_{\langle f, g \rangle}(X) > \vec{0}$, or

(b) $\Omega_{\langle f, g \rangle}(X) \sim \vec{0}$ and $\bar{f} \to \bar{g}$.

The reader can verify that if a coordinate mapping satisfies EC, then it satisfies EL. Theorem 8 then gives the following result.

Theorem 10: Let $\mathcal{S}$ be a valid program specification and $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ a coordinate mapping satisfying EC. If $\bar{\mathcal{S}}$ satisfies L2, then it is a valid program specification and is equivalent to $\mathcal{S}$. For any relevant occurrence pair $\bar{f}, \bar{g}$ of $\bar{\mathcal{S}}$, we can let $\langle \bar{f}, \bar{g} \rangle$ equal $\Omega_{\langle f, g \rangle}(\langle f, g \rangle)$.

The following result shows that any coordinate mapping can be obtained from a sequence of coordinate mappings, each of which changes just one node.

Theorem 11: Let $\mathcal{S}$, $\bar{\mathcal{S}}$ be valid program specifications and $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ a coordinate mapping satisfying EC. Let α be any DO node of $\mathcal{S}$ which is changed by Ω such that no descendant of α is changed by Ω. Then there exists a valid program specification $\mathcal{S}'$ and coordinate mappings $\Omega' : \mathcal{S} \to \mathcal{S}'$ and $\Omega'' : \mathcal{S}' \to \bar{\mathcal{S}}$ satisfying EC such that Ω' changes only α.

The Coordinate Algorithm

We now describe the coordinate algorithm. Given a program specification $\mathcal{S}$ and a DO node I of $\mathcal{S}$, the coordinate algorithm generates a program specification $\bar{\mathcal{S}}$ and a coordinate mapping $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ such that (i) Ω changes only I and satisfies EC, and (ii) $\bar{\mathcal{S}}$ satisfies L2. Theorem 10 implies that $\bar{\mathcal{S}}$ is equivalent to $\mathcal{S}$. The algorithm can find any possible coordinate mapping satisfying (i) and (ii). By Theorem 11, we want to apply the algorithm repeatedly, starting from the innermost loop nodes.

We now describe, explain and illustrate the five major steps of the algorithm.

1. Let $>>$, $\approx$ be the relations $>$, $\sim$ on $\mathcal{S}$ which would result if the I node were changed to a DO SIM node. Define the relation $\dashrightarrow$ on $\mathcal{O}(I)$ and the subset $\mathcal{B}$ of $\eta(I)$ as follows. For each relevant pair of occurrences f, g in $\mathcal{O}(I)$, and each $X \in <f, g>$ with $X > \vec{0}$:

(a) If $X \approx \vec{0}$, then include the relation $f \dashrightarrow g$.

(b) If $X << \vec{0}$, then: for each descendant J of I which either equals or is an ancestor of $f \cap g$, if $\Pi_J^{f \cap g}(X) << \vec{0}$, then let J be an element of $\mathcal{B}$.

The relations $\dashrightarrow$ are the additional relations $\rightarrow$ required if we were to simply change the I node to a DO SIM node. The existence of an X satisfying (b) immediately precludes this possibility.

The nodes in $\mathcal{B}$ are "blocking nodes". This means that for each $J \in \mathcal{B}$, J cannot appear inside a DO SIM I node, and none of the nodes into which I is changed can be moved below J.

Applying step 1 to the I node of Program 1 gives the relations $\dashrightarrow$ shown in Figure 4. It finds $\mathcal{B} = \{J\}$. E.g., for the occurrence pair u2, u4 we have $(1, 0) \in <u2, u4>$, $(1, 0) > \vec{0}$ and $(1, 0) \approx \vec{0}$. Hence, (a) gives u2 $\dashrightarrow$ u4.

For the occurrence pair a1, a1: for any $i > 0$ we have $(i, 0) \in <a1, a1>$, $(i, 0) > \vec{0}$ and $(i, 0) \approx \vec{0}$. Hence (a) gives a1 $\dashrightarrow$ a1. For any $j < 0$, we have $(i, j) \in <a1, a1>$, $(i, j) > \vec{0}$ and $(i, j) << \vec{0}$. Since $\Pi_J^{a1 \cap a1}(i, j) = (i, j)$, part (b) places J in $\mathcal{B}$.

Note that if L were a DO node, then step 1 applied to e1, e2 would place L in $\mathcal{B}$. This shows why the algorithm should be applied to inner loops first.

2. Let $\Longrightarrow$ denote the relation on $\mathcal{O}(I)$ formed by the union of the relations $\Rightarrow$, $\rightarrow$ and $\dashrightarrow$, and let $\overset{*}{\Longrightarrow}$ denote its tree completion. Complete the set $\mathcal{B}$ as follows: for each node α of $\eta(I)$, if $\alpha \overset{*}{\Longrightarrow} \alpha$ then add α to $\mathcal{B}$.

For Program 1, every relation $\alpha \overset{*}{\Longrightarrow} \beta$ on $\eta(I)$ for which $\alpha \Longrightarrow \beta$ does not hold is indicated by an "*" in Figure 4. Step 2 then adds the following nodes to $\mathcal{B}$: a1, a2, c1, c2, d1, d2.

3. Choose a tree partition $\mathcal{P}$ of $\mathcal{T}(I)$ such that:

(a) Any non-terminal node of $\mathcal{P}$ consists of a single loop node of $\eta(I)$ which is not in $\mathcal{B}$.

(b) The relation induced on $\mathcal{O}(\mathcal{P})$ by $\Longrightarrow$ is tree consistent.

To obtain the maximum amount of parallelism, the partition $\mathcal{P}$ should be chosen to satisfy the following conditions as well:

(1) For any $\alpha \in \eta(I)$: if $\alpha \notin \mathcal{B}$ and $\eta(\alpha) \cap \mathcal{B} \neq \emptyset$, then $\{\alpha\}$ is one of the sets of $\mathcal{P}$.

(2) For any $\alpha, \beta \in \eta(I)$: if $\eta(\alpha) \cap \mathcal{B} = \emptyset$ and $\beta \in \mathcal{B}$, then α and β belong to different sets of $\mathcal{P}$.

There is an algorithm for choosing such a $\mathcal{P}$. Applying it to our example, and then combining the resulting sets {d1, d2} and {b4} into a single set, gives the tree partition $\mathcal{P}$ shown in Figure 5. In general, finding the best tree partition $\mathcal{P}$ is a major implementation problem.

4. Let $N_1, \ldots, N_m$ be the terminal nodes of $\mathcal{P}$. Define the program tree of $\bar{\mathcal{S}}$ as follows:

(a) $\eta(\bar{\mathcal{S}}) = \{\bar{\alpha} : \alpha \in \eta(\mathcal{S}), \alpha \neq I\} \cup \{\bar{I}_1, \ldots, \bar{I}_m\}$.

(b) For any nodes α, β of $\mathcal{S}$ not equal to I: $\bar{\alpha}$ is a descendant of $\bar{\beta}$ if and only if α is a descendant of β.

(c) For any node α of $\mathcal{S}$ not equal to I:

(i) $\bar{\alpha}$ is a descendant of $\bar{I}_j$ if and only if $\alpha \in N_j$.

(ii) $\bar{\alpha}$ is an ancestor of $\bar{I}_j$ if and only if either α is an ancestor of I or $\{\alpha\}$ is an ancestor of N_j.

This defines a tree in which the I node is split into the nodes $\bar{I}_1, \ldots, \bar{I}_m$. For each j, $\eta(\bar{I}_j) = \{\bar{\alpha} : \alpha \in N_j\} \cup \{\bar{I}_j\}$.

In our example, $\bar{\mathcal{S}}$ has the tree of Figure 2.

5. Define the relation $\rightarrow$ on $\bar{\mathcal{S}}$ as follows. For any occurrence $\bar{f}, \bar{g}$ in $\bar{\mathcal{S}}$, we let $\bar{f} \rightarrow \bar{g}$ if either:

(a) $f \rightarrow g$ in $\mathcal{S}$, or

(b) $f \dashrightarrow g$, $f \in N_i$, $g \in N_j$ and either (i) $i \neq j$, or (ii) $i = j$ and $\bar{I}_j$ is a DO SIM node.

In our example, step 5(b) gives the following relations: (i) $\overline{u1} \rightarrow \overline{u2}$, $\overline{b5} \rightarrow \overline{b2}$ and (ii) $\overline{u2} \rightarrow \overline{u4}$, $\overline{u5} \rightarrow \overline{u2}$.

The mapping θ of Definition 6 is defined by $\theta(\bar{\alpha}) = \alpha$ if $\alpha \neq I$, and $\theta(\bar{I}_j) = I$. Parts S3 and S4 of the specification $\bar{\mathcal{S}}$ and the coordinate mapping $\Omega : \mathcal{S} \rightarrow \bar{\mathcal{S}}$ are then defined in the obvious way.

The equivalence of $\mathcal{S}$ and $\bar{\mathcal{S}}$ is implied by Theorem 10 and Theorem 12 below. Note that Theorem 10 indicates how to compute the sets $<\bar{f}, \bar{g}>$ in order to apply the coordinate algorithm

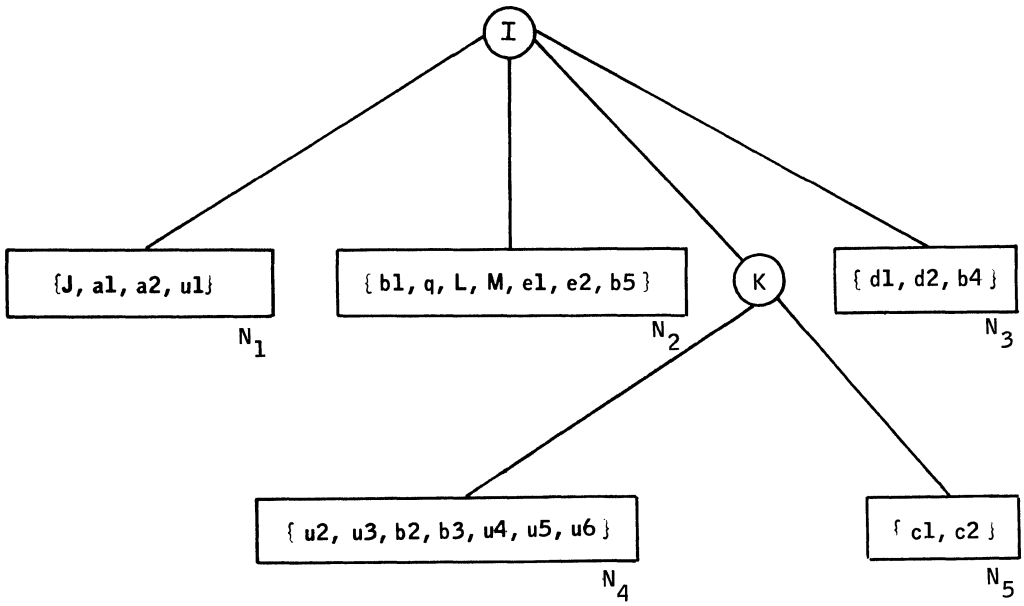


Figure 5

again to $\bar{\mathcal{S}}$. Theorems 11 and 13 show that the coordinate algorithm can be used to obtain any desired coordinate mapping.

Theorem 12: Let $\mathcal{S}$ be a valid program specification, I a DO node of $\mathcal{S}$, and let $\bar{\mathcal{S}}$ and $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ be constructed by the coordinate algorithm. Then $\bar{\mathcal{S}}$ satisfies L2, and Ω is a coordinate mapping which satisfies EC.

Theorem 13: Let $\mathcal{S}, \bar{\mathcal{S}}$ be valid program specifications, and let $\Omega : \mathcal{S} \to \bar{\mathcal{S}}$ be a coordinate mapping satisfying EC which changes only the node I. Then $\bar{\mathcal{S}}$ and Ω can be constructed from $\mathcal{S}$ by the coordinate algorithm.

Concluding Remarks

General DO Increments

To handle arbitrary constant DO increments we need only change the definition of the set $\ll f, g \gg$. Let $I^1, \ldots, I^k$ be the branch with $I^k = f \cap g$. Assume that for each j, the I^j loop is a DO loop of the following form:

$$\text{DO } I^j = \ell^j + \sum_{r=1}^{j-1} c_r^j * I^r \, , \, u^j \, , \, d^j$$

where the c_r^j and d^j are integer constants, and ℓ^j is any expression not involving the I^r. More general DO loops must be put into this form by changing the index variable. For purposes of the definition, replace a DO SIM I^j loop by any DO loop whose index set contains $\mathcal{I}_{I^j}$.

Now, define $\ll f, g \gg$ to be the set of all $(x^1, \ldots, x^k) \in \mathbb{Z}_{f \cap g}$ such that there exist $P \in \mathcal{I}_f$ and $Q \in \mathcal{I}_g$ with $(y^1, \ldots, y^k) = \Pi^g_{f \cap g}(Q) - \Pi^f_{f \cap g}(P)$ and

$$y^j = d^j x^j + \sum_{r=1}^{j-1} c_r^j y^r$$

for each j.

With this new definition, all of our results remain valid in the general case of arbitrary constant DO increments.

Further Refinements

Several refinements of the coordinate method to yield more parallelism are possible. For example, it is clear that the computation of U(I+1, J) ** 2 in statement 10 of Program 1 could be done inside a DO SIM I loop. This involves first splitting the J loop into two loops. In general, any node in the set $\mathcal{B}$ of the coordinate algorithm is a candidate for splitting. Such refinements will be described in [5].

Practical Problems

There are many practical problems to be solved in implementing the coordinate method for a real compiler. We list some of these below.

Although described separately, they are all closely related. The solutions of these problems will depend upon the particular parallel computer design.

Choice of the DO Node I. To maximize parallelism, by Theorem 11 we would apply the coordinate algorithm successively to each DO node, working up from the bottom of the tree. However, this may produce more parallelism than can be exploited by the computer. Some procedure is needed to choose the nodes to which the coordinate algorithm should be applied.

Choice of the Tree Partition $\mathcal{P}$. Maximizing the parallelism does not necessarily produce the best program. In our example, we assumed an algorithm clever enough not to put b4 into its own separate DO SIM I loop. However, we might have done better to further decrease the parallelism by putting $\overline{u6}$ in the $\overline{I_5}$ loop, eliminating the need for TMP5.

Translation of the Specification. It is necessary to translate the specification into either FORTRAN or some intermediate language from which the compiler can generate code. Our example indicates that conditional branches can always be handled by converting to conditional assignment statements. However, this will not always be the best procedure. Other improvements are also needed. E.g., in Program 2 we can replace TMP1 and TMP3 by new occurrences of P and Q .

Our example shows how complicated these problems can become. However, most real programs are simpler, and simple solutions will usually be good enough. For example, we might always choose $\mathcal{P}$ to consist of a single set. The coordinate algorithm would then simply try to rewrite the program with a single DO SIM I loop.

Conclusion

We have presented a method of detecting parallelism in sequential programs which generalizes several previous methods. It forms the basis of a sequential to parallel conversion phase of a compiler for a parallel array or vector computer. The techniques employed - particularly the use of the $< f, g >$ sets and the relation $\Rightarrow$ - should be applicable to other areas of program optimization.

References

[1] L. Lamport, Some Remarks on Parallel Programming, Massachusetts Computer Associates, CA-7211-2011, (November, 1972), 17 pp.

[2] Y. Muroaka, Parallelism Exposure and Exploitation in Programs, Ph.D. Thesis, University of Illinois, Urbana (1971), 236 pp.

[3] P.B. Schneck, "Automatic Recognition of Vector and Parallel Operations", Proc. of the ACM 25th Anniversary Conference (August, 1972), pp. 772-779.

[4] L. Lamport, "The Parallel Execution of DO Loops", to appear in the Comm. of the ACM.

[5] L. Lamport, The Complete Coordinate Method, Massachusetts Computer Associates, to appear.

[6] C.V. Ramamoorthy, and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs", AFIPS Proceedings, Fall Joint Computer Conference (1969), pp. 1-15.

MODELING FOR PARALLEL COMPUTATION: A CASE STUDY

J. L. Baer
Computer Science Group
University of Washington
Seattle, Washington 98195

Abstract

A methodology to build models for parallel computation for some special class of algorithms, namely compilation, is presented. The control graph component of the model is of the extended Petri Net form with switches and token absorbers. A detailed example of the use of the graph is given and some formal properties such as conservation of token and proper termination are proven.

I. Introduction.

In recent years, we have seen the emergence of numerous graph models for parallel computation [3]. Depending on the investigators' backgrounds (engineers, logicians, mathematicians), the objectives of the models have been varied (e.g. coherent design of modular parallel systems, correct flow of control in the execution of parallel algorithms, relations between sequential programs and their parallel representations, prediction of cost and performance of multiprocessors).

In this paper, we present (first in Section II) the criteria which have led us to select some particular node and arc primitives and graph properties for the modeling of a specific class of algorithms, namely parallel compilation. The choice of this test vehicle for our modeling methodology is motivated by the following observations. First, techniques to handle automatically the detection of parallelism are best suited for high-level languages and scientific applications and do not carry over well for compilation, which has most often been considered as a sequential process. Therefore some "human insight" appears necessary. Second, this will oblige us to try and uncover some parallelism in the compilation process through algorithm modification, changes in data structures, redundancy, etc. Finally, assuming the efficiency of multiprocessors at run-time, means must be found to use them efficiently at compile time.

In Section III, we apply this modeling methodology to an example taken from the compilation process. Different stages in the modeling are successively introduced. They show the importance and the need for the features introduced in Section II.

Section IV defines formally the graph model and some of its properties. It is shown how the latter can be derived through techniques resembling those used in the theory of formal languages.

II. Graph Primitives and Model Properties.

In the rest of this paper, we assume the reader's familiarity with the basic concepts of graph theory.

1. Places and Transitions of the Control Graph.

Like any other algorithmic process, compilation has three components: control, computation and data. We shall separate the modeling into a control graph (control of operators) and a data-flow graph (action of operators on data). In this paper, we investigate the control part.

The amount of interpretation that is present in a model depends mostly on the goals that one wants to achieve in the modeling process. If the primary objective is to describe specific algorithms or systems, then a total interpretation will be most convenient. Adam's Computation Graph [1] is such an example, and it can be regarded as a parallel programming language. On the other hand, if the derivation of general formal properties and the characterization of parallel algorithms are the main goals, then uninterpretation is necessary and schemata have to be introduced [6]. In our case, we are dealing with compilation considered as a class of algorithms and not with the modeling of a particular compiler. Hence, we will not choose total interpretation. At the same time, we wish to be able to retain some descriptive power and we have to rule out complete uninterpretation. Therefore our model is partially interpreted. Most of the interpretation takes place in the data graph, but some nodes/arcs of the control graph possess specific meanings.

One can view compilation as a general pipeline process, namely:

lexical analysis → syntax analysis →

code generation.

The unit of information flowing through the pipe can vary widely in size. For example, one could choose a subprogram, a block, a statement, a lexical or syntactical entity. Furthermore, each element of the pipe can be broken into a number of substages with appropriate latches. As we shall see in the next section, this pipeline concept can also exist at very fine levels of detail. Independently of the size of the unit of information, a "token machine" is appropriate to represent pipe-line flow. Therefore the control graph is based upon the Petri Net

concept [5,9]. The formal definition of the graph being given in Section IV, we recall here only that a Petri Net is composed of a set of transitions (corresponding to events and denoted by | in figures), a set of places (corresponding to the holding of conditions and denoted ○), and a set of directed arcs linking (input) places to transitions and transitions to (output) places. Places are able to receive tokens which mark the holding of conditions. A place without token is empty; otherwise it is full and the presence of a token will be shown by ⊙ in the figures. An event can occur (equivalently a transition can fire) if all its input places are full. After the firing, a token is removed from each input place and a token is added to each output place. We do not allow a place to be input and output to the same transition in order to "clarify" the description of holdings. Figure 1 shows a two-stage pipe-line process modeled by a Petri Net. When place 1 becomes full, stage 1 can be initiated through the firing of transition a. When stage 1 is completed, transition a_1 will fire and the latch will become full. Now transition b can fire, allowing the processing of stage 2 and the possibility for transition a to fire anew if place 1 becomes full again. Thus, stages 1 and 2 can be active simultaneously. This particular instance of a pipe-line is built in such a way that stage 1 has to wait for the initiation of the i^{th} computation of stage 2 before being able to initiate its own $(i+1)^{th}$ computation. In Figure 2, it is shown how a buffer can be introduced between the two stages (the buffer here being of size 2).

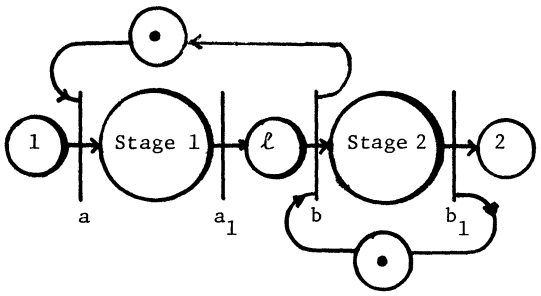


Figure 1. Modeling a Pipe-line Process with Petri Nets.

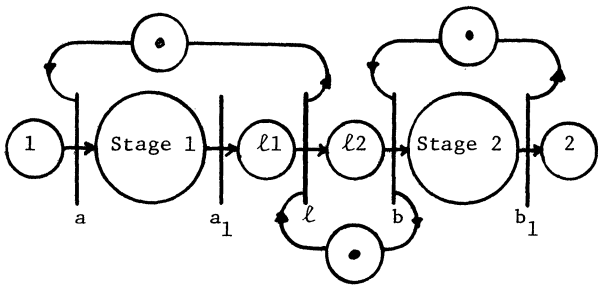
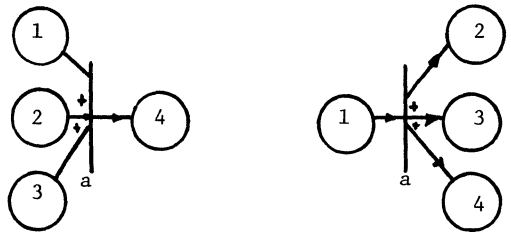


Figure 2. Increasing the size of Buffers between Stages.

Petri Nets in the above form are not easily amenable to represent predicates. Extensions using conjunctive logic as above and disjunctive logic, i.e. EOR [2], have been used to enhance the descriptive power of models [7] while, at the same time incurring no loss in some formal properties [4]. We follow the same approach here, denoting by + the presence of an EOR condition at either the input or output of a transition (cf. Figure 3). The conjunctive logic, i.e. AND condition, is the assumed default situation. We forbid mixed logics since we can always realize the desired boolean condition with the inclusion of appropriate "dummy" places and transitions with simple logic.

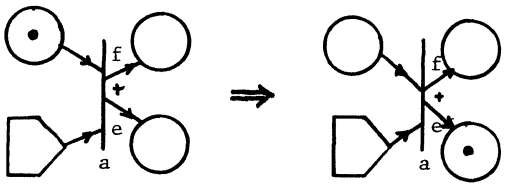


(a) Input Disjunctive Logic. Only one of the input places can be full. Then a can fire.

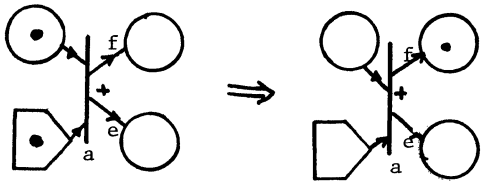
(b) Output Disjunctive Logic. After firing of a, only one of the output places will receive a token.

Figure 3. Disjunctive Logic.

Although the EOR and AND logics have sufficient properties to show the flow of control in algorithms, we introduce nevertheless a new type of place that we call switches. Switches bear some analogy with the construct of the same name found in programming languages and also with Nutt's resolution procedures [8]. However, their actions are purposely more restricted than these procedures so that their presence will not destroy formal properties of the model. As any other place, a switch can either be full or empty. The presence or absence of tokens in a switch does not influence the firing of the transition for which it is an input place; that is to say the conditions for firing are tested on the set of input places from which the switch has been removed. A transition which has a switch as one of its input places (and there cannot be more than one switch per transition) is necessarily of EOR-output logic with only two output places corresponding respectively to a full switch (branch f) and to an empty switch (branch e). Figure 4 illustrates these concepts with the switches denoted by ◗. As we shall see in the next section, switches allow flexibility and short cuts in the modeling of algorithms.



(a) Empty switch



(b) Full switch

Figure 4. Illustration of the Firing of a Transition with an Input Switch.

The transformation of a sequential program into parallel form might introduce some justifiable redundancy. However, processing which has become useless should not be allowed to be carried on and to tie up resources that the rest of the system might need. This point is illustrated by the following example. We want to search a linear table for a given key and two processors can be available. Hence, we desire processor 1 to start at the low end of the table with indices being incremented and processor 2 at the other end with decrementing indices. As soon as one of the processors has found a matching entry, both computations should be terminated. Moreover, if processor 1 was started first and had found the match and processor 2 was not yet initiated, it is evident that processor 2 should be prevented from performing a useless task, and vice-versa. In our modeling process, we use arcs which are token absorbers to represent this situation. Token absorbers also permit token conservation, a property needed, as we see below, for modeling pipe-line processes.

A token absorber is a multiarc with one head (a transition) and one or more tails (places). When the transition from where the head originates fires, tokens are removed from each of the full tail places. Figure 5 shows how this cancelling occurs for the previous example. Transitions a and b correspond to the comparison process; places C and F are the conditions of no-matching; E and H also correspond to no-match but in supplement they indicate that the ends of the (half) tables have been reached, and D and G correspond to a match. When either c or d fires, say c for example, tokens which were possibly present on B, F, G and H are removed. (In this example the presence of a token on one of these places implies the emptiness of the three others.) If we had a match on both processors, because of duplicate elements in the table, c and d could be terminating at the same time. By convention, similar to the realization of an interrupt scheme without priority, two transitions cannot fire simultaneously. If the two signal completion at the same time, one of them will be chosen arbitrarily as the first one to finish. Therefore I and J cannot hold tokens simultaneously (EOR-input logic at transition e), and K corresponds to a match in the search process. On the other hand, transition f will fire when both processors report no success.

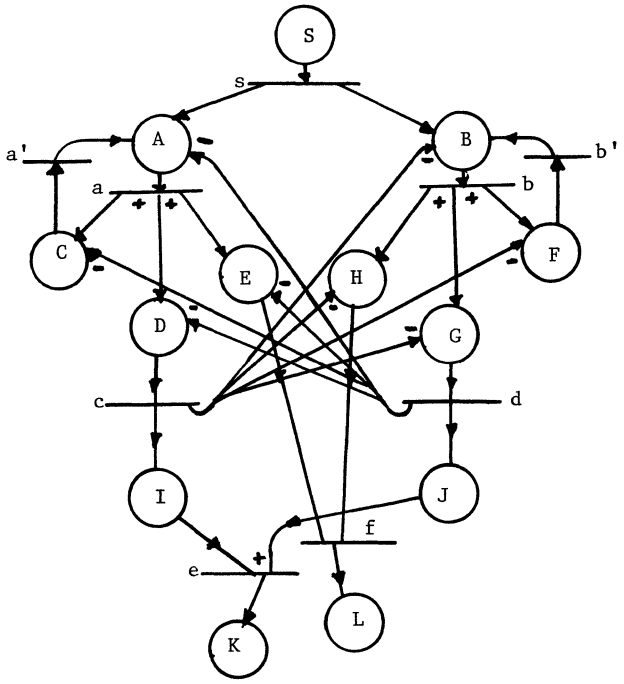


Figure 5. Illustration of the Use of Token Absorbers.

2. Execution Sequences and Properties of the Graph.

A control graph with places and transitions as above cannot describe a computational process per se. A meaning must be given to places and transitions. A first element of this semantic attachment is the data-flow graph associated with the control flow graph. A transition of the control can be linked to an operator in the data-flow graph. Each operator takes its inputs from a range of memory locations, performs a function and outputs values in a domain of memory locations. Furthermore, if the transition is of EOR-output logic, the operator indicates the output place on which a token is to be placed. An interpretation of the model consists of defining the data graph in terms of specific memory cells and their initial values, the operators' func-

tion, ranges and domains, as well as an initial marking of the control graph. The latter indicates which places are initially full, and the number of tokens on each place. In the sequel we will describe markings by the name of places which are full. The name of a place will occur as often as the number of tokens it holds. Starting with this initial marking, an execution sequence in the control graph is a sequence of transition firings. In the example of Figure 5, two out of the possible execution sequences, with an initial marking of a token on S, are:

s a b b' b a' a c e

s a a' b b' a b a' b' b a f .

After each transition firing, the graph is in a new state, or marking. The execution sequence can also be given in terms of sequence of markings. For the two above, we have respectively:

S,AB,BC,CF,BC,CF,AF,DF,I,K

S,AB,BC,AB,AF,AB,BC,CF,BF,AB,AH,EH,L .

If the execution sequence is finite, the last state reached is called a terminal marking.

The control graph should be constructed in such a way that given an initial marking M_0 and a set M of goals, i.e. terminal markings, all execution sequences starting with M_0 should be finite, reach one of the members of M and no other transition should be able to fire. This is akin to Gostelow's proper termination [4] and resembles strongly the acceptance of strings by a finite state automaton. For the example of Figure 5, an initial marking M_0 of a token on S and a set of goals $M = \{K,L\}$ yields proper termination for the graph, if one forbids infinite looping through transitions a,a' and b,b'. The rationale for this restriction will be explained in the following section. It is to be noticed that M_0 and M are at the discretion of the model builder, but proper termination is independent of the data graph and of the operators' functions. In supplement, since our model is oriented towards the representation of pipelining, another important property, namely the conservation of tokens, should be considered in conjunction with proper termination. More precisely, stages in the pipe-line have to be reusable after each activation. Therefore the initial and terminal markings should differ only by the presence of tokens on places which receive or deliver tokens from or to other stages. (The foremost stage as well as the last one constitute the environment or outside world [8,9]). The presence of token absorbers becomes very useful for the realization of this constraint.

An important criterion to judge the formal power of some graph models is the determinacy condition [3,6]. A model is determinate if the sequence of values associated with each memory cell is unique. In our case, determinacy involves the analysis of the data graph. But, because of the token absorbers, one can already see that determinacy cannot be achieved here. Therefore our goal will only be to obtain I/O determinacy, which is the property defined by the fact that for an initial set of input values all execution sequences will yield an identical set of output values. We shall not elaborate on this point, since the scope of this paper is restricted to the control structure of the model. In a similar manner, we define the I/O equivalence of two control graphs associated with a common data graph as the property of the two graphs to be determinate and to yield identical output values for identical initial values.

The programming of a large system should be modular. This structure has to be reflected in the model. Therefore, we need to be able to connect subgraphs. As seen above, the property of conservation of tokens allows the linkage of stages in the pipe-line as shown in Figure 1. Subroutine calling is modeled, in the control graph, by application of an ALGOL-like copy rule [1]. Although other techniques have been proposed [4], none of them applies to reentrant subroutines, the only kind with which one is concerned while writing compilers for multiprocessors.

Finally, one objective of the modeling is to ascertain the amount of parallelism that one could achieve. The first element of parallelism is in the pipe-lining process. The second is in the potential concurrency within each stage. Therefore, one characteristic of a stage is its maximum parallelism, i.e. the maximum number of transitions which are ready to fire simultaneously. For the example of Figure 5 this number is 2.

III. An Example of the Use of the Model.

To illustrate our approach as well as the use of the model, we consider the following example: During the lexical analysis phase of the compilation, it is known that either an identifier or a reserved word is going to be scanned as soon as the first character of a lexical entity has been recognized as a letter. The finite-state automaton, translated in extended Petri Net form, "flow-charting" this simple algorithm is shown in Figure 6.

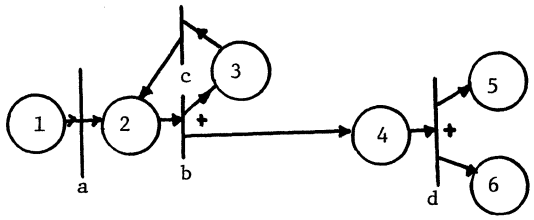


Figure 6. Lexical Analysis: the Obvious Approach.

The places have the following meanings:

1 : the first character is a letter

2 : ready to scan next character

3 : character is either a letter or a digit

4 : separator (i.e. end of lexical entity)

5 : lexical entity is an identifier

6 : lexical entity is a key word.

The transitions correspond to the following actions:

a,b : scan next character (example of the copy rule for reentrant subroutines)

c : dummy procedure to prevent place 2 from being both input and output to transition a.

d : look-up the table of reserved words.

If during the scanning a digit is encountered, the lexical entity cannot be a reserved word. Therefore, we introduce a new output place to transition b with the new meanings:

3 : character is a letter

7 : character is a digit.

If place 7 becomes full, then the lexical entity cannot be a reserved word and transition d should never be activated. This is accomplished in the graph model by the introduction of switch 9 which becomes full after transition e has fired and the latter fires every time a digit is recognized. When a separator is encountered and place 4 becomes full, transition f fires, taking the f branch if a digit had been encountered, the e branch otherwise. Only in this latter case does place 10 become full and allow transition d, i.e. the reserved word search, to be activated. However, two defects are apparent in this graph (Figure 7):

-Tokens are going to accumulate on switch 9. When place 5 is reached, the number of tokens left on 9, if any, is the number of digits encountered minus one. Hence, we need to either introduce token absorbers or change the logic of the graph.

-No parallelism is yet apparent.

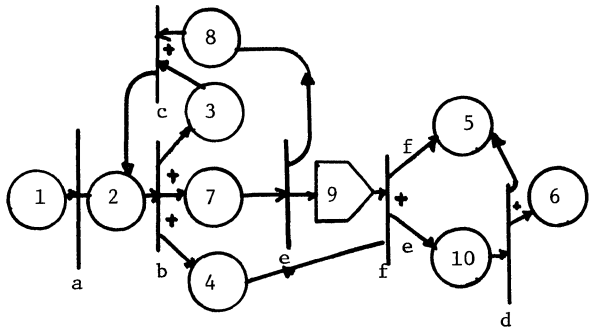


Figure 7. Introduction of a Switch.

Figure 8 shows how this latter weakness is taken care of. As soon as place 1 has been reached, the search in the reserved word table could be initiated if the latter were ordered. For example, we could find pointers to the beginning and end of the subtable corresponding to the letter scanned in place 1. To that effect, transition d is split into:

d : find begin and end pointers

and g : finish the search for the whole lexical entity,

with places 11 and 12 initiating these transitions. We could even refine further by allowing switch 9 to be an alternate output to transition d in case that there exists no reserved word starting with the letter scanned in place 1. However, our main point here is to show a possible concurrency between the scanning process (transition b) and the search process (transition d).

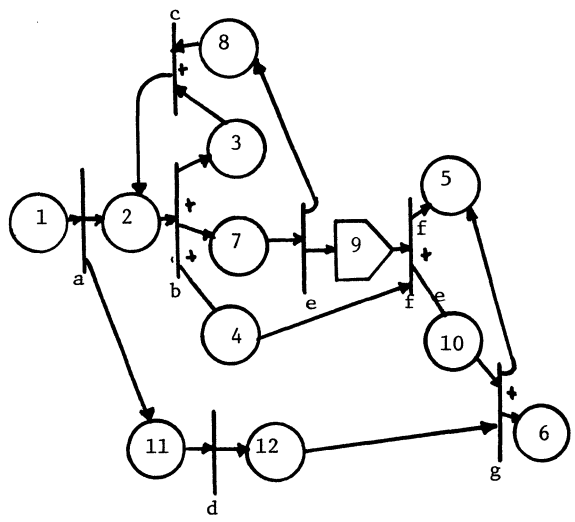


Figure 8. Introduction of parallelism with accumulation of tokens.

Finally, we have to "clean up" the graph so that it terminates properly and shows conservation of tokens (cf. Figure 9). In order to prevent accumulation of tokens on switch 9, two new places:

13 : the first digit is encountered

14 : a digit (not first) has been recognized

and a dummy transition h are placed between place 7 and switch 9. Another switch, 15, directs the output of h on either 13 or 14 and is filled at the beginning of the execution of the graph. i is a dummy transition between 14 and 8 introduced for the same reason as was c. With this logic, switch 9 will receive at most one token. If for a particular execution sequence switch 9 remains empty, then switch 15 will be full when place 6 is reached. Hence a token absorber is sent from transition g to switch 15. Finally, the firing of transition e removes any token present on either place 11 or place 12 via a multiarc token absorber. Thus, if a digit is encountered before the first table searching, this latter computation is cancelled.

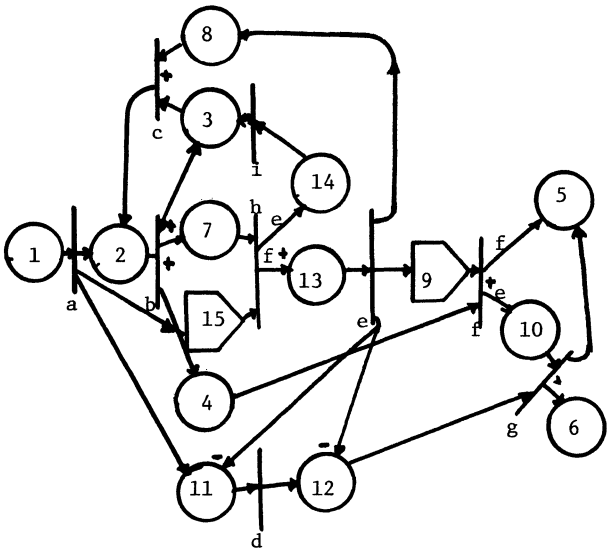


Figure 9. The final Graph for Lexical Analysis.

We have applied the same technique to other detailed algorithms with success. The "cleaning" of the graph is greatly facilitated by the procedure used to check for proper termination as presented in the next section.

IV. Formal Definitions and Properties.

1. Places, transitions and arcs.

The control graph is a triple $P = (X,A,C)$ where:

- $X = P \cup T$ is a finite set of vertices with
  - $P = \{p_1,p_2,\ldots,p_n\}$ being a finite set of places;
  - $T = \{t_1,t_2,\ldots,t_m\}$ being a finite set of transitions;
  - S a possibly empty proper subset of P is a set of switches.
- $A = I \cup O \cup N$ is a finite set of arcs with
  - $I = \{(p_i,t_j) \mid p_i \in P, t_j \in T\}$ being the input arc set and p_i being an input place to t_j
  - $O = \{(t_i,p_j) \mid t_i \in T, p_j \in P\}$ being the output arc set and p_j being an output place to t_i
  - $N = \{(t_i, [p_i,p_j,\ldots,p_k]) \mid t_i \in T, p_i,p_j,\ldots,p_k \in P\}$ being the token absorber set and $p_i,p_j\ldots$ being the cancelled places
- C is the control which associates with each transition a pair of logics, i.e. one of the possible combinations {(AND,AND),(AND,EOR), (EOR,AND),(EOR,EOR)}.

The following topological restrictions are imposed. If $(p_i,t_j) \in I$, then $(t_j,p_i) \notin O$. If $p_i \in S$ and $(p_i,t_j) \in I$, then no other switch is an input place to t_j, t_j is either of (AND,EOR) or (EOR,EOR) logic, and there are only two output places to t_j, the two output arcs leading from t_j being labeled respectively f and e.

2. Tokens, markings and firing expressions.

A place p_i is full if it holds at least one token. Otherwise it is empty. The set P and the number of tokens associated with each of its elements constitute a marking. Equivalently it can be represented by a multiset or bag [4].

The firing of a transition is controlled by the presence of tokens on its input places as well as by its logic. The latter also directs the outcome of the firing. The possible

situations are summarized below by firing expressions [4] for a transition a with the following conventions:

$p_{i_1}, p_{i_2}, \ldots, p_{i_n}$ input places to transition a

$p_{o_1}, p_{o_2}, \ldots, p_{o_m}$ output places to transition a

$p_{n_1}, p_{n_2}, \ldots, p_{n_q}$ cancelled places by transition a and $\overline{p_{n_1}}$ shows the absence of tokens

p_s is a switch and p_{o_e} and p_{o_f} the output places to transition a in that case

-(AND,AND) logic: $p_{i_1} p_{i_2} \ldots p_{i_n} \rightarrow p_{o_1} p_{o_2} \ldots p_{o_m} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

(AND,EOR) logic: $p_{i_1} p_{i_2} \ldots p_{i_n} \rightarrow p_{o_j} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

$j = 1,2,\ldots,m$

or if a switch is present

$p_{i_1} p_{i_2} \ldots p_{i_n} \overline{p}_s \rightarrow p_{o_e} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

$p_{i_1} p_{i_2} \ldots p_{i_n} p_s \rightarrow p_{o_f} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

(EOR,AND) logic: $p_{i_j} \rightarrow p_{o_1} p_{o_2} \ldots p_{o_m} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

$j = 1,2,\ldots,n$

(EOR,EOR) logic: $p_{i_j} \rightarrow p_{o_k} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$

$j = 1,2,\ldots,n$; $k = 1,2,\ldots,m$

or if a switch is present

$p_{i_j} \overline{p}_s \rightarrow p_{o_e} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$ $\quad j = 1,2,\ldots,n$

$p_{i_j} p_s \rightarrow p_{o_f} \overline{p}_{n_1} \overline{p}_{n_2} \ldots \overline{p}_{n_q}$ $\quad j = 1,2,\ldots,n$

The firing expressions for the graph of Figure 9 are shown in Figure 10.

$1 \rightarrow 2\ 11\ 15$	$7\ 15 \rightarrow 13$
$2 \rightarrow 3$	$7\ \overline{15} \rightarrow 14$
$2 \rightarrow 4$	$8 \rightarrow 2$
$2 \rightarrow 7$	$10\ 12 \rightarrow 5\ \overline{15}$
$3 \rightarrow 2$	$10\ 12 \rightarrow 6\ \overline{15}$
$4\ 9 \rightarrow 5$	$11 \rightarrow 12$
$4\ \overline{9} \rightarrow 10$	$13 \rightarrow 8\ 9\ \overline{11}\ \overline{12}$
	$14 \rightarrow 8$

Figure 10. Firing Expressions for the Graph of Figure 9.

3. Execution Sequences and Proper Termination.

A given marking indicates which transition(s), if any, can fire. After firing of one transition, a new marking is generated according to one of the firing expressions of the fired transition. Starting with an initial marking M_o, the sequence of the transition firings (or equivalently of the generated markings) is called an execution sequence. A marking from which no transition can fire is called a terminal marking. For a given P and M_o, we are interested in the finiteness of the execution sequences as well as their terminal markings. Thus, we also define a set of goal terminal markings M. We consider now graph executions as the triple (P,M_o,M). By definition, a graph execution has the property of token conservation if it is properly terminating (cf. below) and, if for every terminal marking $M_i \in M$, the set of full places is composed exclusively of places which either belong also to M_o - with the same number of tokens - or for which there is no transition admitting them as input places.

Before defining the concept of proper termination, we need to introduce two other properties of the graph, namely:

-A control graph P is k-safe if places can hold at most k tokens (a 1-safe graph is simply called safe).

-A graph is repetition-free if the domain of (data) operators associated with (AND, EOR) and (EOR,EOR) logic transitions is modified between two consecutive firings of the transition [4,6].

Now, a k-safe, repetition free graph execution (P,M_o,M) is properly terminating, if, for all interpretations and all execution sequences, if a terminal marking is reached, then:

-No place will ever receive more than k tokens;

-The terminal marking is a member of M.

-All members of M can be reached from M_o by some finite execution sequence.

Theorem: There exists an effective procedure to determine if the execution (P,M_o,M) of a k-safe, repetition free graph P is property terminating.

Proof: The proof is by construction and resembles that of the word problem in automata theory.

Let $|P|$ be the number of places in the graph. If P is k-safe, the number of allowable markings is bounded by $2^{k|P|}$. Consider now the state graph consisting of the $2^{k|P|}$ states (allowable markings) and of a dead state Δ corresponding to a tentative firing of a transition which would fill a place with more than k tokens. By convention no state can be reached from Δ. We construct the connections between different states as follows. We start with M_o and build the set M_o as the set of states which can be reached from M_o by the firing of one transition. We link M_o with members of M_o, each link (or arc) being labeled with the name of the transition. We repeat this process with each element of M_o yielding M'_1 and the labeled arcs between elements of M_o and M'_1. Then the set M_1 is defined by

$$M_1 = M'_1 - (M_o \cup \{M_o\}).$$

At step i, i.e. upon reaching M_{i-1}, the construction is as follows. Let M'_i be the set of markings which can be reached from an element of M_{i-1} by firing of a single transition. We connect elements of M_{i-1} with their appropriate successors in M'_i and determine M_i by

$$M_i = M'_i - (M_{i-1} \cup M_{i-2} \cup \ldots \cup M_o \cup \{M_o\}).$$

Since the number of states is finite, this procedure halts for some j, $j \leq 2^{k|P|}$, such that $M_j = \overline{\Phi}$. Now, let M' be the set of markings belonging to a M_i, $0 \leq i < j$ from which no other marking can be reached. The graph is properly terminating if and only if $M' = M$ and there exists a path from any state belonging to some M_i to at least one member of M. This latter condition is checked easily by some "successor" algorithm. The necessary condition is evident. If $M' \supset M$, then there exists a terminal marking which was not in the set of goals. If $M' \subset M$, then some goal can never be reached. If some state, reachable from M_o cannot reach a member of M, then the execution sequence cannot terminate. The sufficient condition stems from the repetition free property which states in effect that every possible path constructed above will be taken for some interpretation and execution sequence.

Q.E.D.

Figure 11 shows the state diagram and connections for the execution $(P,1,\{5,6\})$ of the safe graph P of Figure 9. States belonging to M' have been noted ◎.

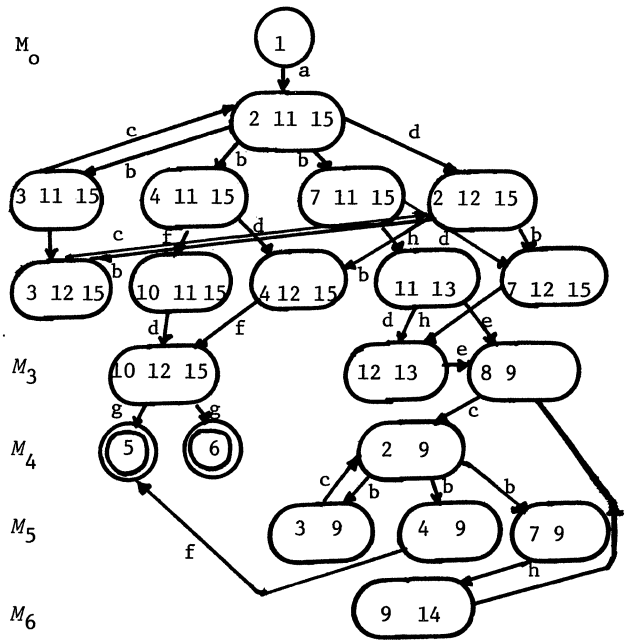


Figure 11. Procedure for Proper Termination.

It is worthwhile to remark that the above procedure allows also:

-The determination of the value k for k-safety; k is the maximum number of repetitions of a place in any marking.

-The determination of the number of transitions which can fire simultaneously, i.e. the maximum parallelism. We explain the process informally here by the example of Figure 11.

We write the execution sequences leading from M_o to the other reachable states as sequences of transition firings. We only consider paths between states which lead from a state in some set M_i to a state in some other set $M_j, j>i$. From a given marking, a boolean expression - sum of products - indicates the possible connections. When a product is present, it shows possible concurrency in the firing or two (or more) transitions. An execution sequence is made up of concatenations of such expressions. From the example of Figure 11, we have the development:

$E_o = a$ (unique transition firing possible)

$E_1 = ab \cup ad \cup a(b \cap d)$

(either b or d or both can fire from (2 11 15))

The concurrency (here $b \cap d$) is recognized when the firing expressions for two transitions have mutually exclusive left hand sides and these left

hand sides are subsets of the same marking. Now from E_1, we obtain E_2 by considering the transitions out from each state of M_1 and we expand appropriately the terms by recognizing which term (i.e. path) led to each marking. For example, (3 11 15) has been reached from ab and also from a(b ∩ d).

$$E_2 = ab(c \cup d \cup (c \cap d)) \cup ab(d \cup f \cup (d \cap f)) \cup ab(d \cup h \cup (d \cap h)) \cup adb \cup a(b(c \cup d \cup (c \cap d)) \cup a(b(d \cup f \cup (d \cap f)) \cap d) \cup a(b(d \cup h \cup (d \cap h)) \cap d) \cup a(b \cap db)$$

We consider next the union operators as distributive since they correspond to distinct paths. Thus, we expand E_2 while at the same time suppressing from it those expressions which correspond to paths leading uniquely to markings in M_o and M_1. It yields

$$E_2 = \cancel{abc} \cup abd \cup \cancel{ab(c \cap d)} \cup abf \cup ab(d \cap f) \cup abh \cup ab(d \cap h) \cup \cancel{adb} \cup \cancel{a(bc \cap d)} \cup \cancel{a(bd \cap d)} \cup \cancel{a(b(c \cap d) \cap d)} \cup a(bf \cap d) \cup \cancel{a(b(d \cap f) \cap d)} \cup a(bh \cap d) \cup \cancel{a(b(d \cap h) \cap d)} \cup \cancel{a(b \cap db)}$$

The terms which are crossed are cancelled for the following reasons:

- abc, a(bc ∩ d) and ab(c ∩ d) because they lead to markings belonging to M_o and M_1, or to the same marking as abd.
- adb because it leads to the same marking as abd.
- Terms of the form $\alpha(\beta x \cap \gamma x)$, where α, β and γ are subsequences, are expanded into $\alpha(\beta \cap \gamma x) \cup \alpha(\beta x \cap \gamma)$ since the firing of transition x cannot be duplicated. For example, a(b(d ∩ f) ∩ d) becomes a(bf ∩ d) ∪ ab(d ∩ f) and these last two terms are already present in E_2.

Hence, we obtain:

$$E_2 = abd \cup abf \cup abh \cup ab(d \cap f) \cup ab(d \cap h) \cup a(bf \cap d) \cup a(bh \cap d)$$

Continuing this process, we will have:

$$E_3 = abdf \cup abdh \cup abhe \cup abh(d \cap e) \cup ab(d \cap he)$$

$$E_4 = abdfg \cup abhec \cup abh(d \cap ec) \cup ab(d \cap hec)$$

$$E_5 = abhecb \cup abh(d \cap ecb) \cup ab(d \cap hecb)$$

$$E_6 = abhecbh \cup abh(d \cap ecbh) \cup ab(d \cap hecbh)$$

Now, the maximum parallelism MP is the maximum number of elements that are linked by an ∩ sign in any given term belonging to an E_i. In this example, MP = 2.

4. The Reduction Procedure.

The number of steps in the above procedure grows exponentially with the number of places in the graph. In a recent paper [4], the U. C. L. A. group has shown how this procedure could be shortened for a certain class of graphs of which our graph without token absorbers and switches is a subclass. This reduction procedure consists of a selective substitution of markings appearing on the lefthand side of the firing expression by the corresponding righthand sides. It can be shown that only a slight modification to the process is necessary to apply equally to the graphs we have defined above. The term reduction is used since the number of sets M_i as well as their cardinalities is diminished through the activation of the procedure. A few steps of the process applied to the graph of Figure 9 are shown in Figure 12(a) as well as the resultant state graph.

Reducing 2

1 → 3 11 15	49 → 5	8 → 7
1 → 4 11 15	49̄ → 10	10 12 → 5 $\overline{15}$
1 → 7 11 15	7 15 → 13	10 12 → 6 $\overline{15}$
3 → 3	7 $\overline{15}$ → 14	11 → 12
3 → 4	8 → 3	13 → 8 9 $\overline{11}$ $\overline{12}$
3 → 7	8 → 4	14 → 8

After reducing 3,8,11 and 14

1 → 4 12 15	7 15 → 13	10 12 → 6 $\overline{15}$
1 → 7 12 15	7 $\overline{15}$ → 4	13 → 4 9 $\overline{12}$
4 9 → 5	7 $\overline{15}$ → 7	13 → 7 9 $\overline{12}$
4 $\overline{9}$ → 10	10 12 → 5 $\overline{15}$	

Final reduction

1 → 4 12 15	7 15 → 13	10 12 → 5 $\overline{15}$
1 → 7 12 15	7 15 → 7 9 $\overline{12}$	10 12 → 6 $\overline{15}$
4 9 → 5	7 $\overline{15}$ → 4	
4 $\overline{9}$ → 10	7 $\overline{15}$ → 7	

Figure 12(a).

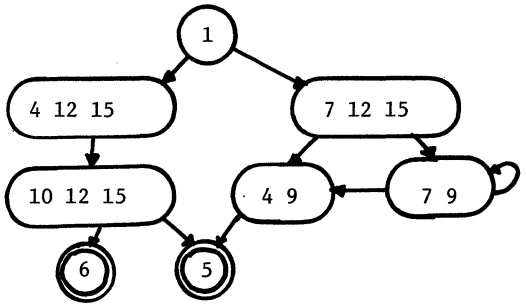


Figure 12. The Reduction Procedure and Terminal State Graph.

V. Conclusion.

In this paper we have presented a methodology for modeling parallel computations by graph models. We have shown what features are particularly appropriate for a specific application, namely compilation. Descriptive aspects (e.g. switches), efficiency aspects (e.g. token absorbers) and formal aspects (e.g. proper termination) were considered. This work is still in its early stages, and it might be necessary to introduce new features in the model. This will be done following the philosophy that we have put forward in this paper; that is, adjunctions to enhance the descriptive power of the model should not be made at the expense of destroying some formal properties, and, conversely, formal properties should not be sought if they do not relate to the application at hand.

References

[1] ADAMS, D. A., "A Computation Model with Data Flow Sequencing", PhD Dissertation, Computer Science Department, Stanford University, 1968.

[2] BAER, J. L., BOVET, D. P., and G. ESTRIN, "Legality and Other Properties of Graph Models of Computations", JACM, 17, 543-552, July 1970.

[3] BAER, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, 5,1, 1973.

[4] GOSTELOW, K., CERF, V., ESTRIN, G. and S. VOLANSKY, "Proper-Termination of Flow of Control in Programs Involving Concurrent Processes", SIGPLAN Notices, 7, 11, Nov 1972.

[5] HOLT, A. W. and F. COMMONER, "Information System Theory Project", Applied Data Research, Inc., 1969.

[6] KARP, R. M. and R. E. MILLER, "Parallel Program Schemata", Journal of Computer and System Sciences, 3, 147-195, May 1969.

[7] NOE, J. D., "A Petri Net Model of the CDC 6400", Proceedings of the ACM/SIGOPS Workshop on Systems Performance Evaluation, 362-378, Apr 1971.

[8] NUTT, G. J., "Evaluation Nets for Computer System Performance Analysis", Proceedings AFIPS 1972 FJCC, 41, 279-786, 1972.

[9] PATIL, S. S. and J. B. DENNIS, "The Description and Realization of Digital Systems", COMPCON 72, 223-227.

MEASUREMENTS OF PARALLELISM IN ORDINARY FORTRAN PROGRAMS*

David J. Kuck**, Paul P. Budnik**, Shyh-Ching Chen**,
Edward W. Davis, Jr.+, Joseph C-C. Han++, Paul W. Kraska+++,
Duncan H. Lawrie**, Yoichi Muraoka***, Richard E. Strebendt**,
and
Ross A. Towle**

Abstract -- This paper reports the results of a measurement of parallelism at the statement level in 86 FORTRAN programs. The amount of parallelism is determined by an analyzer program and is measured in terms of speedup over serial execution, the number of independent processors required, the efficiency of parallel execution and other measures.

The analysis techniques are only sketched in this paper, details may be found in the references. We also outline some machine organization assumptions.

Introduction

In the folklore of computer architecture there has been much speculation about the effectiveness of various machines in performing various computations. While it is quite easy to design a machine (or part of a machine) and study its effectiveness on this algorithm or that, it is rather difficult to make general effectiveness statements about classes of algorithms and machines. We are attempting to move in this direction and the present paper contains experimental measurements of a rather wide class of algorithms. Such measurements should be quite helpful in establishing some parameters of machine organization.

The organization of algorithms and programming for multioperation machines has been attacked in a great variety of ways in the past. These have included new programming languages, new numerical methods, and a variety of schemes to analyze programs to exploit some particular kind of simultaneous processing. The latter have included both hardware and software devices [5, 15,27,32,33,35,37]. Multiprogramming often formed an important part of these studies. None of them apparently tried to extract from one program as many operations as possible which could be executed simultaneously. For a comprehensive survey of many related results see [3].

This paper contains little detail about machine organization--we merely sketch some gross simplifying assumptions below. Then we outline the organization of our program analyzer and discuss its improvements over an earlier version [23]. A set of 86 FORTRAN decks totalling over 4000 cards has been analyzed and these are described in general terms. Then we present a number of tables and graphs which summarize our experiments. These include the possible speedup and number of processors required for the programs analyzed. Finally, we give some interpretations of these results. We conclude that some of the folklore has been in error, at least with respect to the kinds of programs we have measured.

Goals, Assumptions and Definitions

We are attempting to determine for computational algorithms, a set of parameters and their values, which would be useful in computer system design. A direct way of doing this is by the analysis of a large set of existing programs. We have chosen to analyze FORTRAN programs because of their wide availability and because their analysis is about as difficult as any high level language would be. A language with explicit array operations, for example, would be easier to analyze but would restrict our analysis domain to array type algorithms. We are attempting to show that a very wide class of algorithms can be found to possess a good deal of parallelism. The programs we have analyzed in many cases have no DO loops at all, for example, and most decks have less than 40 cards.

The experiments reported here are a substantial improvement over those reported in Kuck, et al [23] for several reasons. First, we have analyzed more than four times as many programs. These have been drawn from a wide variety of sources as described below and represent a wide variety of applications including a number of non-numerically oriented ones. Second, in an attempt to study the sensitivity of our analyses to memory assumptions we have made two sets of runs as described later (see Table III). Third, we have made several improvements to the analyzer itself. These include a new method of handling DO loops which we call the vertical scheme, and a new way

*This work was supported in part by NSF Grant GJ-36936.

**Department of Computer Science
University of Illinois
Urbana, Illinois 61801

+Goodyear Aerospace Corporation
Akron, Ohio

++Chung Shan Institute of Science and Technology
Hsien-Tien, Taipei, Taiwan

+++Control Data Corporation
Minneapolis, Minnesota

***Nippon Telephone and Telegraph Corporation
Tokyo, Japan

of treating IF statements within DO loops. These are discussed later in this paper.

In order to interpret the results of our analysis, we must make a number of assumptions about machine organization. These cannot be discussed in any detail here, but most of them are backed by detailed study as given in our references. Some are of course idealizations which we would not expect to hold in a real machine. Thus the results would be degraded to some extent. On the other hand, since our analyzer still is quite crude in several respects, we might expect these degradations to be offset by better speedups due to an improved analyzer.

We ignore I/O operations, assuming that they do not exist in FORTRAN. We also ignore control unit timing, assuming that instructions are always available for execution as required and are never held up by a control unit. We assume the availability of an arbitrary number of processors, all of which are capable of executing any of the four arithmetic operations (but not necessarily all the same one) at any time. Each of the arithmetic operations are assumed to take the same amount of time, which we call unit time.

Two nonstandard kinds of processing are assumed. To evaluate the supplied FORTRAN functions we rely on a fast scheme proposed in De Lugish [13]. This allows SIN(X), LOG(X), etc., to be evaluated in no more than a few multiply times. We also assume a many-way jump processor. Given predicate values corresponding to a tree of IF statements, this processor determines in unit time which program statement is the successor to the statement at the top of the tree. With up to 8 levels in such a tree, the gate count for the logic is modest [11,12].

We assume the existence of an instantaneously operating alignment network which serves to transmit data from memory to memory, from processor to processor, and between memories and processors. Based on studies of the requirements of real programs, some relatively inexpensive alignment networks have been designed [22,25]. We assume the memory can be cycled in unit time and that there are never any accessing conflicts in the memory. In Lawrie [25], and Budnik and Kuck [7], memories are shown that allow the accessing of most common array partitions without conflict. Hence, we believe that for a properly designed system, accessing and alignment conflicts can be a minor concern and that under conditions of steady state data flow, good system performance could be expected. For more discussion see [21].

Let the parallel computation time T_p be the time measured in unit times required to perform some calculation using p independent processors. We define the speedup over a uniprocessor as $S_p = \frac{T_1}{T_p}$, where T_1 is the serial computation time, and we define efficiency as $E_p = \frac{T_1}{pT_p} \leq 1$, which may be regarded as the quotient of S_p and the maximum possible speedup p. As explained in Kuck, et al [23], computation time may be saved with the sacrifice of performing extra operations. For example, a(b+cde) requires four operations and $T_p = 4$, whereas ab + acde requires five operations and $T_p = 3$. If O_p is the number of operations executed in performing some computation using p processors, then we call R_p the operation redundancy and let $R_p = \frac{O_p}{O_1} \geq 1$, where $O_1 = T_1$. Note that our definition of efficiency E_p is quite conservative since utilization of processors by redundant operations does not improve efficiency. Utilization is defined as $U_p = \frac{O_p}{pT_p} \leq 1$ where O_p is the number of operations which could have been performed. Using R_p, we can rewrite U_p as $U_p = \frac{R_p O_1}{pT_p} = \frac{R_p T_1}{pT_p}$, and by the definition of E_p we have $U_p = R_p E_p$. Thus, if an observer notices that all p processors are computing all of the time he may correctly conclude that the utilization is 1, but he may not conclude that the efficiency is 1 since the redundancy may be greater than 1.

Analysis Techniques

The analyzer accepts a FORTRAN program as input and breaks it into blocks of assignment statements, DO loop blocks, and IF tree blocks. During analysis each block is analyzed independently of the others and $\tilde{T}_1$, $\tilde{T}_p$, $\tilde{p}$, and $\tilde{O}_p$ are found for each block. Next, we find all traces through the program according to the IF and GO TO statements. We accumulate $\tilde{T}_1$, $\tilde{T}_p$, and $\tilde{O}_p$ for each block in each trace to give T_1, T_p, and O_p. The maximum $\tilde{p}$ found in any block in each trace becomes p. R_p, E_p, S_p, and U_p are calculated for each trace.

A block of assignment statements (BAS) is a sequence of assignment statements with no intervening statements of any kind. Statements in a BAS can be made independent of each other by a process called forward substitution. For example, A = B + C; R = A + D by forward substitution becomes A = B + C; R = B + C + D. By using the laws of associativity, commutativity, and distributivity as in Muraoka [30], and Han [16], we find the parallel parse tree for each statement. The algorithm of Hu [17] is applied to this forest of trees to give $\tilde{p}$. $\tilde{T}_p$ is the maximum height of the individual trees and $\tilde{O}_p$ is the sum of the operations in the forest. This collection of techniques is called tree-height reduction.

An IF tree block is a section of a FORTRAN program where the ratio of IF statements to assignment statements is larger than some predetermined threshold. An IF tree block is

transformed into (1) a BAS consisting of every set of assignment statements associated with each path through the decision tree, (2) a BAS consisting of the relational expressions of the IF statements which have been converted to assignment statements (i.e., $X \geq Y$ is converted to B = SIGN(X-Y)), and (3) a decision tree into which the IF statements are mapped. The tree-height reduction algorithm is then applied to (1) and (2) combined. Davis [11] shows how to evaluate an eight-level decision tree in unit time. Thus a dual purpose is served: speedup is increased by increasing the size of the BAS through combination of the smaller BAS's between IF statements, and a number of decision points in a program are reduced to a single multiple decision point which can be evaluated in parallel. The complete IF tree algorithm is described in Davis [11,12].

There are two types of parallelism in DO loop blocks which can be found most often in programs. First, the statement

```
      DO  1 I = 1, 3
1     A(I) = A(I+1) + B(I) + C(I) * D(I)
```

can be executed on a parallel machine in such a way that three statements, A(1) = A(2) + B(1) + C(1) * D(1), A(2) = A(3) + B(2) + C(2) * D(2) and A(3) = A(4) + B(3) + C(3) * D(3) are computed simultaneously by 3 different processors. Thus, we reduce the computation time from $T_1 = 9$ to $T_p = 3$. This type of parallelism (array operations) we will call Type-1 parallelism. If we apply tree-height reduction algorithms to each of these three statements, we can further reduce the computation time to 2 for a 6 processor machine.

The second type of parallelism lies in statements such as

```
(i)     DO 1 I = 1, 5
     1  P = P + A(I)

(ii)    DO 1 I = 1, 5
     1  A(I) = A(I-1) + B(I)
```

which both have a recurrence relation between the output and input variables. In example (ii), if we repeatedly substitute the left-hand side into the right-hand side and apply the tree-height reduction algorithms to each resultant statement, we can execute all 5 statements in parallel, e.g., A(1) = A(0) + B(1), A(2) = A(0) + B(1) + B(2), ..., A(5) = A(0) + B(1) + B(2) + B(3) + B(4) + B(5). This will decrease the computation time from 5 to 3. For a single variable recurrence relation as in example (i), we can use the same techniques and compute only the last output P = P + A(1) + A(2) ... + A(5) in 3 unit steps instead of 5. We will call this type of parallelism Type-0 parallelism.

In order to exploit these parallelisms in DO loops, an algorithm described in Kuck et al [23], called the horizontal scheme can be used to transform the original loop into an equivalent set of small loops in which these potential parallelisms will be more obvious. A modification of that algorithm called the vertical scheme has now been implemented. We illustrate these schemes with the following example:

```
      DO S6  I = 1, 3, 1
S1    T(I) = G(I) + M
S2    G(I) = T(I) + D(I)
S3    E(I) = F(I-1) + B(I)
S4    F(I) = E(I) + G(I)
S5    H(I) = A(I-1) + H(I-1)
S6    A(I) = C(I) + N
```

Due to limited space, we are unable to describe the details of the implementation [20,23], and we only give the essential parts of the vertical scheme:

a) Find the dependence graph among statements (Figure 1). In the dependence graph each node represents a statement; and a path from S_i to S_j indicates that an input variable of S_j during certain iterations has been updated by S_i during the same or an earlier iteration, according to the original execution order.

b) Separate the dependence graph into completely disconnected subgraphs, and arrange each subgraph as a DO loop in parallel as shown in Figure 2(a).

c) Apply the forward substitution technique to each subloop and the tree-height reduction algorithms to all resultant statements.

After this, the statements can be computed in parallel. The required $\tilde{p}$ and $\tilde{T}_p$ for each subloop resulting from use of the vertical and horizontal schemes are shown in Figure 2.

For this example, both schemes give us a nice speedup: $T_1 = 18$, $T_p = 6$ for the horizontal scheme and $T_p = 4$ for the vertical scheme. The latter has a better speedup but uses more processors. Note also that the total number of processors listed in Figure 2, which is 12 for the horizontal scheme and 32 for the vertical scheme, can be further reduced by Hu's algorithm [16,17] without increasing the number of steps, provided that some of the subtrees formed by the resultant statements are not completely filled, which is usually the case in most programs.

The basic difference between these two schemes is that the horizontal scheme tends to facilitate the extraction of Type-1 parallelism while the vertical scheme helps to find Type-0 parallelism. At present, we do not have a general method of determining, a priori which scheme will give a better result for a particular DO loop. Although many cases yield the same result using either scheme, in some cases a higher speedup (with or without lower efficiency of the use of processors) can be achieved using one scheme or the other.

IF and GO TO statements increase the number of possible paths through a DO loop and complicate the finding of the dependence graph, when there

are more than a few IF and GO TO statements. We find all possible paths and then analyze each path separately and call this strategy DO path. Thus, when p, O_p, etc., are being calculated for an entire program we treat each path through each DO loop separately rather than combining the numbers for each DO loop path into one set of numbers that describe the DO loop as a whole as was done in Kuck, et al [23].

Description of Analyzed Programs

A total of 86 FORTRAN programs with a total of 4115 statements were collected from various sources for this set of experiments. They have been divided into 7 classes; JAN, GPSS, DYS, NUME, TIME, EIS, and MISC. JAN is a subset of the programs described in Kuck, et al [23], and came from Conte [10], IBM [18], Lyness [26], and the University of Illinois subroutine library. GPSS contains the FORTRAN equivalents of the GPSS (General Purpose Simulation System) assembler listings [11] of 22 commonly used blocks. The DYSTAL (Dynamic Storage Allocation Language in FORTRAN [34]) library provided the programs in DYS. NUME contains standard numerical analysis programs from Astill, et al [2], Carnahan [8], and other sources. TIME is several time series programs from Simpson [36]. EIS is several programs from EISPACK (Eigensystem Package) which are FORTRAN versions of the eigenvalue programs in Wilkinson and Reinsh [38]. Waste paper baskets provided elementary Computer Science student programs, civil engineering programs, and Boolean synthesis programs. These and programs from Kunzi, et al [24], and Nakagawa and Lai [31] make up MISC. Table I and Figures 3 and 4 describe the 86 programs analyzed.

Results

The analyzer determines values of T_1, T_p, p, E_p, S_p, O_p, R_p, and U_p for each trace in a program. Each program was analyzed separately using both the horizontal and vertical schemes of DO loop analysis. The results of vertical or horizontal analysis were then used depending on which scheme gave better results for a particular program. The values of T_1, T_p, etc., for each trace were then averaged to determine an overall value for a program $\overline{T}_1$, $\overline{T}_p$, etc. Thus, we assume that each trace is equally likely, an assumption required by the absence of any dynamic program information. We feel this assumption yields conservative values since the more likely traces which are probably large and contain more parallelism are given equal weight with shorter, special case traces. Figures 5-9 are histograms showing $\overline{T}_1$, $\overline{T}_p$, $\overline{E}_p$, $\overline{S}_p$, $\overline{U}_p$, respectively, versus the number of programs.

The overall program values $\overline{T}_1$, $\overline{T}_p$, etc., are averaged to obtain ensemble values $\hat{T}_1$, $\hat{T}_p$, etc., for groups of related programs (see Table I). Table II shows these ensemble values for each group of programs as well as for all programs combined. As we can see, for a collection of ordinary programs we can expect speedups on the order of 10 using an average of 37 processors with an average efficiency of 35%. The use of averages in these circumstances is open to some criticism but we feel it is acceptable in view of the facts that the data are well distributed and the final averages are reasonably consistent, e.g., $\hat{p}\hat{E}_p \simeq \hat{S}_p$. Such anomalies as $\hat{T}_1/\hat{T}_p \geq \hat{S}_p$ can be attributed to occasional large T_p values in our raw data.

At this time we should stress several points about our source programs. First, four programs were discarded because they contained nonlinear recurrence relations and caused analysis difficulties. Their inclusion would have perturbed the results in a minor way, e.g., speedup would be low for these four. One was discarded because T_1 was so large that it effected the final averages too strongly ($\overline{T}_1 = 10953$). Second, all the programs were quite small (see Table I). Third, the number of loop iterations was 10 or less for all but one of the programs (where it was 20) whose data is shown in Table II. Higher speedups, efficiencies, etc., would be expected using a more realistic number of iterations (see Figures 10-12). Finally, we have not employed any multiprogramming, i.e., we do not account for the fact that more than one program can be executed simultaneously, (c.f. [11]). Multiprogramming would of course allow the use of more processors, in general.

For the results shown in Figures 5-12 and Table II, the analyzer accounts for memory stores but not for any memory fetches. The effect of accounting for fetches is shown in Table III, which lists the ensemble values for 65 programs run with and without memory fetches. As we can see, accounting for memory fetches improves our results. In reality, a lookahead control unit and overlapped processing and memory cycling would perhaps result in numbers somewhere between these values.

Finally, Figures 10-12 show $\hat{S}_p$ versus $\hat{T}_1$, $\hat{p}$ versus $\hat{T}_1$, and $\hat{S}_p$ versus $\hat{p}$, respectively, for each ensemble JAN, GPSS, etc., as well as for all programs. Additionally, we took the programs in JAN, GPSS, NUME, TIME, and EIS, which had DO loops with a variable limit (about 40% of the programs), and set the DO loop limits to 10. The resulting program values were averaged with all other programs in these groups and the final average plotted in Figures 10-12. The analyses were repeated using DO limits of 20, 30, and 40, and the resulting averages plotted as before.

Conclusions

Our experiments lead us to conclude that multioperation machines could be quite effective in most ordinary FORTRAN computations. Figure 12

shows that even the simplest sets of programs (GPSS, for example, has almost no DO loops) could be effectively executed using 16 processors. The overall average (ALL in Figure 12) as shown in Table III is 35 processors when all DO loop limits are set to 10 or less. As the programs become more complex, 128 or more processors would be effective in executing our programs. Note that for all of our studies, $T_1 \leq 10{,}000$, so most of the programs would be classed as short jobs in a typical computer center. In all cases, the average efficiency for each group of programs was no less than 30%. While we have not analyzed any decks with more than 100 cards, we would expect extrapolations of our results to hold. In fact, we obtained some decks by breaking larger ones at convenient points.

These numbers should be contrasted with current computer organizations. Presently, two to four simultaneous operation general purpose machines are quite common. Pipeline, parallel and associative machines which perform 8 to 64 simultaneous operations are emerging, but these are largely intended for special purpose use. Thus, we feel that our numbers indicate the possibility of perhaps an order of magnitude speedup increase over the current situation. Next we contrast our numbers with two commonly held beliefs about machine organization.

Let us assume that for $0 \leq \beta_k \leq 1$, $(1-\beta_k)$ of the serial execution time of a given program uses p processors, while β_k of it must be performed on $k \leq p$ processors. Then we may write

(assuming $0_1 = 0_k + 0_p$): $T_p = \frac{\beta_k T_1}{k} + (1-\beta_k)\frac{T_1}{p}$,

and $E_p = \frac{T_1}{\frac{p}{k}\beta_k T_1 + (1-\beta_k)T_1} = \frac{1}{1+\beta_k(\frac{p}{k}-1)}$. For example, if $k = 1$, $p = 33$, and $\beta_1 = \frac{1}{16}$, then we have $E_{33} = \frac{1}{3}$. This means that to achieve $E_{33} = \frac{1}{3}$, 15/16 of T_1 must be executed using all 33 processors, while only 1/16 of T_1 may use a single processor. While $E_{33} = 1/3$ is typical of our results (see Figure 7), it would be extremely surprising to learn that 15/16 of T_1 could be executed using fully 33 processors. This kind of observation led Amdahl [1] and others [9,35] to conclude that computers capable of executing a large number of simultaneous operations would not be reasonably efficient, or to paraphrase them "Ordinary programs have too much serial code to be executed efficiently on a multioperation processor".

Such arguments have an invalidating flaw, however, in that they assume $k = 1$ in the above efficiency expression. Evidently, no one who repeated this argument ever considered the obvious fact that k will generally assume many integer values in the course of executing most programs. Thus, the expression for E_p which we gave above must be generalized to allow all values of k up to some maximum.

The technique used in our experiments for computing E_p is such a generalization. For some execution trace through a program, at each time step i, some number of processors k(i) will be required. If the maximum number of processors required on any step is p, we compute the efficiency for any trace as

$$E_p = \frac{\sum_{i=1}^{T_p} k(i)}{pR_pT_p}$$, assuming p processors are available. Apparently no previous attempt to quantify the parameters discussed above has been successful for a wide class of programs. Besides Kuck, et al [23], the only other published results are by Baer and Estrin [4], who report on five programs.

Another commonly held opinion, which has been mentioned by Minsky [29] is that speedup S_p is proportional to $\log_2 p$. Flynn [14] further discusses this, assuming that all the operations simultaneously executed are identical. This may be interpreted to hold 1) over many programs of different characteristics, 2) for one fixed program with a varying number of processors, or 3) for one program with varying DO loop limits. That the above is false under interpretation 1 for our analyses is obvious from Figure 12. Similarly, it is false under interpretation 2 as the number of processors is varied between 1 and some number as plotted in Figure 12. As p is increased still farther, the speedup and efficiency may be regarded as constant or the speedup may be increased at a decreasing rate together with a decreasing efficiency. Eventually, as p becomes arbitrarily large, the speedup becomes constant and in some region the curve may appear logarithmic. Under interpretation 3, there are many possibilities--programs with multiply nested DO loops may have speedups which grow much faster than linearly, and programs without DO loops of course do not change at all. Rather than discuss the above any further, we turn to the following.

Abstractly, it seems of more interest to relate speedup to T_1 than to p. Based on our data, we offer the:

Observation For many ordinary FORTRAN programs (with $T_1 \leq 10{,}000$), we can find p such that

1) $T_p = \alpha \log_2 T_1$ for $2 \leq \alpha \leq 10$,

and 2) $p \leq \frac{T_1}{.6 \log_2 T_1}$,

such that 3) $S_p \geq \frac{T_1}{10 \log_2 T_1}$ and $E_p \geq .3$.

The average α value in our experiments was about 9. However, the median value was less than 4,

since there were several very large values.

A complete theoretical explanation of this observation would be difficult, at present. But the following remarks are relevant. Theoretical speedups of $O(T_1/\log_2 T_1)$ for various classes of arithmetic expressions have been proved in Brent, et al [6], Maruyama [28], and Kogge and Stone [19]. Many DO loops yield an array of expressions to be evaluated simultaneously and this leads to speedups greater than $O\left(\frac{T_1}{\log_2 T_1}\right)$. Other parts of programs use fewer processors than the maximum and yield lower speedups. However, we have typically observed speedups of two to eight in programs dominated by blocks of assignment statements and IF statements, assuming the IF tree logic of Davis [11].

In practice one is generally given a set of programs to be executed. If the problem is to design a machine, i.e., choose p, then the above approach is a reasonable one. Alternatively, the problem may be to compile them for a given number of processors. If the number available is less than that determined by the above analysis, the speedup will be decreased accordingly. If the number to be used is greater than that determined above, one must face reduced efficiency or multiprogramming the machine.

We gain several advantages by the analysis of programs in high-level languages. First, more of a program can be scanned by a compiler than by lookahead logic in a control unit, so more global information is available. Second, in FORTRAN, an IF and a DO statement, for example, are easily distinguishable, but at run time the assembly language versions of these may be quite difficult to distinguish. Third, a program can be transformed in major ways at compile time so it may be run on a particular machine organization. All of these lead to simpler, faster control units at the expense of more complex compilation.

Finally, we point out that a number of realities of actual machines have been glossed over in this paper. We mentioned a number of these in our section on Goals, Assumptions and Definitions. A more detailed discussion of the philosophy of our analysis work may be found in [21,23].

Acknowledgment

We gratefully acknowledge the contributions of C. Cartegini, J. Claggett, W. Hackmann, D. Romine, W. Tao, and D. Wills, who provided programming assistance. This research was supported by the National Science Foundation, Grant No. GJ-36936 and by NASA, Contract No. NAS2-6724.

References

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conference Proceedings, Vol. 30 (1967), pp. 483-485.

[2] K. N. Astill and B. W. Arden, Numerical Algorithms: Origins and Applications, Addison-Wesley, (1970).

[3] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, Vol. 5, No. 1 (March 1973), pp. 31-80.

[4] J. L. Baer and G. Estrin, "Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations," IEEE Transactions on Computers, Vol. C-18, No. 11 (Nov. 1969), pp. 1012-1014.

[5] H. W. Bingham, E. W. Riegel and D. A. Fisher, "Control Mechanisms for Parallelism in Programs," Burroughs Corp., Paoli, Pa., ECOM-02463-7 (1968).

[6] R. Brent, D. Kuck and K. Maruyama, "The Parallel Evaluation of Arithmetic Expressions Without Division," IEEE Transactions on Computers, Vol. C-22, No. 5 (May 1973), pp. 532-534.

[7] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Transactions on Computers, Vol. C-20, No. 12 (Dec. 1971), pp. 1566-1569.

[8] B. Carnahan, H. A. Luther and J. O. Wilkes, Applied Numerical Methods, John Wiley and Sons, (1969).

[9] T. C. Chen, "Unconventional Superspeed Computer Systems," AFIPS Conference Proceedings, Vol. 38 (1971), pp. 365-71.

[10] S. D. Conte, Elementary Numerical Analysis, McGraw-Hill (1965).

[11] E. W. Davis, Jr., A Multiprocessor for Simulation Applications, Ph.D. thesis, Dept. of Computer Science, University of Ill., Urbana, Rep. No. 527 (June 1972).

[12] E. W. Davis, Jr., "Concurrent Processing of Conditional Jump Trees," Compcon 72, IEEE Computer Society Conference Proceedings, (Sept. 1972), pp. 279-281.

[13] B. De Lugish, A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer, Ph.D. thesis, Dept. of Computer Science, University of Ill., Urbana, Rep. No. 399 (June 1970).

[14] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, No. 9 (Sept. 1972), pp. 948-960.

[15] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," IEEE Transactions on Computers, Vol. C-21, No. 12 (Dec. 1972), pp. 1411-1415.

[16] J. Han, Tree Height Reduction for Parallel Processing of Blocks of FORTRAN Assignment Statements, M.S. thesis, Dept. of Computer Science, University of Ill., Urbana, Rep.

No. 493, (Feb. 1972).

[17] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," Operations Research, Vol. 9 (Nov.-Dec. 1961), pp. 841-848.

[18] IBM, "System/360 Scientific Subroutine Package Version III," GH 20-0205-4 (Aug. 1970).

[19] P. Kogge and H. S. Stone, A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, Dig. Systems Lab., Stanford, Calif., Rep. No. 25 (Mar. 1972).

[20] D. J. Kuck, NASA Final Report, Contract NAS2-6724 (Dec. 1972).

[21] D. J. Kuck, "Multioperation Machine Computational Complexity," Proceedings of Symposium on Complexity of Sequential and Parallel Numerical Algorithms, invited paper, May 1973, to be published by Academic Press.

[22] D. J. Kuck, D. H. Lawrie and Y. Muraoka, "Interconnection Networks for Processors and Memories in Large Systems," Compcon 72, IEEE Computer Society Conference Proceedings (Sept. 1972), pp. 131-134.

[23] D. J. Kuck, Y. Muraoka and S-C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-up," IEEE Transactions on Computers, Vol. C-21, No. 12 (Dec. 1972).

[24] H. P. Kunzi, H. G. Tzschach and C. A. Zehnder, Numerical Methods of Mathematical Optimization with ALGOL and FORTRAN, (Werner C. Rheinboldt, tran.), Academic Press (1971).

[25] D. H. Lawrie, Memory-Processor Connection Networks, Ph.D. thesis, Dept. of Computer Science, University of Ill., Urbana, Rep. No. 557 (Feb. 1973).

[26] J. N. Lyness, "Algorithm 379 SQUANK (Simpson Quadrature Used Adaptively--Noise Killed)," Communications of the ACM, Vol. 13, No. 4 (April 1970).

[27] D. Martin and G. Estrin, "Experiments on Models of Computations and Systems," IEEE Trans. Electron. Comput., Vol. EC-16 (Feb. 1967), pp. 59-69.

[28] K. Maruyama, "On the Parallel Evaluation of Polynomials," IEEE Transactions on Computers, Vol. C-22, No. 1 (Jan. 1973), pp. 2-5.

[29] M. Minsky, "Form and Content in Computer Science," 1970 ACM Turing Lecture, Journal of the ACM, Vol. 17, No. 2 (1970), pp. 197-215.

[30] Y. Muraoka, Parallelism Exposure and Exploitation in Programs, Ph.D. thesis, Dept. of Computer Science, University of Ill., Urbana, Rep. No. 424 (Feb. 1971).

[31] T. Nakagawa and H. Lai, Reference Manual of FORTRAN Program ILLOD--(NOR-B) for Optimal NOR Networks, Dept. of Computer Science, University of Ill., Urbana, Rep. No. 488 (Dec. 1971).

[32] C. V. Ramamoorthy and M. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," AFIPS Conference Proceedings, Vol. 35, AFIPS Press (Fall 1969), pp. 1-15.

[33] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," IEEE Transactions on Computers, Vol. C-21, No. 12 (Dec. 1972), pp. 1405-1411.

[34] J. M. Sakoda, "DYSTAL Manual," Dept. of Sociology and Anthropology, Brown Univ., Providence, R. I. (1965).

[35] D. Senzig, "Observations on High-Performance Machines," AFIPS Conference Proceedings, Vol. 31 (1967), pp. 791-799.

[36] S. M. Simpson, Jr., Time Series Computations in FORTRAN and FAP, Vol. 1, Addison-Wesley (1966).

[37] G. Tjaden and M. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE Transactions on Computers, Vol. C-19 (Oct. 1970), pp. 889-895.

[38] J. H. Wilkinson and C. Reinsh, Linear Algebra, F. L. Bauer, ed., Springer Verlag (1971).

	JAN	GPSS	DYS	NUME	TIME	EIS	MISC
Av. # BAS Outside DO	12.2	16.7	8.5	5.4	8.4	1.13	5.0
Av. # BAS Inside DO	3.7	0.3	2.5	4.3	3.5	6.6	4.4
Av. # DO Loops	1.8	0.3	1.5	1.9	3.1	1.5	2.4
Av. # Nested DOs	1.0	0.0	0.5	1.2	0.4	2.8	0.6
Av. # IFs	6.9	11.3	4.9	3.4	4.1	3.0	3.6
Av. # IF Trees	1.5	1.9	1.3	0.8	1.5	0.0	0.8
Av. # Traces	75.5	36.1	21.4	29.7	12.1	5.5	24.6
Av. # Statements	72.5	61.9	44.9	33.4	45.0	32.2	32.8
Total # Programs	12	22	10	10	8	8	16

Table I. Characteristics of Analyzed Programs

	$\hat{T}_1$	$\hat{T}_p$	$\hat{p}$	$\hat{E}_p$	$\hat{S}_p$	$\hat{O}_p$	$\hat{R}_p$	$\hat{U}_p$
JAN	357	48	62	.37	12.1	654	2.3	.43
GPSS	30	12	14	.30	3.2	67	2.5	.54
DYS	224	146	19	.47	4.9	1969	2.4	.47
NUME	654	77	51	.35	20.7	676	1.2	.38
TIME	174	22	23	.42	7.1	207	1.5	.46
EIS	896	208	82	.32	22.6	2292	2.8	.34
MISC	274	32	39	.32	8.4	486	2.1	.41
ALL	310	63	37	.35	9.8	739	2.2	.45

Table II. Average Measured Values for Seven Program Groups and for all Programs Combined

Table III. Comparison of Average Measured Values With and Without Memory Fetches

	Without Memory Fetches	With Memory Fetches
$\hat{T}_1$	678	967
$\hat{T}_p$	148	164
$\hat{p}$	35	35
$\hat{E}_p$	.33	.41
$\hat{S}_p$	9.2	11.1
$\hat{O}_p$	1212	1443
$\hat{R}_p$	2.4	1.9
$\hat{U}_p$	.45	.44

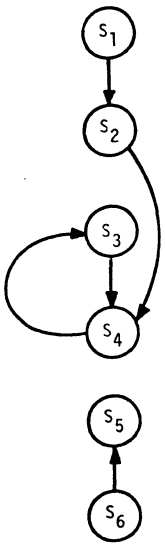
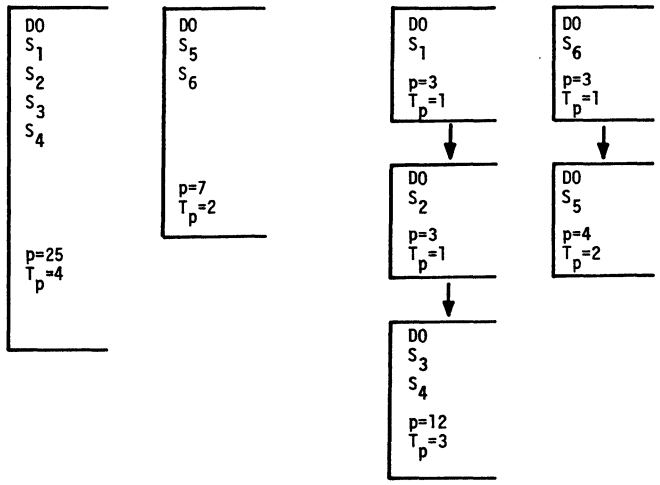


Figure 1. Dependence Graph



Figrue 2. Decomposition of DO Loops

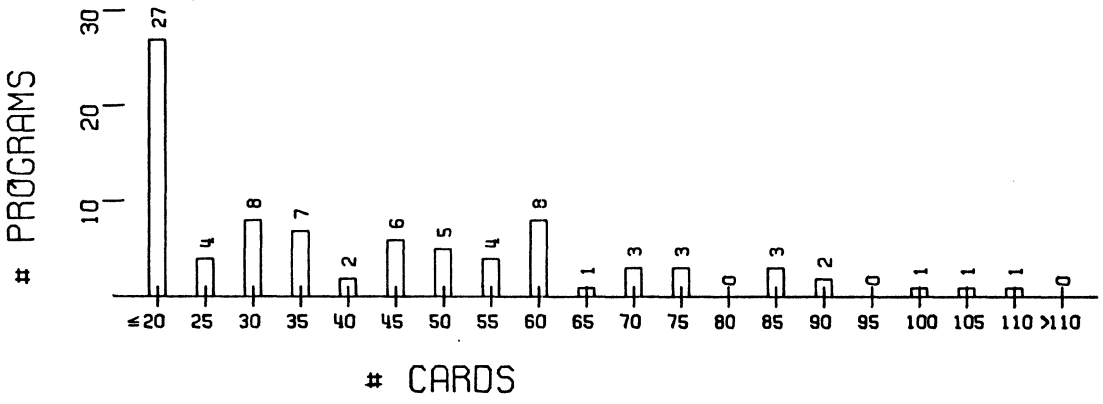


Figure 3. Number of Programs Versus Number of Cards in Program

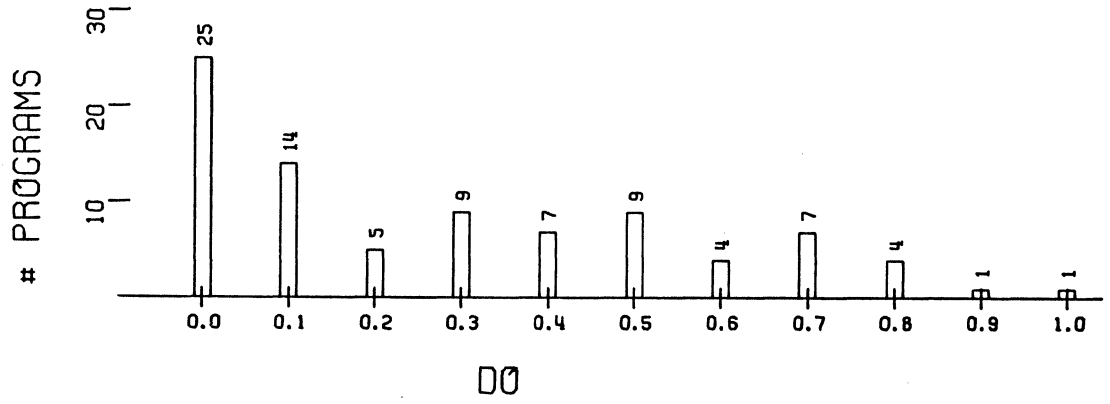


Figure 4. Number of Programs Versus Fraction DO Loops

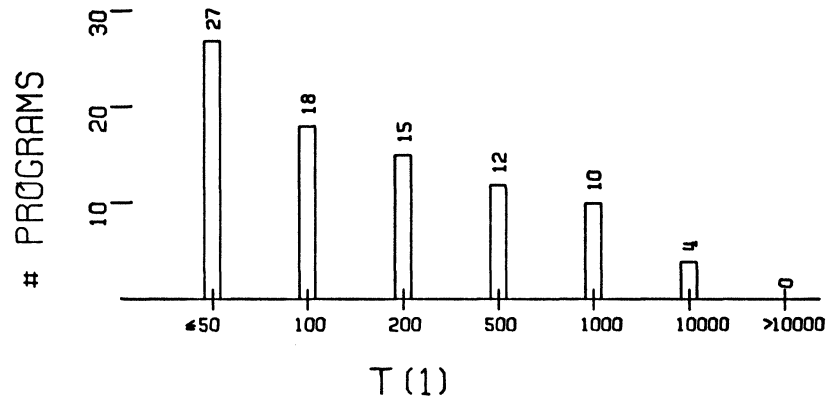


Figure 5. Number of Programs Versus $\bar{T}_1$

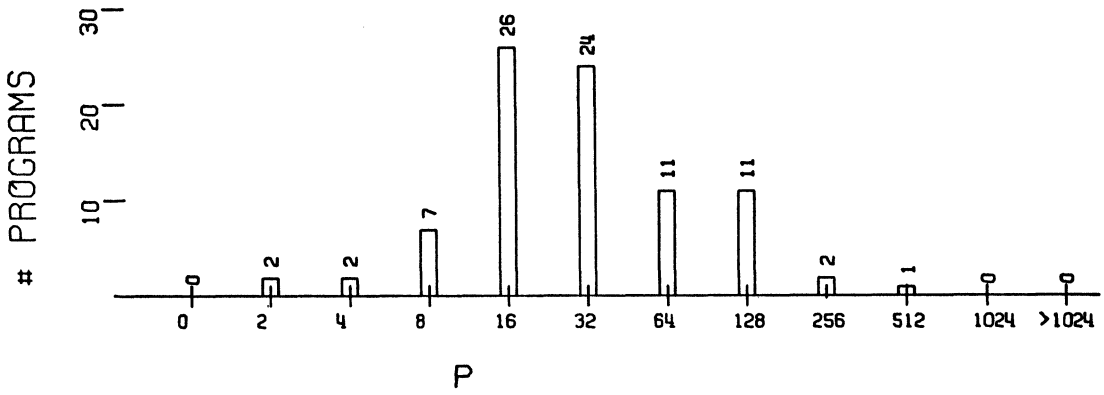


Figure 6. Number of Programs Versus $\bar{p}$

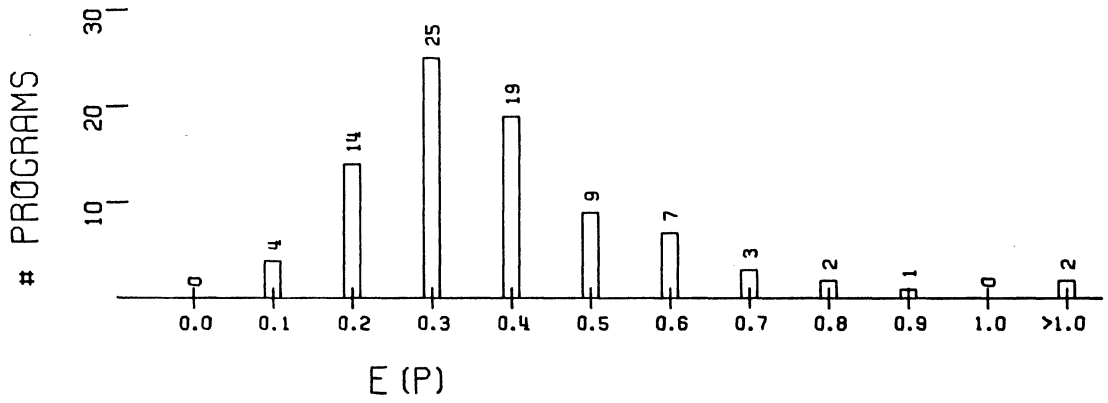


Figure 7. Number of Programs Versus $\bar{E}_p$

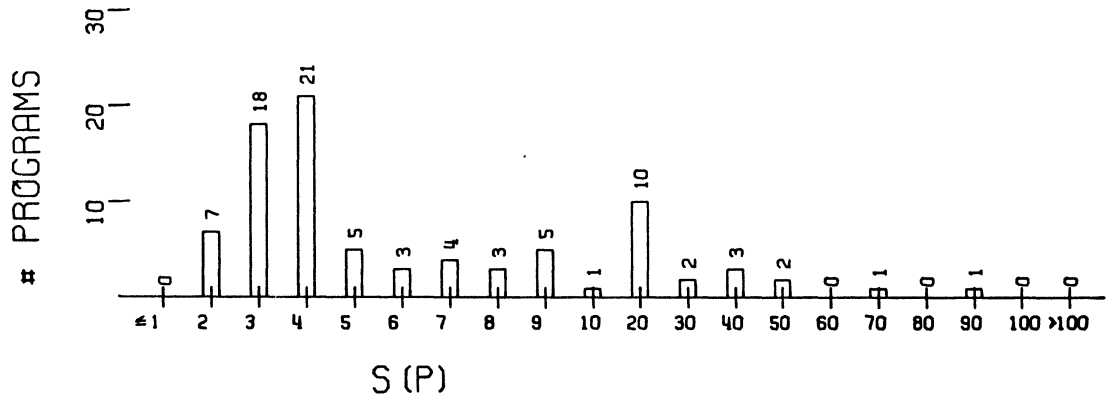


Figure 8. Number of Programs Versus $\bar{S}_p$

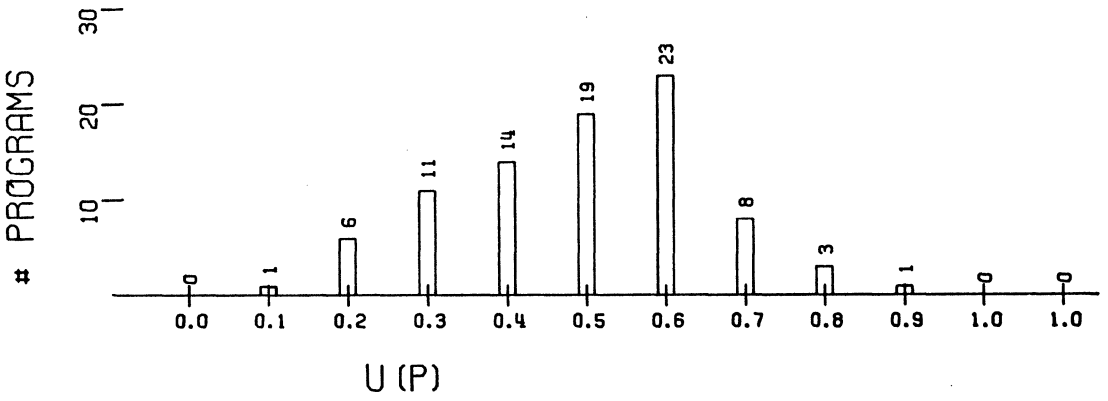


Figure 9. Number of Programs Versus $\bar{U}_p$

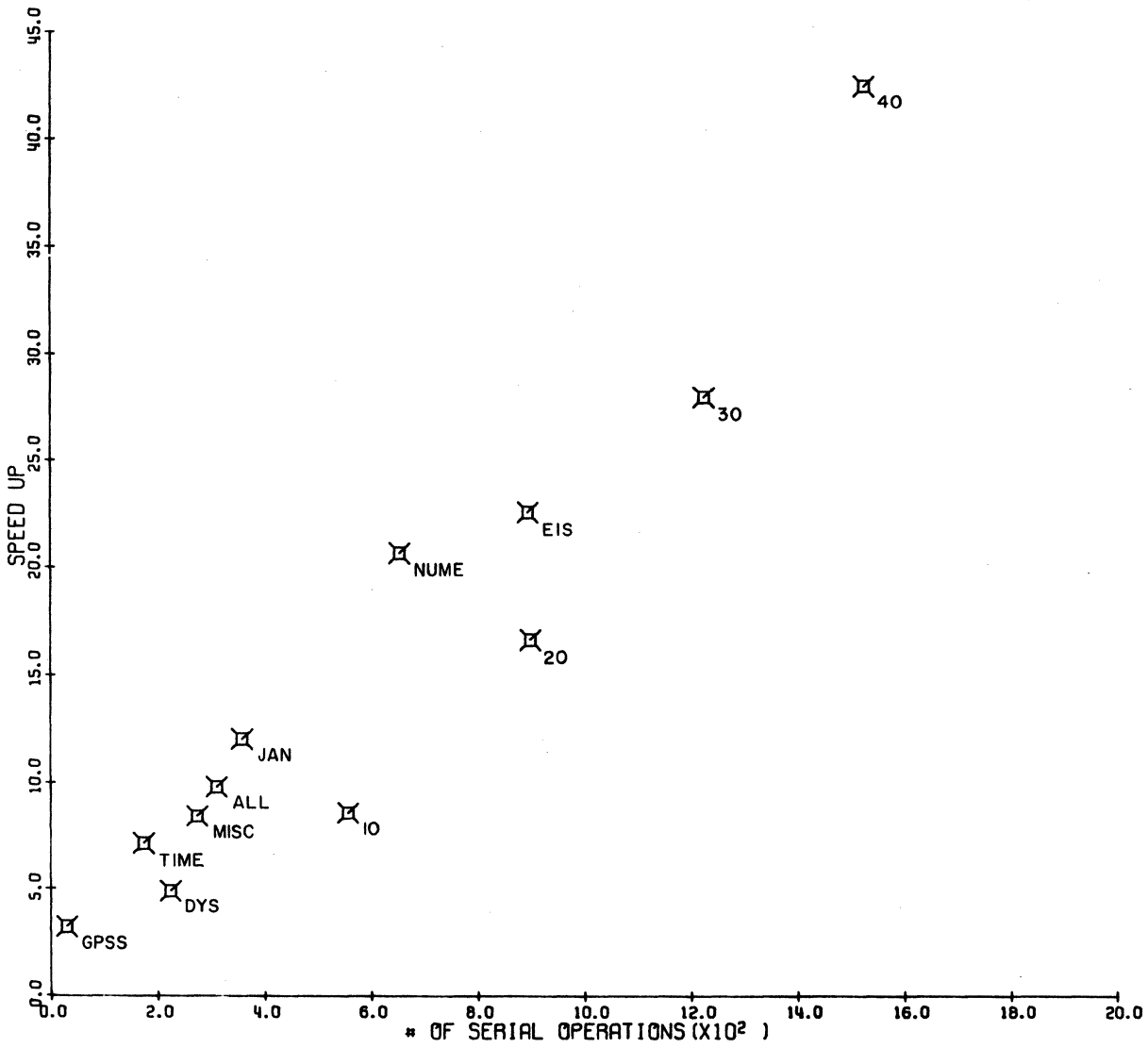


Figure 10. $\hat{S}_p$ Versus $\hat{T}_1$

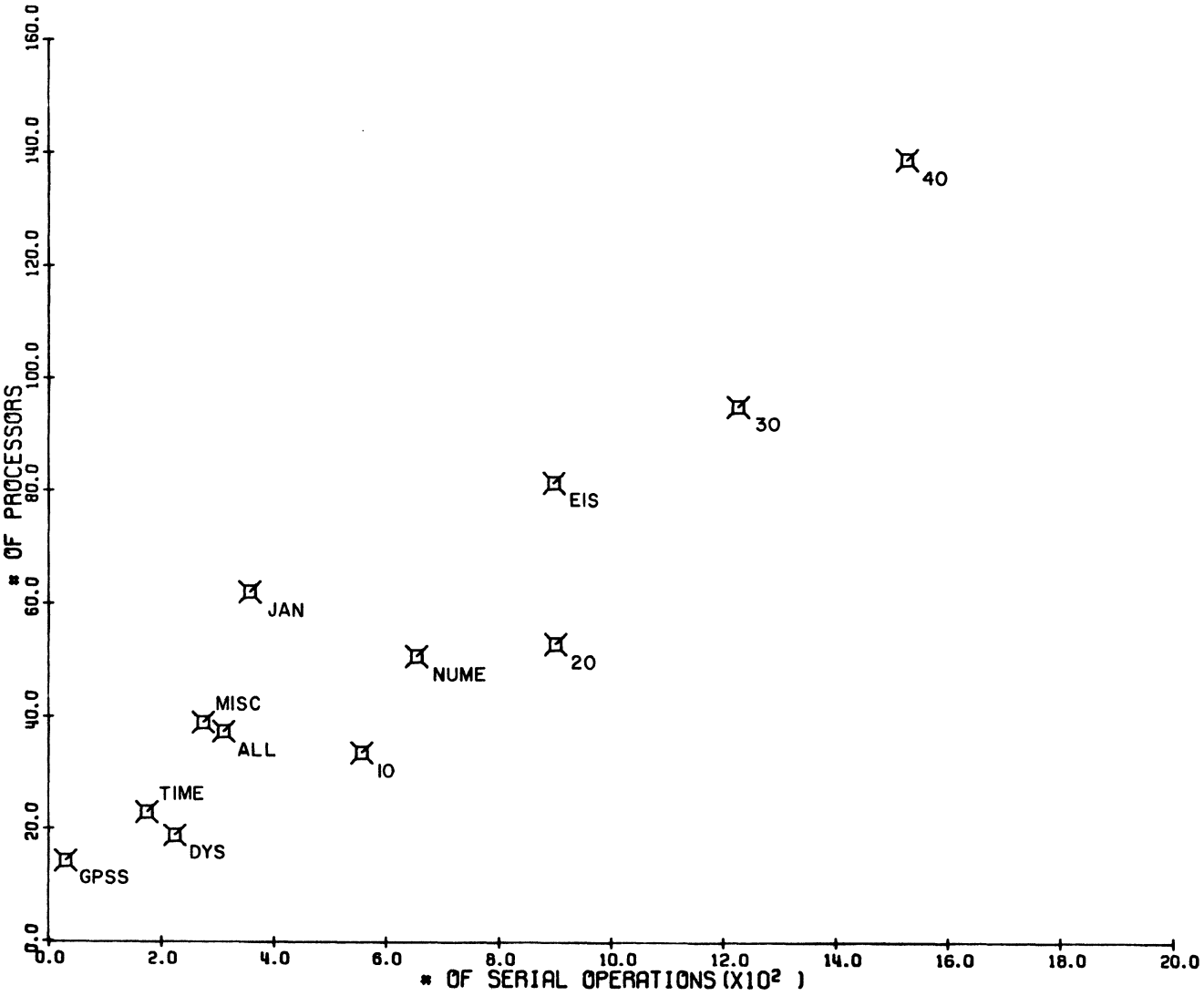


Figure 11. $\hat{p}$ Versus $\hat{T}_1$

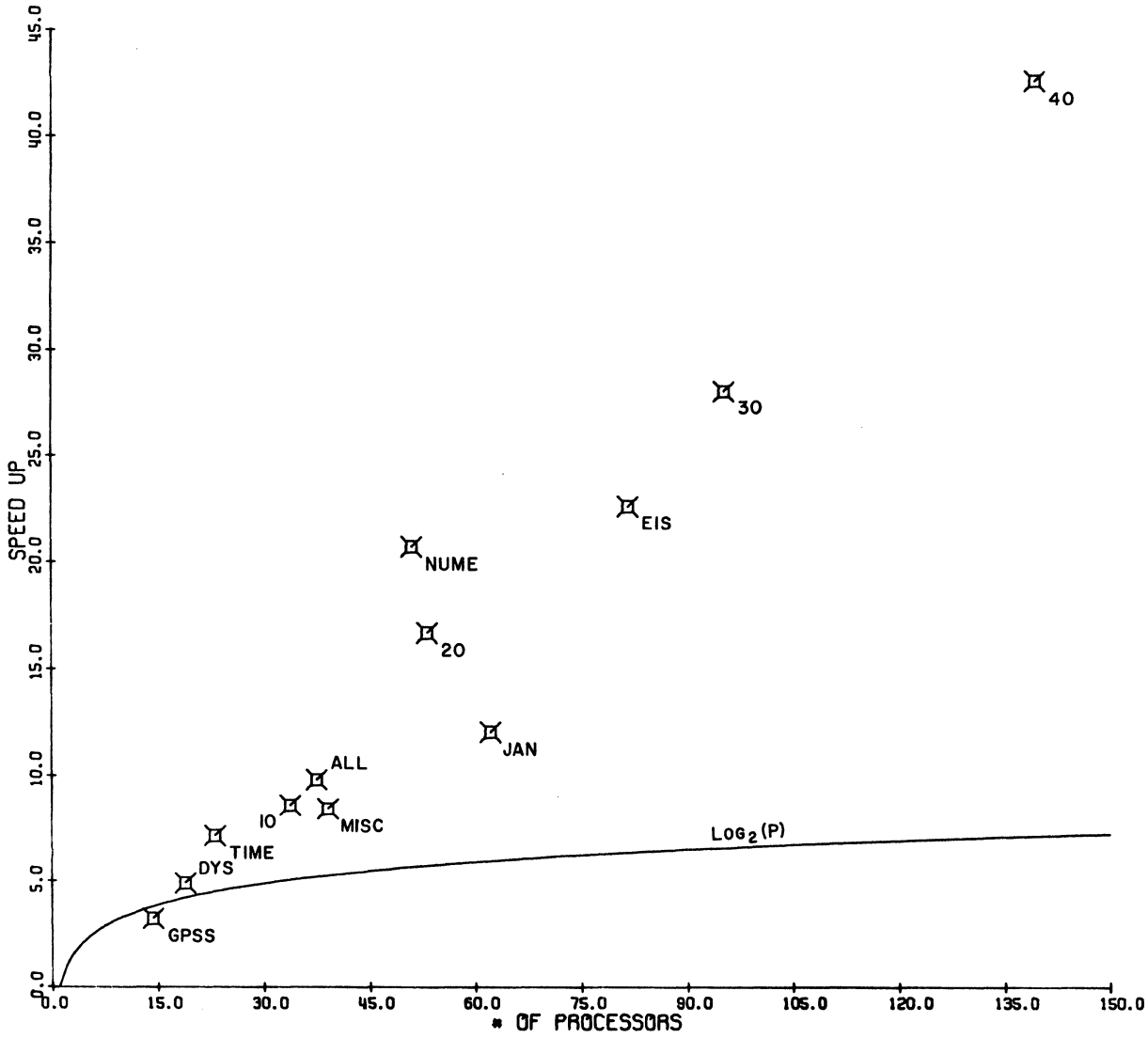


Figure 12. $\hat{S}_p$ Versus $\hat{p}$

A LANGUAGE FOR CONTROLLING PARALLEL PROCESSES

Bill R. Hays
Computer Science Department
Brigham Young University
Provo, Utah 84602

Summary

The design of computers with parallel capabilities, either as complete processors or multiple units, has raised the question of how one can take advantage of this increased computational ability. The paper presents a language designed for control of parallel processes.

There are many formal notations for parallelism (1,2), but the approach here utilizes formal language concepts and reduces the notational complexity. Parallel computer organization usually includes a control state and a similar idea is used here. In effect, one has a pushdown store automaton (3) controlling or scheduling other automatons. Parallel units will be called subacceptors or acceptors for discussion. The control is either local or global with the distinction that one global control state can permit a subacceptor to control another subacceptor (local control). The global control state can: (A) communicate with the subacceptors, (B) sequence subacceptors, (C) permit local control, and (D) select the proper subacceptor and determine if it is available. A stack is associated with each subacceptor for control and communication. Notationally, if A_i is a parallel subacceptor, then a symbol associated with its pushdown control state will be denoted by the subscript A_i. The production rules of the acceptors for parallel control are of the form: $(i,q_j,\phi_k)\rightarrow(q_m,b_r,\phi_{11}\phi_{12}\cdots\phi_{1n})$, with q_j the current state, q_m the next state, i the expected input symbol(s), ϕ_k the symbol(s) expected on top of the stack, b_r the output symbol(s), $\phi_{11}\cdots\phi_{1n}$ the output to the stack ($n\geq 0$). The rules are executed by simultaneously examining the current symbol in the input string and the top symbol of the stack. A production is executed only if both symbols are present. Successive rules associated with a given state are considered until a rule is executed or no production remains (this implies one must specifically provide the production rules for error conditions). After execution, the scan device is moved to the next symbol and a transfer is made to the specified state. All of the actions do not have to be performed and λ indicates the absence of such an action. In practice, one would simply omit it. The rules: $(\lambda,c,\phi_{A_i})\rightarrow(A_i,\lambda)$ and $(\lambda,c,\lambda)\rightarrow(A_i,\phi_{A_i})$ illustrate a transition based on reading the stack and no output with the second rule representing a transfer with output to the stack. Hence, the only required elements are the current state and next state.

Direct control of subacceptors will be accomplished by an associative list. Each element of the list corresponds to a subacceptor or state and contains control information. For example, if A_i and A_k are parallel subacceptors, then the list would contain ϕ_{A_i},ϕ_{A_k} as acceptor equivalents. The presence of ϕ_{A_i} indicates an acceptor is available and its absence indicates it is busy. ϕ_{A_i} will be a special symbol used only for selecting a subacceptor and is included in production rules as if the associative list were a stack. A read selects the symbol from a fixed place in the list and a write, by the same production, places the output symbol in this sublist. Hence, production rules accessing the associative list can write only at the entry at which it reads. The distinct types of production rules are:

(A) Standard read-only, erase-only, read-write, read-erase and read-erase-write rules. (B) Control productions of the form: 1- $(a,A,\phi_{X_i})\rightarrow(X_i,\phi_i)$ which reads the associative control list and activates the parallel subacceptor X_i, assigning it the stack ϕ_i. 2- $(\lambda,A,\phi_{X_i})\rightarrow(X_i,\phi_i\phi_A)$ which releases 'A' for further activation by placing ϕ_A back on the associative control list. 3- $(\lambda,A,(\phi_{X_i}\phi_i)_A\phi_{X_i})\rightarrow(X_i,\phi_i\phi_A)$ which reads a request for X_i to process the stack ϕ_i and performs the request by activating X_i and passing the required information. (C) Local control productions of the form: 1- $(w,X_i,a_iw_j\phi_y)\rightarrow(Y,w_j(\phi_{X_i},\phi_i\phi_A)$ which activates 'Y' to process the stack ϕ_i and requests a return to state X_i. 2- $(w,X_i,a_iw_i\phi_Y)\rightarrow(Y,w_j\phi_i\phi_{X_i})$ if no waiting is necessary.

(D) Subacceptor productions of the form: 1- $(\lambda,X_i,a_i\phi_c)\rightarrow(c,(\phi_i\phi_Y)_c\phi_{X_i})$ which request the control state 'C' to activate X_j to process the pushdown store ϕ_i. 2- $(\lambda,X_i,a_i\phi_c)\rightarrow(C,(\phi_{X_i}\phi_i\phi_Y)_c\phi_{X_i})$ which also requests a return to state X_i.

The language could be used to write the procedures, but it would be expected that only the control procedures would be written in the language.

References

(1) J. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computation" CACM, 9 (March, 1966), pp. 143-155.

(2) E. W. Dijkstra, "Cooperating Sequential Processes", Programming Languages, F. Genuys, Ed. Academic Press, 1968.

(3) Hopcroft and Ulman, Formal Languages and Their Relation to Automata, Addison, Wesley, 1969.

THE TRANSFORMATION OF FLOW DIAGRAMS INTO MAXIMALLY PARALLEL FORM

G. Urschler
System Development Division
IBM Corporation
Endicott, N.Y. 13760

Abstract — The algorithmic transformation of flow diagrams into a goto- and variable-free parallel program representation is described. It is shown, how the control mechanism for these parallel programs works and that it exhibits dynamically maximum parallelism in a certain, well-defined sense. The method presented is new and gives the optimum that can be achieved in intra-task parallelism.

Introduction

General Introduction

In an attempt to categorize the types of parallelism, the following definitions are presented:

1. Inter-task parallelism. Dependencies between concurrently executing work units are allowed. Synchronization and deadlock prevention techniques are required as well as explicit language features for the specification of parallelism.

2. Intra-task parallelism. No dependencies between concurrently executing work units are allowed. Required are methods for the automatic detection of parallelism.

3. Parallelism on the hardware level.

This paper is concerned with intra-task parallelism, and the area of particular interest is "maximum" parallelism. Although it has been proven that the parallelization problem is an undecidable one [1], the results presented in this paper were possible because of a different understanding of the term "parallelization."

Adding of redundancy, for instance, commonly is not regarded as parallelization. However in this paper also the detection and exploitation of an already existing redundancy is not regarded as parallelization, but as optimization. Thus the above mentioned proof is regarded as a proof for the undecidability of the optimization problem and thus not conflicting with the contents of this paper.

Scope of the presented parallelization method

Core language. The method has been developed for an input language consisting of read and write statements, assignment statements, and branch and decision statements (flow diagrams). Expressions are restricted to either simple data variables (a, b, ...) or to simple expressions (a+b, f(a, b, c), ...).

Extended language. The method obviously also applies to each language being translatable into the core language. Thus it works for a language additionally containing composite expressions, fixed data structures, and constant references (A[1] for instance, as opposed to A[i]).

Extension possibilities. Not described in this paper, but known, are extensions of the method to a core language containing subroutines and to the parallelization of more than one task.

Not covered. Not known at the present time are extensions to languages involving varying data structures, computed references (pointers, subscripted variables with subscripts to be evaluated dynamically), and exception handling.

Maximum parallelism

Based on the above core language, a more precise definition of the notion of "maximum parallelism" can be given. A statement obviously can be executed as soon as:

1. all decisions on which this statement execution is dependent upon have been resolved (this kind of dependency is called a control dependency),

2. all input values required for the statement's execution have been generated (the corresponding dependency is called an input dependency), and

3. it is known that these input values have been generated (the corresponding dependency is called a data dependency); the need for the latter case, being more subtle than the previous ones, is illustrated in Figure 1.

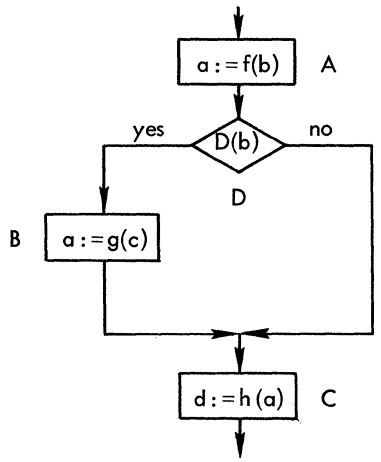


Fig. 1 - Flow Diagram Showing a Data Dependency

If a resolution of D to the no branch is assumed, then the input values for C have been produced before execution of D by the preceding A, but this fact becomes apparent only after resolution of D and thus C has to wait for D too.

Sequencing constraints caused by control-, input- and data dependencies only, are called necessary ones. Maximum parallelism now means that the only logical sequencing constraints to be followed at execution time are necessary sequencing constraints.

Benefits of the method

Most of the conventional parallelization methods [2], [3] parallelize on a program basis, trying to divide a program into independent program blocks. Thus the parallelism which can be detected inherently is limited by the size of the given program. The presented method, however, parallelizes on a computation (program execution) basis, thus giving the more (potentially infinite) parallelism, the lengthier the computation is. As byproducts, new and highly efficient program analysis methods as well as a quite unusual parallel program concept are developed. The latter gives both an insight into the nature of parallelism on the intra-task level and a certain understanding of what a machine exploiting this kind of parallelism might look like.

Paper overview

The method is illustrated by means of the program in Figure 2 (the function $y = \sum_{x=1}^{N} \sqrt{x}$ is computed with an error precision f for the square root calculations). ▽ denotes the program beginning and △ the program end. The capital letters are used later for the symbolic reference of statements and program blocks, respectively.

In the following, a thorough program analysis is made of this program, and based upon this analysis the program is translated at first into a "single assignment" form (in which each variable is written to, at most, once) and finally into a variable-free form. As auxiliary tool (regular) production systems from the theory of syntax are used.

Program Analysis

Control flow analysis

1. Determination of the program logic. A program like the one shown normally is — because of the unlimited use of branching — a bowl of spaghetti. Thus the first step in the analysis is the determination of the logic (the structure) of the given program (for a more exhaustive description of this step see Reference [4]).

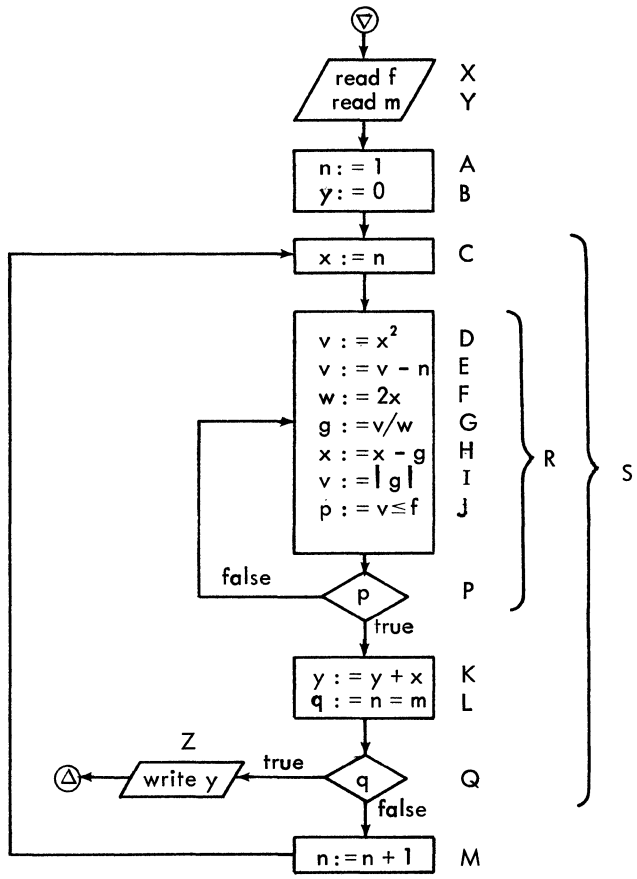


Fig. 2 - The Source Program to be Parallelized

As an auxiliary tool the notion of "immediate post dominator" is used [5], [6]. For a statement branching unconditionally, the immediate post dominator is identical to the successor of this statement. For decisions, the immediate post dominator is that (uniquely determined) statement at which all branches evolving from the decision join for the first time. Thus in the given example, decision P has the immediate post dominator K and decision Q the immediate post dominator Z. If a chain of succeeding immediate post dominators is referred to as a control flow, the control flow of the given program can be described by means of the following "production":

▽ ::= X Y A B C D E F G H I J P K L Q Z △

The undefined elements in this production are "modules" P and Q, which again are described by productions as follows (ϵ denotes the empty string; the first alternative describes the "true" branch and the second one the "false" branch):

$$P ::= \{_{p}\ \epsilon \mid D\ E\ F\ G\ H\ I\ J\ P\}$$

$$Q ::= \{_{q}\ \epsilon \mid M\ C\ D\ E\ F\ G\ H\ I\ J\ P\ K\ L\ Q\}$$

Productions for decisions are derived by constructing the control flow for each successor of the decision and following it as long as the "scope" of the decision is not left (which means that the immediate post dominator of the decision itself is not yet reached). Particularly, it thus can happen — as in the above example — that a decision alternative becomes empty.

2. Program reduction. The translation of a flow chart into a goto-free form has, in essence, been achieved by the copying of program text (and not as in the Boehm/Jacopini method [7] by the introduction of "control switches"). Thus the new program, in general, becomes much larger than the old one. This inconvenience is removed in the following by the introduction of abbreviations for lengthy strings occurring more than once. This reduces the program to the dimensions of the source program. For the given example this results in:

$$\nabla ::= X\ Y\ A\ B\ S\ Z\ \Delta$$

$$S ::= C\ R\ K\ L\ Q$$

$$R ::= D\ E\ F\ G\ H\ I\ J\ P$$

$$P ::= \{{}_{p}\epsilon \mid R\}$$

$$Q ::= \{{}_{q}\epsilon \mid M\ S\}$$

Data flow analysis

1. Derivation of local production systems. From the above "global" production system, the following set of "local" production systems is derived, each of which describes the program from the point of view of a single data variable only. "M^v" has the meaning "Module M as seen from variable v", "S^R" means "read operation in statement S" and analogously "S^W" means "write operation in statement S". Productions for modules not containing a certain variable are omitted. Altogether this gives the results shown in Table I.

$$\nabla \begin{cases} \nabla^x ::= S^x \\ \nabla^v ::= S^v \\ \nabla^n ::= A^W\ S^n \\ \nabla^w ::= S^w \\ \nabla^g ::= S^g \\ \nabla^f ::= X^W\ S^f \\ \nabla^p ::= S^p \\ \nabla^y ::= B^W\ S^y\ Z^R \\ \nabla^q ::= S^q \\ \nabla^m ::= Y^W\ S^m \end{cases} \qquad S \begin{cases} S^x ::= C^W\ R^x\ {}^2K^R\ Q^x \\ S^v ::= R^v\ Q^v \\ S^n ::= C^R\ R^n\ {}^1L^R\ Q^n \\ S^w ::= R^w\ Q^w \\ S^g ::= R^g\ Q^g \\ S^f ::= R^f\ Q^f \\ S^p ::= R^p\ Q^p \\ S^y ::= {}^1K^R\ K^W\ Q^y \\ S^q ::= L^W\ Q^R\ Q^q \\ S^m ::= {}^2L^R\ Q^m \end{cases} \qquad P \begin{cases} P^x ::= \{\epsilon \mid R^x\} \\ P^v ::= \{\epsilon \mid R^v\} \\ P^n ::= \{\epsilon \mid R^n\} \\ P^w ::= \{\epsilon \mid R^w\} \\ P^g ::= \{\epsilon \mid R^g\} \\ P^f ::= \{\epsilon \mid R^f\} \\ P^p ::= \{\epsilon \mid R^p\} \end{cases}$$

$$R \begin{cases} R^x ::= D^R\ F^R\ {}^1H^R\ H^W\ P^x \\ R^v ::= D^W\ {}^1E^R\ E^W\ {}^1G^R\ I^W\ {}^1J^R\ P^v \\ R^n ::= {}^2E^R\ P^n \\ R^w ::= F^W\ {}^2G^R\ P^w \\ R^g ::= G^W\ {}^2H^R\ I^R\ P^g \\ R^f ::= {}^2J^R P^f \\ R^p ::= J^W P^R P^p \end{cases} \qquad Q \begin{cases} Q^x ::= \{\ \epsilon \mid S^x\} \\ Q^v ::= \{\ \epsilon \mid S^v\} \\ Q^n ::= \{\ \epsilon \mid M^R\ M^W S^n\} \\ Q^w ::= \{\ \epsilon \mid S^w\} \\ Q^g ::= \{\ \epsilon \mid S^g\} \\ Q^f ::= \{\ \epsilon \mid S^f\} \\ Q^p ::= \{\ \epsilon \mid S^p\} \\ Q^y ::= \{\ \epsilon \mid S^y\} \\ Q^q ::= \{\ \epsilon \mid S^q\} \\ Q^m ::= \{\ \epsilon \mid S^m\} \end{cases}$$

Table I - Local Production Systems Describing the Flow of Data for the Given Source Program

2. Determination of module interfaces. A local module of the kind M^v defines (in syntactical terms) a certain language, the sentences of which are composed of read and write operations only. A module is called read-like if the corresponding language contains read operations only. A module is said to require input (denoted by $>M^v$), if at least one of the sentences defined by it starts with a read operation. Analogously it is said that output is required from a module (denoted by $M^v<$), if it is not a read-like module and if there is at least one occurrence of M^v which is followed either by a read operation or by an input requiring module.

3. Incorporation of data dependencies. Local modules from which an output is required almost behave like a write operation in the sense that a variable becomes redefined by them. This is, however, not always true. Whenever there is an alternative which has a read-like behavior (only read operations are involved), then the external appearance of this module becomes inconsistent. Sometimes it redefines the corresponding variable and sometimes it does not. This is exactly the situation which earlier was referred to as a data dependency. It is removed by introducing copy statements (like x:=x) in those alternatives of modules for which output is required (in which, otherwise, no redefinition would occur). Thus, in the given example, two copy statements (symbolically denoted by N and O) have to be introduced in the modules P^x and Q^y, respectively.

4. Introducing logical variables. Whenever one and the same physical variable becomes redefined, then from a logical point of view this is a new variable. This can be indicated by segmenting each alternative in such a way that after each write operation and after each local module from which output is required, a new segment (being the scope of a new logical variable) begins (denoted by $A^w \|_n S^n$ for instance). In addition, each alternative of an input requiring module also starts with a segment. The variable names for the segments can be chosen freely with the restrictions that (a) all variable names within the same alternative have to be different, (b) different alternatives of the same module have to begin and end with identically named variables, and (c) the last variable name in each alternative should be that of the corresponding physical variable. Altogether for the given example, the following set of extended local production systems, incorporating all data flow analysis information, is obtained (Table II).

$$\nabla \left\{ \begin{array}{l} \nabla^x ::= S^x \\ \nabla^v ::= S^v \\ \nabla^n ::= A^w \|_n S^n \\ \nabla^w ::= S^w \\ \nabla^g ::= S^g \\ \nabla^f ::= X^w \|_f S^f \\ \nabla^p ::= S^p \\ \nabla^y ::= B^w \|_{ya} S^y \|_y Z^R \\ \nabla^q ::= S^q \\ \nabla^m ::= Y^w \|_m S^m \end{array} \right. \qquad S \left\{ \begin{array}{l} S^x ::= C^w \|_{xa} R^x \|_x {}^2K^R Q^x \\ S^v ::= R^v Q^v \\ >S^n ::= \|_n C^R R^n {}^1L^R Q^n \\ S^w ::= R^w Q^w \\ S^g ::= R^g Q^g \\ >S^f ::= \|_f R^f Q^f \\ S^p ::= R^p Q^p \\ >S^y< ::= \|_{ya} {}^1K^R K^w \|_{yb} Q^y \|_y \\ S^q ::= L^w \|_q Q^R Q^q \\ >S^m ::= \|_m {}^2L^R Q^m \end{array} \right. \qquad P \left\{ \begin{array}{l} >P^x< ::= \{ \|_{xa} N^R N^w \|_x \mid \|_{xa} R^x \|_x \} \\ P^v ::= \{ \epsilon \mid R^v \} \\ >P^n ::= \{ \epsilon \mid \|_n R^n \} \\ P^w ::= \{ \epsilon \mid R^w \} \\ P^g ::= \{ \epsilon \mid R^g \} \\ >P^f ::= \{ \epsilon \mid \|_f R^f \} \\ P^p ::= \{ \epsilon \mid R^p \} \end{array} \right.$$

$$R \left\{ \begin{array}{l} >R^x< ::= \|_{xa} D^R F^R {}^1H^R H^w \|_{xb} P^x \|_x \\ R^v ::= D^w \|_{va} {}^1E^R E^w \|_{vb} {}^1G^R I^w \|_v {}^1J^R P^v \\ >R^n ::= \|_n {}^2E^R P^n \\ R^w ::= F^w \|_w {}^2G^R P^w \\ R^g ::= G^w \|_g {}^2H^R I^R P^g \\ >R^f ::= \|_f {}^2J^R P^f \\ R^p ::= J^w \|_p P^R P^p \end{array} \right. \qquad Q \left\{ \begin{array}{l} Q^x ::= \{ \epsilon \mid S^x \} \\ Q^v ::= \{ \epsilon \mid S^v \} \\ >Q^n ::= \{ \epsilon \mid \|_{na} M^R M^w \|_n S^n \} \\ Q^w ::= \{ \epsilon \mid S^w \} \\ Q^g ::= \{ \epsilon \mid S^g \} \\ >Q^f ::= \{ \epsilon \mid \|_f S^f \} \\ >Q^y< ::= \{ \|_{ya} O^R O^w \|_y \mid \|_{ya} S^y \|_y \} \\ Q^q ::= \{ \epsilon \mid S^q \} \\ >Q^m ::= \{ \epsilon \mid \|_m S^m \} \end{array} \right.$$

Table II - Extended Local Production Systems Showing all Data Flow Analysis Information

Program Transformation

Transformation into a "single assignment" form

The information given by the data flow analysis makes it possible to translate the original global production system into a program form in which each occurring variable is defined at most once (see References [8], [9]). To do this, each module is associated with three kinds of parameters:

1. A decision variable. Based on the value of this variable, a corresponding alternative is chosen. If the module is an unconditional one, there is no decision variable.

2. A list of input variables. This is a list of all those variables that the data flow analysis has shown are required as input to this module.

3. A list of output variables. This is the list of all those variables that the data flow analysis has shown are required as input to this module.

The syntactic notation chosen is illustrated by the following example:

> Q(q)[n, f, yb, m; y] (q is the decision variable, input and output variables are separated by a semicolon, and y is an output variable).

The right side of the definition of an unconditional module is an alternative, being a list of statements separated by semicolons and enclosed in braces. Conditional modules are described by conditional expressions, for instance, in the form:

p = 1 → alternative 1
p = 2 → alternative 2

(the truthvalue true is represented by 1, and false by 2). The variables occurring in each statement are taken from the corresponding segment of the extended local production systems. Altogether for the given example the single-assignment form shown in Table III is obtained (note that the previously introduced symbolic statement names are indicated above the lines).

In such a program, the "basic statements"(assignment and I/O statements) and "expansion statements" (module call statements) can be distinguished. The program can be executed in parallel as follows: Starting with an instance (i.e. a copy) of the begin statement ∇, execution of this statement expands into a set of statement instances of the corresponding alternative. In this set, an instance of a basic instruction becomes executable as soon as all its input variables have a defined value (because of the single-assignment property there is no misinterpretation of the definition point possible). An instance of an expansion statement is executable, as soon as its decision variable — if any — has been defined. Thus instances of unconditional expansion statements always are executable. Execution of an expansion statement instance evolves in an expansion incorporating the corresponding alternative, whereby passing by name of parameters is assumed and "internal" variables not occurring in any parameter list are assumed to be newly created.

This execution mechanism gives maximum parallelism because the only sequencing constraints are given by the following facts: 1) a statement instance has to wait until it has been generated (which according to the program structure means that it has to wait until all control dependencies have been resolved), and, 2) it has to wait for its inputs (coming either direct from the "input producer" in which case an input dependency is resolved or from a copy statement, in which case a data dependency

```
                X       Y       A       B                          Z
∇ ::= {read f; read m; n:=1; ya:=0; S[n,f,ya,m;y]; write y}

                        C                          K        L
S[n,f,ya,m; y]::={xa:=n; R[xa,n,f;x]; yb=ya+x; q:=n=m;

                  Q(q)[n,f,yb,m;y]}

                   D        E        F       G        H
R[xa,n,f;x]::={va:=xa²; vb:=va-n; w:=2xa; g:=vb/w; xb:=xa-g;

                 I       J
              v:=|g|; p:=v≤f; P(p)[xb,n,f;x]}

P(p)[xa,n,f;x]::=
                          N
          p=1 ⟶ {x:=xa}
          p=2 ⟶ {R[xa,n,f;x]}

Q(q)[na,f,ya,m;y]::=
                          0
          q=1 ⟶ { y:=ya}
                          M
          q=2 ⟶ { n:=na+1; S[n,f,ya,m;y]}
```

Table III - The Original Flow Diagram in "Single-Assignment" Form

is resolved). "Overhead" statements such as all unconditional expansion statements different from ▽, are no obstacle for maximum parallelism, because their instances are unconditionally executable and can be regarded as part of the expansion of the preceding conditional expansion statement instance. For an equivalence proof of the single-assignment program and the original flowchart, see Reference [10].

The drawback of the single-assignment representation is that the control mechanism is not totally explicit. Although logically it is clear when a variable gets a defined value, the signalling of the arrival of this value to the involved statement instances is not shown in the control mechanism. This drawback is removed in the next and last transformation step.

Transformation into a "variable-free" form

1. Introduction of "distribution statements". In the single-assignment program form each variable is defined at most once. There is no limitation on the number of readings from one variable, however. By introducing distribution statements (being multiple assignments distributing a variable value to all places in an alternative where this value is needed) a program form can easily be reached, where each variable also is read at most once. The idea of this transformation is to store a generated value not indirectly to a data base (from where it can be retrieved under its name), but directly to all places where it is needed (which makes a "local" determination of executability possible).

2. Introduction of "buffer statements". The problem with the exploitation of the previous transformation is that when a value has to be inserted directly in all reference places, then these places have to exist, i.e. they have to be allocated. This means that a synchronization between "value definition" and "value place allocation" is necessary, which can subvert maximum parallelism.

The solution to this problem is the introduction of "buffer statements" (being copy statements), which are inserted between a value generating basic- or expansion-statement and the corresponding expansion statement requiring this value as input. No buffer statements are used if the basic statement is a simple one (involves no expression evaluation).

3. Replacing variables by addresses. In the following, each module alternative is assumed to be arranged linearly, so that each symbol occurring in it has an "address" (relative to this alternative). Each alternative later is assumed to be preceded by an "address vector", being the list of addresses of all module parameters with respect to this alternative. (Note that addresses are denoted by an arrow over a variable name (e.g., $\vec{a}$); they point to the place where the plain variable name occurs (e.g., a) which is not to be interpreted as a variable, but as a "placeholder"). When a parameter does not occur in an alternative, the corresponding address is denoted by the "null" address, →. Input parameters occurring in a basic statement are called direct input parameters; all other input parameters (occurring again in expansion statements) are indirect ones (denoted by an underlining of the corresponding address in the address vector).

Within each alternative, each variable not being a global parameter occurs exactly twice. The general rule for the replacement of variables by addresses is that the place where the "allocation" is being done (or in case the allocation is done by the surrounding module, then the place where the definition is done) becomes the address of the corresponding mate and the mate is interpreted as a placeholder. Thus if both variable occurrences are in basic statements, then the definition place becomes the address of the reference place and if one variable occurrence is within an expansion statement (as "local" parameter) and the other in a basic statement, then the parameter becomes the address of the other variable occurence (independent of whether the latter is used for reference or for definition). Parameter lists as well as the case distincting conditions become redundant now. All that is needed is a description of alternatives, which for the given example is a self explanatory form is given by Table IV. (Note that the symbolic statement names are indicated above the lines.)

Program Execution

A program obtained can be regarded as a parallel machine program being executed as follows:

1. A copy of the "body" of the begin module ▽ is fetched into a "control storage," thereby replacing relative addresses by absolute ones.

2. An instance of a basic statement becomes executable, if all its definition places are (absolute) addresses and all its reference places are values. It is executed by evaluating the "right side" expression, storing the obtained value to the indicated addresses (in case of a null address no storing takes place), and deleting the executed statement instance in the control storage afterwards.

3. An instance of an expansion statement becomes executable, if its reference place (the previous decision variable) — if any — is a value and if all of its parameter places are (absolute) addresses. It is executed by fetching a copy of the body of the corresponding alternative into the control storage (if there is enough space), thereby replacing absolute addresses by relative ones and performing the following "parameter passing": The address of a direct input parameter is written to the address given by the corresponding "actual" (contained in the invoking statement instance) parameter. Addresses found in actual output or actual indirect input parameter places are, however, written to the address of the corresponding newly allocated "formal" parameter (thus in this case the passing of parameters has the "conventional" direction, whereas in the former case the passing

```
        X        Y      A      B          S ▽          Z
▽::={read f; read m; n:=1; ya:=0; S[n→,f→,ya→,m→;y→]; write y}

                                    SN       SF        SM       C        RB       RC
S::=[n→,f→,ya→,m→;y→] {na→,nb→,nc→,nd→:=n; fa→,fb→:=f; ma→,mb→:=m; xa:=na; naa:=nb; faa:=fa;

            RS          K        L          QA       QB       QC       QD
    R[xa→,naa→,faa→;x→]; yb→:=ya+x; q→:=nc=ma; nda:=nd; fba:=fb; yba:=yb; mba:=mb;

                    Q
    Q(q)[nda→,fba→,yba→,mba→;y]}

                                    RX       RN       RF       D        E        F
R::=[xa→,n→,f→;x→]{xaa→,xab→,xac→:=xa; na→,nb→:=n fa→,fb→:=f; va→:=xaa→²; vb→:=va-na; w→:=2xab;

    G          RG       H          I          J        PA       PB       PC
    g→:=vb/w; ga→,gb→:=g; xb→:=xac-ga; v→:=|gb|; p:=v≤fa;xba:=xb; nba:=nb; fba:=fb;

                    P
    P(p)[xba→,nba→,fba→;x]}

                    N
P1::=[xa→,→,→;x→]{x:=xa}

                    RP
P2::=[xa→,n→,f→;x→]{R[xa→,n→,f;x]}

                    0
Q1::=[→,→,ya→,→;y→]{y:=ya}

                    M          SA       →  SQ
Q2::= [na→,f→,ya→,m→;y→]{n→:=na+1; na:=n; S[na,f,ya,m;y]}
```

Table IV — The Original Flow Diagram in "Variable-free" Representation

direction is reversed).

This execution mechanism is illustrated in Table V by a possible execution begin for the given program and the assumed input values f = 0.1 and m = 2.

All execution possibilities for the previous program inputs are described symbolically by the "precedence-graph," shown in Figure 3.

Analysis of the graph shows that the computation of different square roots can be done in parallel (thus the "outer" loop in the original flow diagram is a parallel one), whereas the iterations required to compute the same square root have to be done in serial.

Summary

The paper has shown how flow diagrams of a certain restricted standard form automatically can be transformed (at compile time) into a set of goto-free and variable-free set of modules, constituting a highly parallel program structure. The intelligence incorporated into the resulting programs not only allows the exploitation of maximum parallelism, but also provides for dynamic storage allocation, dynamic relocation and direct data processing (as opposed to indirect data processing implied by the use of variables).

The techniques used can, if properly understood, be very fruitful for the further development of many different areas including (optimizing) compilers, operating systems (paging techniques), and new (highly parallel) machine concepts.

References

[1] A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. of Electr. Comp. (Oct. 1966), pp. 757-763.

[2] H. W. Bingham et al, Automatic Detection of Parallelism in Computer Programs, Burroughs Corp., Paoli, Pa., Technical Report (Nov. 1967)

[3] M. R. Shapiro et al, The Representation of Algorithms, Applied Data Research Inc., New York, N.Y., Technical Report, (Sept., 1969).

[4] G. Urschler, "Automated Functional Programming," paper available from author.

[5] R. T. Prosser, "Application of Boolean Matrices to the Analysis of Flow Diagrams," 1959 Proc. of the EJCC, pp. 133-138.

[6] E. S. Lowry and C. W. Medlock, "Object Code Optimization," CACM, Vol. 12, (Jan., 1969).

[7] C. Boehm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only two Formation Rules," CACM, Vol. 9, (1966) pp. 366-371.

[8] L. G. Tesler and H. J. Enea, "A Language Design for Concurrent Processes," Proc. of the SJCC 1968, pp. 403-408.

[9] D. D. Chamberlin, "The 'Single Assignment' Approach to Parallel Processing," Proc. of the FJCC 1971, pp. 263-269.

[10] G. Urschler, The Inherent Parallelism of Flow Diagrams, IBM Lab, Vienna, Technical Report 25.129 (July, 1972) p. 40.

Simultaneously performed statement instances	Result
∇_0	X_0 Y_0 A_0 B_0 $S\nabla_0$ Z_0 read f_0; read m_0; $n_0 := 1$; $ya_0 := 0$; $S[\vec{n}_0, \vec{f}_0, \vec{ya}_0, \vec{m}_0; \vec{y}_0]$; write y_0
S_0	X_0 Y_0 A_0 B_0 SN_0 SF_0 read $\vec{f}_0$; read $\vec{m}_0$; $\vec{n}_0 := 1$; $\vec{ya}_0 := 0$; $\vec{na}_1, \vec{nb}_1, \vec{nc}_1, \vec{nd}_1 := n_0$; $\vec{fa}_1, \vec{fb}_1 := f_0$; SM_0 C_0 RB_0 RC_0 RS_0 $\vec{ma}_1, \vec{mb}_1 := m_0$; $xa_1 := na_1$; $naa_1 := nb_1$; $faa_1 := fa_1$; $R[\vec{xa}_1, \vec{naa}_1, \vec{faa}_1; \vec{x}_1]$; K_0 L_0 QA_0 QB_0 QC_0 QD_0 $\vec{yb}_1 := ya_0 + x_1$; $\vec{q}_1 := nc = ma$; $nda_1 := nd_1$; $fba_1 := fb_1$; $yba_1 := yb_1$; $mba_1 := mb_1$; Q_0 Z_0 $Q(q_1)[\vec{nda}_1, \vec{fba}_1, \vec{yba}_1, \vec{mba}_1; \vec{y}_0]$; write y_0
X_0, Y_0, A_0, B_0, RS_0	SN_0 SF_0 SM_0 C_0 RB_0 $\vec{na}_1, \vec{nb}_1, \vec{nc}_1, \vec{nd}_1 := 1$; $\vec{fa}_1, \vec{fb}_1 := 0.1$; $\vec{ma}_1, \vec{mb}_1 := 2$; $xa_1 := na_1$; $naa_1 := nb_1$; RC_0 RX_0 RN_0 RF_0 D_0 $faa_1 := fa_1$; $\vec{xaa}_2$, $\vec{xab}_2$, $\vec{xac}_2 := xa_1$; $\vec{na}_2, \vec{nb}_2 := naa_1$; $\vec{fa}_2$ $\vec{fb}_2 := faa_1$; $va_2 := xaa_2^2$; E_0 F_0 G_0 RG_0 H_0 $\vec{vb}_2 := va_2 - na_2$; $\vec{w}_2 := 2xab_2$; $\vec{g}_2 := vb_2 / w_2$; $\vec{ga}_2, \vec{gb}_2 := g_2$; $xb_2 := xac_2 - ga_2$; I_0 J_0 PA_0 PB_0 PC_0 $\vec{v}_2 := \lvert gb_2 \rvert$; $\vec{p}_2 := v_2 \leq fa_2$; $xba_2 := xb_2$; $nba_2 := nb_2$; $fbc_2 := fb_2$; P_0 K_0 L_0 QA_0 $P(p_2)[\vec{xba}_2, \vec{nba}_2, \vec{fba}_2; \vec{x}_1]$; $\vec{yba}_1 := 0 + x_1$; $\vec{q}_1 := nc = ma$; $nda_1 := nd_1$; QB_0 QC_0 QD_0 $fba_1 := fb_1$; $yba_1 := yb_1$; $mba_1 := mb_1$; $Q(q_1)[\vec{nda}_1, \vec{fba}_1, \vec{yba}_1, \vec{mba}_1; \vec{y}_0]$; Z_0 write y_0;

Table V — Execution Begin of a Parallel, Variable-free Program

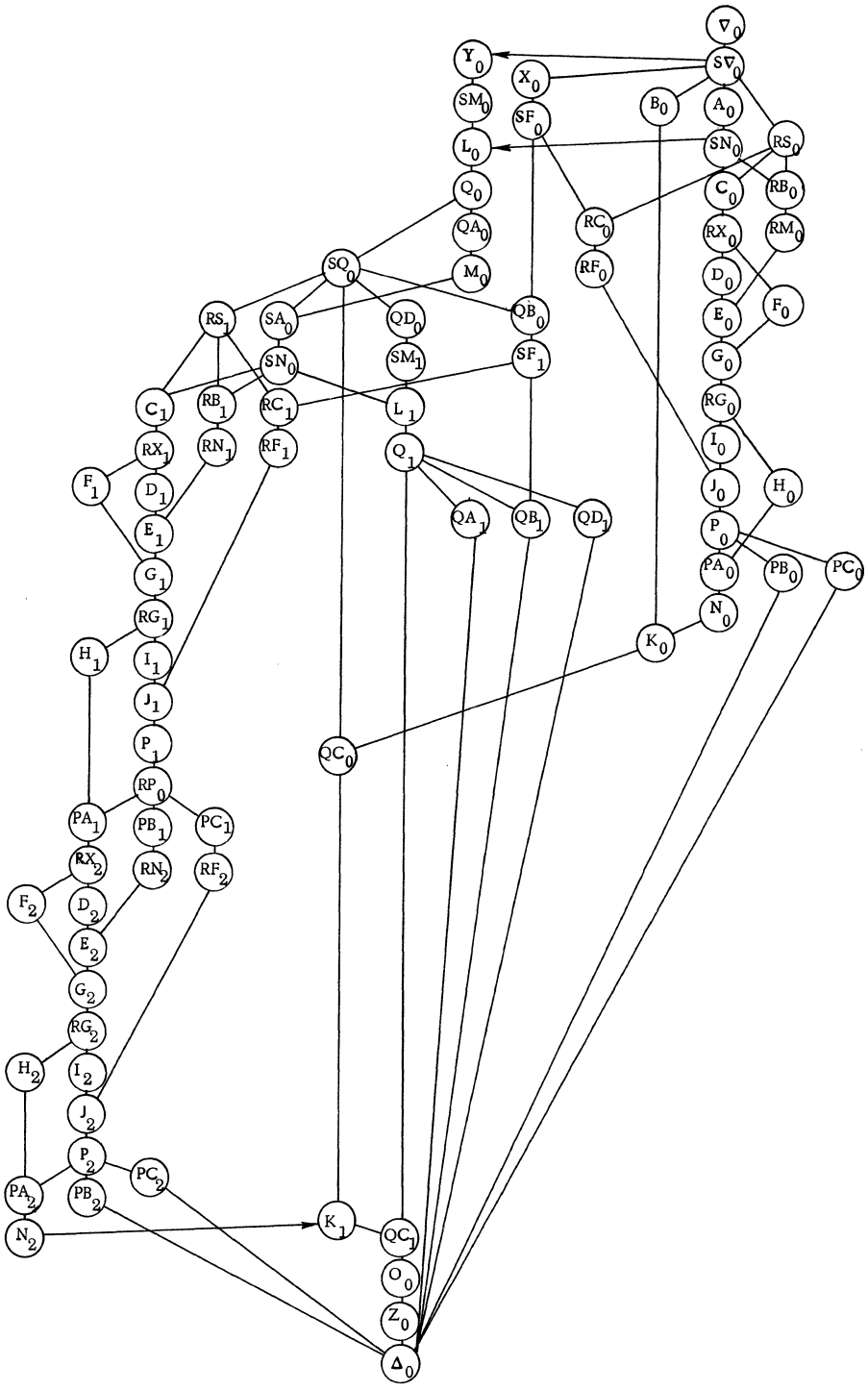


Fig. 3 - Precedence Graph, Describing Execution Possibilities of One Program Run

FORMAL TRANSFORMATIONS FOR PARALLEL PROCESSING LOGIC

Edward P. Stabler
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, N.Y. 13210

Abstract -- Formal transformations are described which convert sequential processes into parallel processes preserving the logical behavior. The formal transforms are carried out on a logic design language. The results of the transformations are alternative designs expressed in the same language. The technique is applied to several sample design problems.

Introduction

Languages for describing the structure of computers and other digital systems are receiving increased attention. The motivation for such activity is a hope that higher order languages for hardware structures will provide the same sorts of benefits in hardware design as programming languages provide for software design. In particular, a satisfactory system description language should provide a means for coping with the complexity found in typical logic systems.

The structural complexity of parallel processing systems is greater than that of serial processors. As a result the need for control of complexity is increased and the task is more difficult.

In this article the application of system description languages to parallel processor system is examined. The questions of interest are:

1. Can a system description language provide adequate compact and precise description of parallel processing logic networks?

2. What are the transformations which can be carried out on system descriptions which will affect the speed of operation (degree of parallelism) while preserving the essential logical behavior?

3. Can useful and economical designs for parallel processes be obtained utilizing formal transformations?

4. What is the relationship between such formal transformations on the logic and related transformations on programs?

The basic ideas behind the transformations required to increase parallelism of combinatorial and sequential logic designs are well-known [1]. However the implementation of these algorithms will depend critically on the representation system used for the design. Various methods of representing designs must be studied to determine the simplicity and efficiency of the operations to be carried out on the designs.

The language should provide an adequate data interface with other design automation programs for testing, layout, wiring, and simulation. The language described here is APL-based so it has the advantage of having a set of vector and array type operators.

The paper describes the principal features of a system design language and some design transformations within the languages. Several useful logic design examples are considered. Designs with a high degree of parallelism activity are studied to determine whether these designs could be generated by a straight-forward application of automatic design transformations.

The transformations are essentially logical in nature. A machine description is converted to a logically equivalent machine description where the derived machine exhibits more parallelism than the original.

The introduction of parallelism generally substitutes a spatial iteration of signals for the original time iteration. Hence the two machines are not equivalent in the sense usually used with respect to sequential machines. They are equivalent in the sense that there is a mapping of (output signal, time) of the original machine to (output signal, time) in the new machine which preserves the logical behavior of the machine.

Register Transfer Language

Many different notations have been suggested for describing systems. They can be divided into two broad classes, those which describe behavior and those which describe structure. The former type of description particularly useful for simulation while the latter is useful in design automation systems.

The language used here utilizes many APL features and is a register transfer language intended to describe the structure of a digital system. Since the descriptions tend to look like programs it is important to remember the differences between descriptions and programs.

1. A system description describes a structure and not a process;

2. The order in which the statements occur in a system description has no significance.

The designer using a system description is usually thinking in terms of the behavior of the system rather than its structure. The description can be viewed as a specification of behavior or of a process. In what follows, the description

will be considered to specify a structure and the transformations will be designed to derive alternative structures with equivalent behavior.

Kernel Language

The kernel language is the most primitive form of the language which is adequate to describe any system describable by the complete language. The strategy used here is to define a very simple kernel language whose properties are simple. Then complex linguistic facilities are added to the language and these facilities are defined by a translation process which eliminates the occurrence of a complex feature and yields an equivalent description in the kernel language. If this technique is used the complete sophisticated language can describe no more than the kernel language. However the complete language will generally allow vastly more compact system descriptions with no loss of precision. The same technique has been suggested for defining programming languages [2].

There are only five types of statement in the kernel language:

1. The conditional register transfer, A $\not$ B ← C, having the form <name > $\not$ <name > ←<name>
2. The synonym statement, A = B, having the form < name > = <name >
3. The AND statement, A = AND (B, C, D) having the form <name > = AND (< name list >)
4. The OR statement, A = OR (B, C, D) having the form <name >= OR (< name list >)

and 5. The NOT statement, A = NOT (B) having the form < name >= NOT (< name >)

With a sufficient number of statements in the kernel language any network of logic involving registers and logic gates can be represented. The form of the conditional transfer implies synchronous logic and the exact logic associated with the register input is unspecified. The kernel language cannot describe asynchronous objects such as delay lines and one-shots without the addition of new kernel statements.

All the sophisticated linguistic facilities which are added from this point on are defined by means of a translation process which eliminates complex structure and derives an equivalent set of kernel statements.

The kernel language is extended by

1. extension of naming to allow naming of vectors and arrays of higher dimensions,
2. extension of operations to apply to vectors and arrays
3. addition of programming language to allow computation to generate primitive language statements,
4. facilities for defining macro system descriptions with formal parameters,
5. facilities for declaring types such as register, arithmetic variables, etc., and
6. the addition of a set of functions whose values are related to the system description parameters.

With these extensions precise and compact descriptions of digital systems can be developed. The complex networks associated with MSI and LSI usually exhibit sufficient repetitive structure so that the facilities of the language can be used to good effect.

APL conventions are used to extend the range of operators to vectors and arrays. The macro facility corresponds to function definition within a programming language. The mention of a macro name with actual parameters specified calls for the addition of the text which is the body of the macro, with formal parameters replaced by actual parameters. A conventional programming language can be used to control the generation of text and the computation of literal subscripts. It is important to realize that the programming language portion of the system description is not used to define a program but to generate a body of text.

The addition of the sophisticated linguistic facilities does not extend the range of system which can be described. Each of the added linguistic types can be translated into an equivalent set of primitive statements. The technique has been proposed to simplify the concepts underlying conventional programming languages. The advantage is that the range meaning of a description is not changed by the sophisticated techniques of description. The description still corresponds to specification of a network of gates and registers.

However repeated use of macro system descriptions permit the design objects to correspond to more and more complex networks. The system description language provides desirable simplification of the description as long as there is some regularity and iterative structure in the network.

Register Transfers

The basic algorithm to be used for speeding up sequential logic has the effect of doubling the computational rate. The derived machine does in one clock cycle what the original machine does in two cycles. The equivalence relation between the two machines relates pairs of inputs, and outputs which occur in time sequence to pairs which occur in spatial sequence. The algorithm generates a set of combinatorial logic equations virtually identical to the original register transfer equations. The combinatorial logic equations generate the intermediate values of the register variables so a double time step is performed on each clock beat.

The following single statement is a description of binary counter.

```
1 $  A ← X
     NET(A:X)
```

```
∇ NET(B:C)          Definition of NET
  C = B + D
  NET2 (B:D)
∇

∇ NET2 (E:F)        Definition of NET2
  I ← 0
  F[I] = 1
G:  F[I + 1] = F[I] ∧ E[I]
  → ((I ← I + 1) < ρE)/G
∇
```

The definition of NET2 specifies a network with input E and output F as formal parameters. The value of F is 1 in all positions corresponding to consecutive 1's in E and in the position of the first 0. The diagram is shown in Figure 1a. The network defined by NET includes an occurrence of NET2 and has a bank of exclusive-or gates in addition. Hence, the total diagram is as shown in Figure 1b.

To describe the system which will count two for each unit of time it is only necessary to duplicate the network

```
1.  ∮  A ← Y
       NET(A:X)
       NET(X:Y)
```

yielding a net of the form shown in Figure 2.

Our example is a very simple one but the basic idea is the same in what follows. The next step in our simple example is to take advantage of the array naming features of the system description language. Consider the extension to an array of logic which causes the counter to count by N in each unit of time. A network generating macro call UNET can be used to replicate the network to form an array.

```
1  ∮  A ← W(;N)
      UNET(A: N: NET: W)

   ∇  UNET(a: n: net:w)
      i ← 0
      w[; 0] = a
   C:  net(w[; i]:  w[; i + 1])
   →  (n > i ← i + 1) / C
```

The macro UNET when mentioned generates the system description of a network of N binary counter networks connected end to end so that a count up by N occurs. Refer to Figure 3.

The example is a simple one involving only one register, no conditional transfers and no inputs. The transformation method can be extended to cover the more general case. A description of a serial adder is the two statements shown below. It is a slow serial adder since the shifting and the addition are not overlapped in time.

```
t ∮ S_n ← + (A, B, C); C ← MAJ(A, B, C);
        t ← ~ t
  t ∮ S ← ϕ S; t ← ~t
```

A and B are assumed to be input strings representing the numbers to be added.

The conditionals can be brought over to the right hand side to obtain

```
1 ∮  S ← X; C ← Y;  t ← t_1
  X = (t ∧ (A + B + C), 1 ↓ S) ∨ (~ t) ∧ ϕ S
  Y = (t ∧ MAJ(A,B,C) ) ∨ ~ t ∧ C
  t_1 = ~t
```

Networks with formal parameters can be defined

```
NET4  (a: b: c: s: t:x )

NET5  (a: b: c: s: t:y)
```

where the defining equations are essentially as above.

A and B are external input sequences for which subscripts can be used to designate successive inputs to an array. The macro definition of a network for doing n steps of the original machine is

```
∇ UNET2 (a: b: c: s: t: x: y: n)
   c1[0] = c
   i ← 0
 C:  NET4 (a[i]; b[i]: c1[i]: s1[i]: t[i]
          x[i])
     NET5 (a[i]: b[i]: c1[i]: s1[i]: t[i]
          y[i])
      c1[i + 1] = y1[i]
      s1[;i + 1] = x1[;i]
       t[i + 1] = t[i]
  →  (n > i ← i + 1) /C
     y = y1[i]
     x = x1[;i]
     ∇
```

A single mention of UNET2 with n = 2 will result in a logic network which overlaps in time the shifting and adding. It can be seen that actually two independent systems are formed. The first does a shift and add simultaneously while the other does an add and shift simultaneously. The network is a spatial sequence of combinatorial networks. Only one of the two networks will be active depending on the initial value of t.

This phenomenon of generating multiple systems is a general one in the transformation. The transformation generates n machines which differ from one another in phase. The initial conditions will normally cause a selection of one of the machines. For example, in the case of the serial adder are initial condition of t=1 selects the machine which adds and then shifts in the spatial iteration.

If the transformation is applied again with n = N an N bit parallel adder array is produced.

Combinatorial Logic

The combinatorial portion of system description corresponds to a set of boolean equations. The equations describe a multiple

input, multiple output network. Usually the designer will have utilized the facilities of the language to describe the iterative portions of the combinatorial logic but the description can be translated to a large set of primitive equations when necessary.

For networks of this type the maximum depth is defined as the maximum number of gates which must be passed through in going from an input through the network to an output. The transformations described have the effect of increasing this depth so that very large deep combinatorial nets can be generated for array type logic. The delay through the network is proportional to its depth and the delay can become a decisive faster in the overall speed of the system.

The algebraic identities needed to reduce the depth of a network of gates are well-known and various strategies can be utilized in transforming a network. Depth reduction transformations are shown in Figure 5. The process using DeMorgan's theorem is used to push the inverters through the AND, OR gates in order to produce subnetworks on which the processes of parenthesis removal or multiplying out can be performed. In practice a number of practical constraints must be observed. The transformations must not eliminate output wires and gates whose outputs drive more than one gate must be transformed with care. In addition there are normally fan-in and fan-out limits which will eventually be exceeded. Fan-out limits do not affect the achievable speed since network duplication can be used to provide the necessary number of outputs.

The structured nature of the system descriptions permits a kind of controlled reduction of the delay in the combinatorial portion of the network which is different from the technique of reducing to primitive statements and applying boolean algebra transformations. We expect that the large delay values will be generated by network forms such as shown in Figure 6. A combinatorial net is replicated and interconnected in such a fashion that the delay is proportional to the degree of replication. In the system description this would appear as a definition in which the outputs and inputs wires are connected according to some recursion formula.

To reduce the total delay there are two main choices; reduce the value of Δ the delay per network element, or form a new network element which need be replicated by a lower factor without increasing Δ by the same factor as shown in Figure 7. The replicated network is assumed to be arbitrary complexity.

If the system description for the combinatorial network is simply translated by macro substitution to form a large set of primitive gate statements, the iterative structure of the network is lost, or at least hidden. As a result the task of transforming the set of equations to an alternative set which has smaller depth cannot easily take advantage of the iterative structure. It is desirable to separate the two methods of reducing overall depth. In the first case an attempt is made to reduce the value of Δ, the delay associated with one element of the replicated network. This requires reduction of that element to primitive gate statements and the application of the boolean algebra transformations to reduce the delay. Having done the calculation once, the result can be used to realize the replicated elements.

The second case requires definition of a larger more complex network element so that the replication factor is reduced. Then the larger defined element is processed to reduce the depth and to reduce the total delay. The definition of the more complex network element can be obtained in a straight-forward way from the definitions of the orginal network.

Assume that the replication factor is to be divided by 2 by combining the functions performed by two elements. If the interconnection is simple linear one then the process proceeds as shown in Figure 8.

Assume NET(A; B: C: D) is defined and is a replicated element. Then if I, Z are n element vectors the linear interconnection can be defined by

```
∇ LINET(I:Z)

  NET(I: E: Z: F)
  E = 1 ϕ F
```

A double element equivalent to two linearly interconnected NET elements with linear interconnection would be NET2(A; B; C; D) and defined by

```
∇ NET2(A: B : C:D)

  NET(A[0]: B:  C[0]:  X)
  NET(A[1]: X:  C[1]:  D)
∇
```

The linear interconnection of four elements is shown in Figure 8b.
The maximum delay is unaffected by the change. The advantage of the new form is that NET2 can now have its delay reduced using the boolean transformations of the primitive statements corresponding to NET2. The complete network is can be described by a n/2 replication of NET2. For the more general case combining N networks into a single element UNETN may be defined which consists of N of the original elements with the internal connections defined by the recursion formula of the original network.

The main steps in forming NETN are given below:

1. Replicate NET

   NETN = N ρ NET

2. Form internal connections

from the definition of casade structure we have:

D[I] = B[f(I)]

where typically f[I] = I + CONST
Hence, an internal connection is specified if

(f(I) ÷ N) = 0 for I < N

otherwise an external connection is needed.

3. Form a Linear Structure of NETN elements

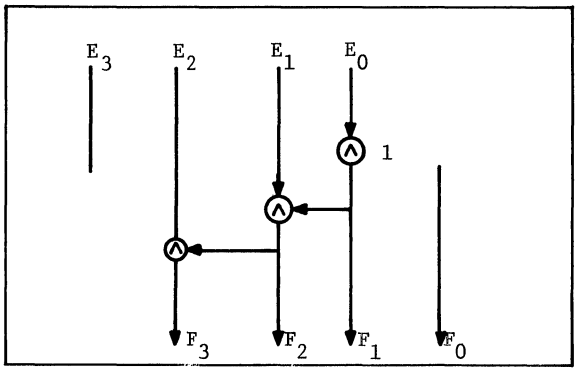
LS = K ρ NETN

4. Form of Connections between the NETN elements

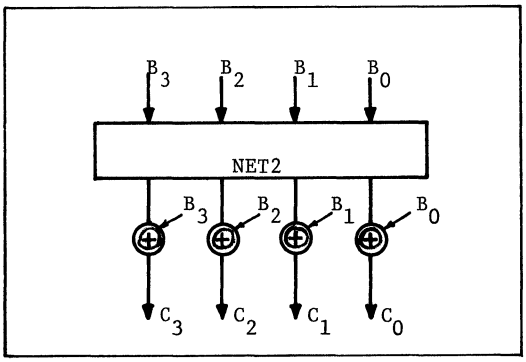
D[I] = B[f(I)]

D[L ; M] = B[f(M*N+L) | N] [f(M*N+L)÷6]

Which includes previously defined internal connections.



Form of NET2
Figure 1a

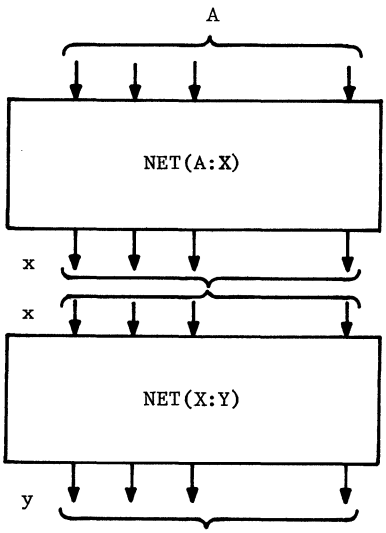


Form of NET
Figure 1b

Comparisons

The processes described here are intended for use in a software system to aid the logic designer. The techniques shown are quite different from those which are under study for parallel programming and for parallel organization of computing systems. In parallel programming studies a basic control mechanism is assumed and parallelism consists of allowing two or more controllers to proceed more or less independently. The logical and arithmetic processes being performed are considered only to the extent that they influence the flow of control. In register transfer system descriptions no real distinction is made between control and processing activity although such distinctions may play an important role in the thinking of the designer.

When a logical transformation is performed on the system to speed it up the logical networks are replicated if there is no possibility for concurrent operation. However the network is not replicated if concurrent operation is possible. This is illustrated in the example of the serial adder in the paper. The first speedup caused an time overlapping of shift and add with essentially no increase in hardware. Further transformations to create parallel addition required replication of the basic adding network.



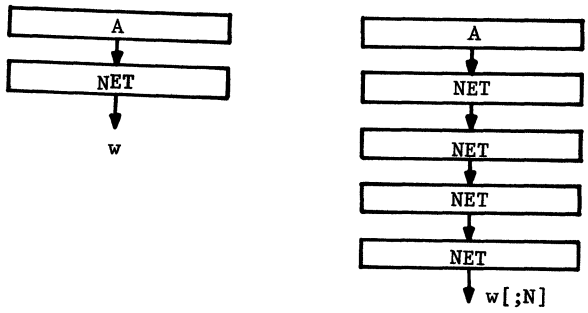
Double Step Counter
Figure 2

References

[1] F.C. Hennie, Finite State Models for Logical Machines, John Wiley, 1968.

[2] A. van Wijngaarden, "Recursive Definition of Syntax and Semantics", Formal Language Description Languages, North Holland, 1966.

[3] O.L. MacSorley, "High Speed Arithmetic in Binary Computers", PIRE, January 1961, pp. 67-91.



N Step Counter
Figure 3

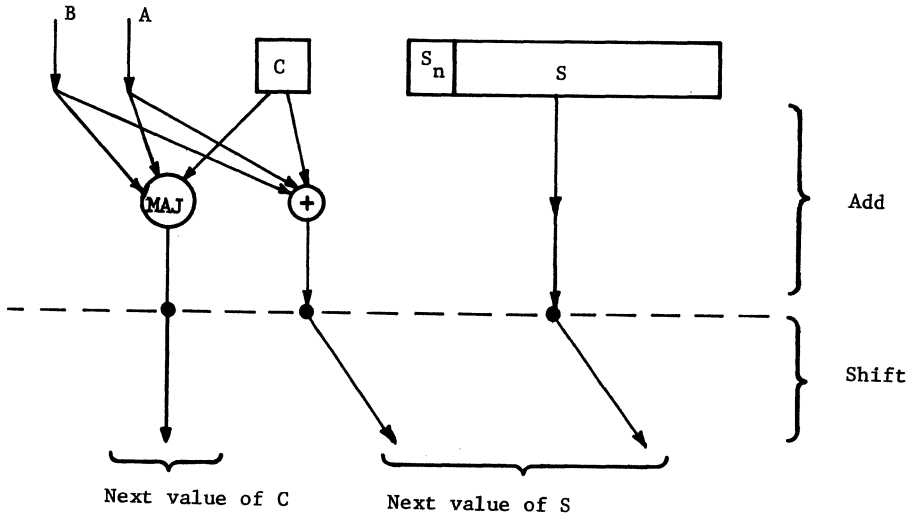
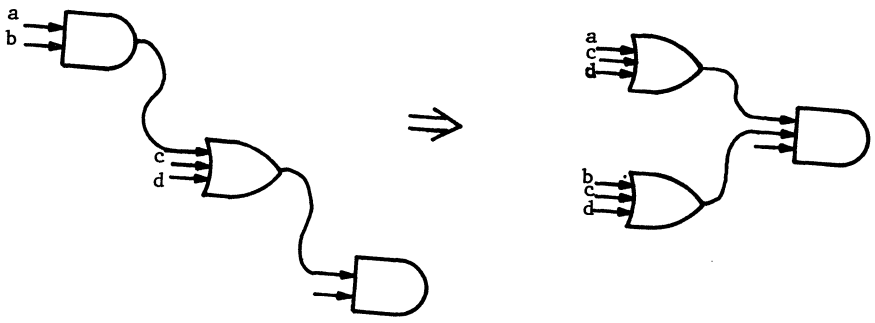
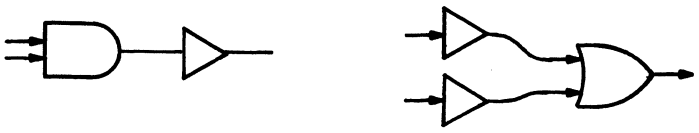


Figure 4. Add and Shift Spatial Sequence



5a. Multiplying Out



5b. DeMorgan's Theorem
Figure 5. Depth Reduction for Combinatorial Nets

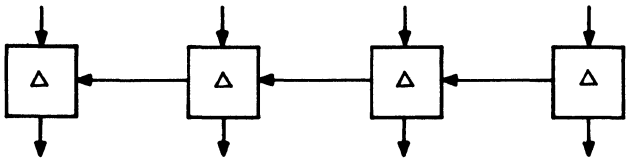


Figure 6. Network Form with Large Delay

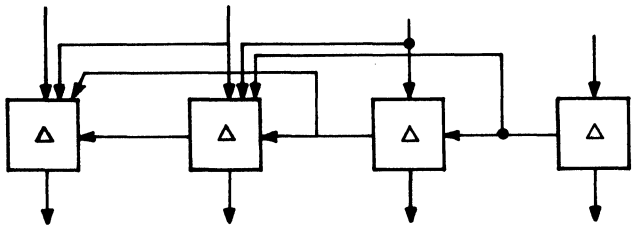


Figure 7. Network Form with Reduced Delay

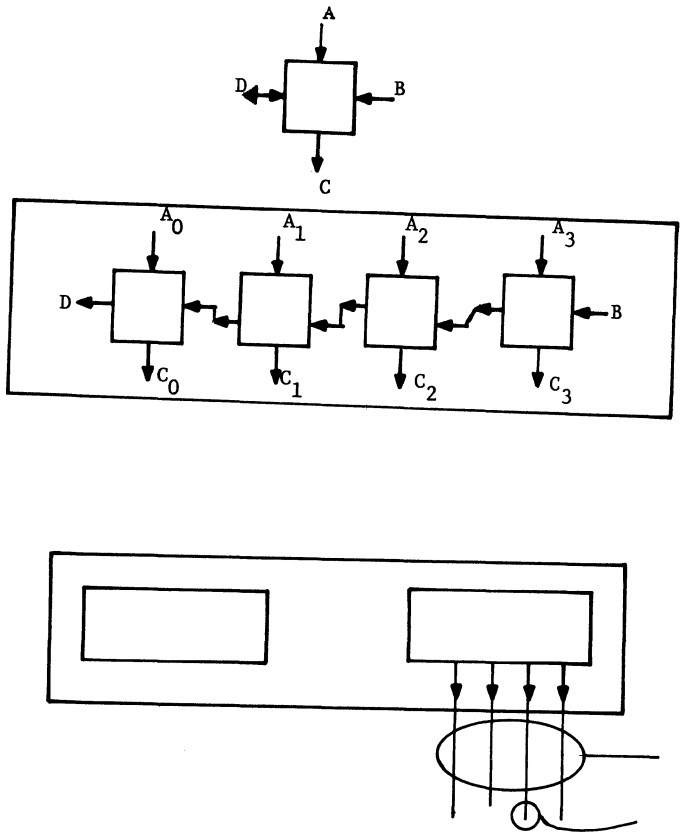


Figure 8. Reduction of Replication Factor

A STRUCTURED APPROACH TO CONCURRENT PROCESS SYNCHRONISATION

Santosh K. Shrivastava[a]
Computer Laboratory, University of Cambridge, England

Summary

This paper briefly describes a concurrent process synchronisation method to be used with secretaries [1] or monitors [2]; an operating system structuring concept developed by Dijkstra and Hoare.

The well known example of readers and writers [4] is used below to illustrate the method and the monitor concept.

```
file:monitor;
begin rr, aw: shared integer; free: shared boolean;
nowriter:condition (aw = o);
noreader:condition (rr = o & free);
procedure startread;
begin await nowriter; with rr do rr:=rr+1; end
procedure endread;
begin switch:boolean; switch:=false;
with rr do begin rr:=rr-1;
if rr=o then switch:=true; end
if switch then test noreader;
end endread;
procedure startwrite;
begin with aw do aw:=aw+1; await noreader;
with free do free:=false;
end startwrite;
procedure endwrite;
begin switch:boolean; switch:=false;
with free, aw do begin free:=true;
aw:=aw-1; if aw=o then switch:=true; end
if switch then testall nowriter else test noreader
end endwrite;
with aw,rr, free do begin aw,rr:=o; free:=true;end
note give initial values;
end file;
```

A file is to be used for reading or writing. Any number of 'readers' may read simultaneously, but 'writer' must have exclusive use; further, writers are given priority.

Calls on a monitor procedure are of the form: monitor name.procedure name (--parameters--); Thus, the readers will use the code: file.startread; 'read operation'; file.endread; to use the file. A monitor is treated as a critical region so that processes have exclusive use of it. A 'condition variable' represents some condition for the resource use, expressed as a boolean expression involving the monitor variables. With each condition variable we also associate at compile time, (a) two boolean variables 'state' and 'current', when 'current' is true, the value of 'state' is taken to represent the value of the expression, when 'current' is false, this is not so, and (b) a queue for waiting processes. The operation 'await condition name' is defined as follows: if 'current' and 'state' of that condition variable are true, the executing process continues; if 'current' is true and 'state' is false the process releases the monitor exclusion and waits on that condition's queue; if 'current' is false, the process evaluates the expression and sets 'state' accordingly, 'current' is made true, the process now continues or waits as described above. The operation 'test condition name' is defined as follows: if 'current' and 'state' of that condition variable are true, the executing process removes one waiting process (if any) from the condition's queue and puts it on the queue of processes trying to enter the monitor; of 'current' is false, the process evaluates the expression, sets 'state' accordingly, 'current' is made true, if 'state' is now true, a waiting process is scheduled as described above. A 'testall condition name' operation is similar, except that instead of one, all the waiting processes are scheduled. A resumed process, when given entry to the monitor, reexamines the 'await' condition as described. The monitor variables that occur in the condition expressions are declared shared, operations on a shared variable are permitted only through the notation 'with shared variable name do S.' This operation is defined as follows: all the 'current' bits of the condition variables, condition expressions of which refer to that shared variable, are made false, then S is executed. No 'test' or 'await' is permitted inside S.

It is now easily seen that in this synchronisation method, evaluation of condition expressions is kept to a minimum. Thus, when readers are reading, the first writer to find this will make 'state' of 'noreader' condition false and 'current' true. Any other writers entering the monitor consequentively now, do not evaluate 'noreader' to find out that they must wait. As conditions can be arbitrarily complex, when resources are heavily utilized, this method particularly becomes attractive.

The method can be incorporated in high-level software writing languages with the monitor concept. A detailed evaluation of various synchronisation techniques, including the existing proposals [2,3] and parallel programming techniques using monitors will appear in the author's Ph.D. thesis.

References

[1] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes", Acta Informatica 1 (1971), pp. 115-138.

[2] C.A.R. Hoare, "Monitors: an Operating System Structuring Concept", (to be published).

[3] P.B. Hansen, "Structured Multiprogramming" C.A.C.M. (July 1972), pp. 574-578.

[4] P.J. Courtois, F. Heymans, D.L. Parnas, "Concurrent Control With Readers and Writers" CACM (October 1971).

(a) On leave from the Plessey Co. Ltd. Poole, Dorset, England.

PARALLELISM IN TAPE-SORTING

Shimon Even†
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

Abstract -- Two methods for employing parallelism in tape-sorting are presented. Method A is the natural way to use parallelism. Method B is new. Both approximately achieve the goal of reducing the processing time by a divisor which is the number of processors.

I. Introduction

It is reasonable to assume that one is willing to use P processors instead of one if the computation time is cut down by the same factor. In certain applications this has been shown to be impossible.

Fortunately, this kind of saving in time is possible in the case of external sorting. Two methods for achieving this goal are described. The first one is natural and uses known techniques. The second method uses new ideas and is believed to be more elegant and easier to program.

The description is in terms of tapes, but any linear mass storage can be used instead.

II. Method A

Assume we have N records, P processors and $4P$ tapes. Also assume that initially the N records are all stored on one tape. The sorting is achieved through the following steps:

(1) The N records are distributed to $2P$ tapes in such a way that each of them has approximately $N/2P$ records. This step takes N units of time.

(2) Every one of the P processors is assigned 4 tapes: two of them are loaded, with $N/2P$ records on each, and two are empty. Each processor performs the well-known algorithm of tape-sorting using the 4 tapes it controls. (For a few more details see the Appendix.) This step takes

$$\frac{N}{P} \log_2 \frac{N}{P}$$

units of time.

(3) We now have P tapes which are each loaded with a group of N/P records and the records on each of these tapes are sorted. We perform $\log_2 P$ phases of sort through merge. In the first phase every two groups are sort-merged into a group of $2N/P$ records. In the i-th phase every two groups of

$$\frac{2^{i-1}N}{P}$$

records are sort-merged into one group of

$$\frac{2^{i}N}{P}$$

records, etc. The whole process takes

$$\sum_{i=1}^{\log P} 2^i \frac{N}{P} = 2N - \frac{2N}{P}$$

units of time. We conclude that the time method A takes is

$$\frac{N}{P} \log_2 N - \frac{N}{P}(\log_2 P + 2) + 3N \ . \qquad (1)$$

The method can be used even with a very large P. In the extreme case when $P = N/4$ the sort time reduces to $3N$. However, the more practical cases are when $P \leq \log_2 N$, when (1) is well approximated by

$$\frac{N}{P} \log_2 N + 3N \ . \qquad (2)$$

Except for the $3N$ term this achieves the best possible saving; namely, the best sorting time for one processor, which is $N \log_2 N$, is divided by the number of processors.

III. Method B

For simplicity, let us assume first that N is a power of 2 and that the number of processors available is

$$P = \log_2 N + 1 \ .$$

We shall use $4P$ tapes (in addition to the input tape). As we shall see later, the number of processors can be reduced to $\log_2 N$ and the number of tapes to $4(\log_2 N - 1)$.

The tapes are divided into quadruples:

$$T_1^i, T_2^i, T_3^i, T_4^i$$

† Visiting at the Department of Computer Science, Cornell University, Ithaca, New York, summer 1973.

for $i = 1,2,\ldots,P$. Time is measured in the unit of time necessary for reading and writing one record. The processors are denoted by $\Pi_1, \Pi_2, \ldots, \Pi_P$.

During time $1 \leq t \leq N$ Π_1 reads the input tape and writes the records on

$$T_1^1,\ T_2^1,\ T_3^1 \text{ and } T_4^1,$$

according to the following rule:

(i) if $t \equiv 1 \pmod 4$, Π_1 writes on T_1^1,

(ii) if $t \equiv 2 \pmod 4$, Π_1 writes on T_2^1,

(iii) if $t \equiv 3 \pmod 4$, Π_1 writes on T_3^1,

and

(iv) if $t \equiv 0 \pmod 4$, Π_1 writes on T_4^1.

Π_k is active during

$$2^k - 1 \leq t \leq N + 2^k - 2 .$$

For $k = 2,3,\ldots,P$ its activity is as follows: It reads from tapes of the (k-1)st quadruple and writes on tapes of the k-th quadruple. It performs, repeatedly, a sort-merge of two sorted lists of length

$$2^{k-2}$$

into one sorted list of length

$$2^{k-1} .$$

The tapes are used according to the following rule:(a)

(i) if

$$\left\lceil \frac{t-2^k+2}{2^{k-1}} \right\rceil \equiv 1 \pmod 4$$

then Π_k reads from

$$T_1^{k-1} \text{ and } T_2^{k-1}$$

and writes on

$$T_1^k ,$$

(ii) if

$$\left\lceil \frac{t-2^k+2}{2^{k-1}} \right\rceil \equiv 2 \pmod 4$$

then Π_k reads from

$$T_3^{k-1} \text{ and } T_4^{k-1}$$

and writes on

$$T_2^k ,$$

(iii) if

$$\left\lceil \frac{t-2^k+2}{2^{k-1}} \right\rceil \equiv 3 \pmod 4$$

then Π_k reads from

$$T_1^{k-1} \text{ and } T_2^{k-1}$$

and writes on

$$T_3^k , \text{ and}$$

(iv) if

$$\left\lceil \frac{t-2^k+2}{2^{k-1}} \right\rceil \equiv 0 \pmod 4$$

then Π_k reads from

$$T_3^{k-1} \text{ and } T_4^{k-1}$$

and writes on

$$T_4^k .$$

An example of $N = 8$ is shown in the diagram on the next page. Successive rows represent successive time. A solid line in the column

$$T_i^k$$

in row t means that Π_k is writing on

$$T_i^k$$

during time t; a broken line means that

$$\Pi_{k+1}$$

may be reading from it.

The reader may establish for himself the validity of the following claims:

(1) Every tape is emptied (the records it has contained are read) before it is loaded again with a sorted list. Thus, a tape of the k-th quadruple never contains more than

$$2^{k-1}$$

records.

(2) The reason for the difference in the starting times is that Π_k is waiting for

$$\Pi_{k-1} \text{ to load } T_1^{k-1} \text{ and } T_2^{k-1} .$$

This takes 2^{k-1} units of time. Thus the starting time is

$$2^{k-1} - 1 + 2^{k-1} ;$$

namely, $2^k - 1$.

(a) Let $\lceil x \rceil$ denote the least integer which does not exceed x, i.e. $\lceil 3.5 \rceil = \lceil 4 \rceil = 4$.

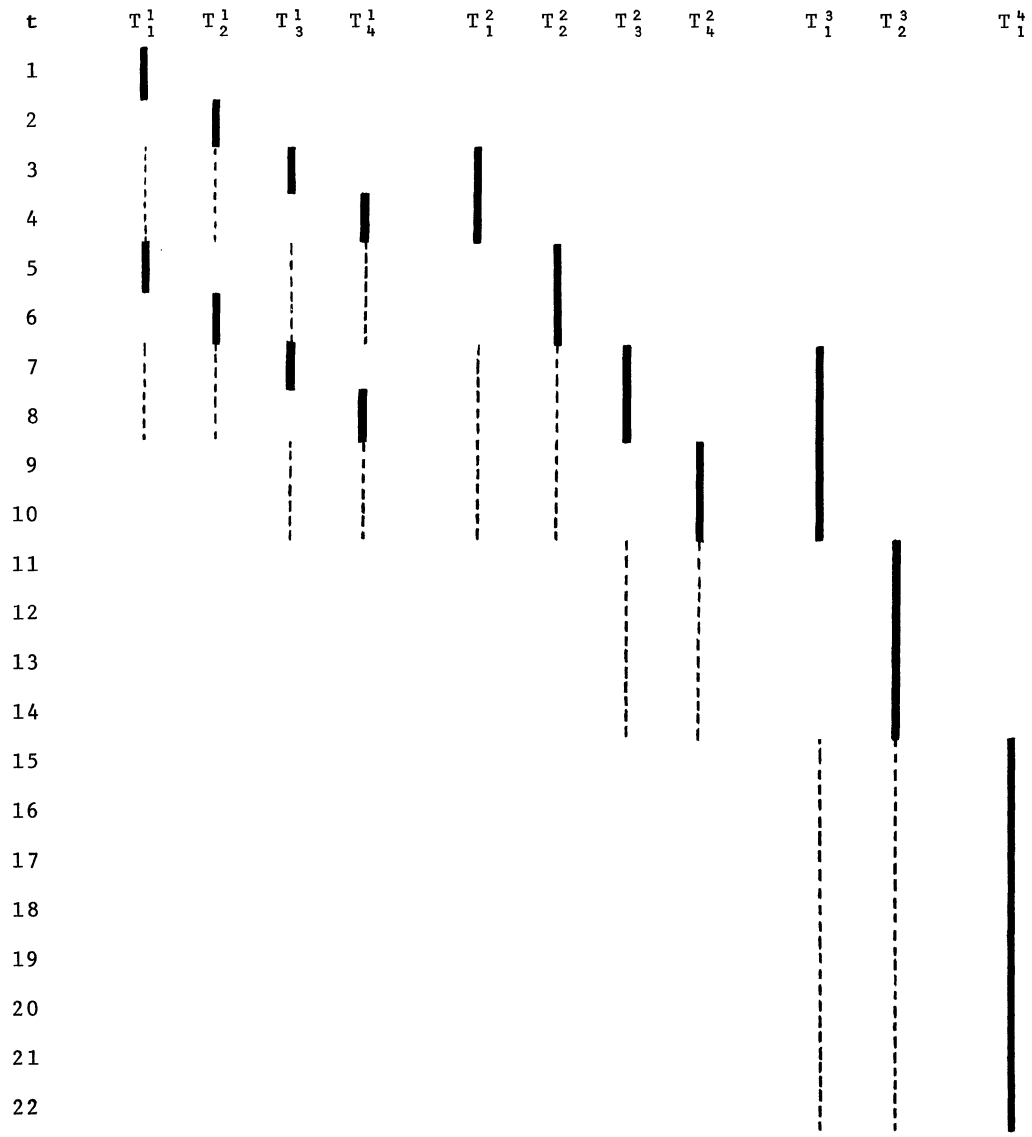


Figure 1.

(3) T_4^{P-2} is first loaded during

$$t = 2^{P-2} - 1 + 3 \cdot 2^{P-3} \quad .$$

Thus,

$$t = 2^{P-1} + 2^{P-3} - 1, \text{ or}$$

$$t = \frac{5}{4}N - 1 \quad .$$

For $N > 4$ this is larger than N. Since T_1^1 is not used after $t = N$, we can use the same tape for both tasks.

(4) T_2^{P-1} is first loaded during

$$t = 2^{P-1} - 1 + 2^{P-2} \quad .$$

Thus,

$$t = \frac{3}{2}N - 1 \quad .$$

For $N > 2$ this is larger than N. Since T_2^1 is not used after $t = N$, we can use the same tape for both tasks.

(5) T_3^{P-1} and T_4^{P-1} are never used.

(6) T_1^P is first loaded during

$$t = 2^P - 1 = 2N - 1 \quad .$$

For $N > 3$ this is larger than $N + 2$. Since T_3^1 is not used after $t = N + 2$ we can use the same tape for both tasks.

(7) T_2^P, T_3^P and T_4^P are never used.

(8) Claims (3) to (7) imply that only $4(\log_2 N-1)$ tapes are necessary.

(9)

$$\Pi_1 \quad \text{and} \quad \Pi_p$$

can be the same. The reasons are as in (6). Thus, only $\log_2 N$ processors are required.

(10) During $t = N$ there are $\log_2 N$ processors in action and $4(\log N-1)$ tapes are occupied. Thus, no further saving is possible unless basic changes are made in the procedure.

The whole process takes $3N-2$ units of time, and

$$T_1^P$$

is the output tape. This compares favorably with Method A (see (2)) which requires approximately $4N$ units of time in case $P = \log_2 N$. In my opinion, Method B is more elegant and easier to implement.

Let us now discuss the case $P \neq \log_2 N$. (The reader should notice that we have started with $P = \log_2 N + 1$ but have reduced the number of processors to $\log_2 N$).

Method B is not suitable for using much more than $\log_2 N$ processors. Clearly, when $\log_2 N$ is not integral we can use

$$\lceil \log_2 N \rceil$$

processors and pretend we have

$$2^{\lceil \log_2 N \rceil}$$

records by filling in "dummy records". (Some improvements on this are possible but essentially the processing time is

$$3 \cdot 2^{\lceil \log_2 N \rceil} - 2 \quad .$$

A similar problem occurs in Method A if P is not a power of 2.) However, there is no way to use more than

$$\lceil \log_2 N \rceil$$

without changing the method considerably.

More interesting is the case when $P < \log_2 N$. For simplicity, let us discuss the case of

$$N = 2^{(P-1)Q}$$

where both P and Q are positive integers. Let

$$M = 2^{P-1} \quad .$$

The computation is done in Q passes. In each pass the output tape contains output lists which are M times longer than before. The number of processors used is P and the number of tapes is $4P - 2$. (Ignore here the savings discusses in Claims (3) to (10); only

$$T_3^P \quad \text{and} \quad T_4^P$$

are not needed.)

In the first pass we use the same procedure as discussed before, except that after

$$T_1^P$$

is loaded with a sorted list of length M another sorted list is loaded next to it, etc. This continues until all N records are on

$$T_1^P$$

in sorted lists of length M.

In the second pass

$$T_1^P$$

is used as the input tape and

$$T_2^P$$

as the output tape. The length of the list on

$$T_i^k \quad , \quad k < P \quad , \text{ is } \quad M\,2^{k-1}$$

and the timing is now in multiples of M. The lists on

$$T_2^P$$

are now of length M^2.

After Q passes the sorting is complete.

The i-th pass takes

$$N + M^{i-1}(2^P - 2)$$

units of time. Thus, the total time is

$$Q \cdot N + (2^P-2) \sum_{i=1}^{Q} M^{i-1}$$

$$= Q \cdot N + (2^P-2)\,\frac{M^Q-1}{M-1}$$

$$= Q \cdot N + 2(M^Q-1)$$

$$= \frac{N \cdot \log_2 N}{P-1} + 2(N-1) \qquad (4)$$

which is similar to (2).

The method can be improved by starting the next pass before the present one is over. However, this will only reduce

the second term of (4).

Appendix: Tape-Sorting by One Processor and Four Tapes

This well-known and widely used algorithm runs as follows: Assume the records are stored on two tapes, T_1 and T_2, each containing $n/2$ records, while the other two tapes, T_3 and T_4 are empty. Also, assume the data on T_1 and T_2 is already partially sorted in the following way: The $n/2$ records are divided into groups of

$$2^i$$

records. Each of these groups is already sorted, say from low to high, and the groups are stored consecutively on the tape. Thus, the number of groups on each tape is

$$\frac{n}{2^{i+1}} .$$

Initially $i = 0$. For simplicity, let us assume that

$$n = 2^{\ell} .$$

The algorithm goes through ℓ Phases.

In Phase 1 we read the first record from each input tape (T_1 and T_2) and store both on T_3 in increasing order. Next we read the second record from each input tape and store both on T_4 in increasing order. Next we return to load a group of two on T_3, etc. After n units of time (since each record is read once and is written once) all the records are distributed to T_3 and T_4 in ordered groups of size 2 ($=2^1$).

In Phase i, $i < \ell$, we perform a merge of the two groups which are presently on top of the two input tapes and store the merged double size group on one of the output tapes, alternatively. (If i is odd then T_1 and T_2 are the input tapes and T_3 and T_4 are the output tapes; if i is even, tasks are reversed.) The merging of these two groups is achieved by reading the top record from each group and writing the smaller one on the output tape. After each such writing the top record from the same group, as the one which has just been written, is read and compared with the record still in memory, etc. This is continued until one of the groups is exhausted; the remainder of the other group is directly transferred to the output tape.

We continue merging groups of size

$$2^{i-1} ,$$

one from each input tape, into groups of size

$$2^i$$

which are stored on the output tapes, changing the output tape after each group.

In Phase ℓ, there is one sorted group of size

$$2^{\ell-1} = \frac{n}{2}$$

on each input tape and they are merged into one sorted group of size n which is stored on one of the output tapes.

The whole operation takes $n \log_2 n$ units of time.

A PARALLEL ALGORITHM FOR MAXIMUM FLOW PROBLEM

Yu K. Chen and Tse-yun Feng
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, New York 13210

Summary

This algorithm is developed for solving the maximum flow problem in an associative processor. It is based upon the matrix multiplication approach [1] for finding the flow route. Given a capacity matrix $C^1 = [c_{ij}]$, with c^1_{ij} to be the capacity from node i to node j and $c^1_{ii} = 00$ for all values of i, and $C^{1(a)}_1$ to be the first row of C^1, one can generate C^2_1, C^3_1,..., to C^m_1 successively by the matrix multiplication $C^m_1 = C^{m-1}_1 \times C^1$, where the ordinary matrix product is performed with the following modifications: (1) $c_{ik} \cdot c_{kj} \equiv \min (c_{ik}, c_{kj})$, and (2) $\sum_k c_{ik} \equiv \max_k (c_{ik})$. Under the new definitions of matrix multiplication, c^m_{1i}, the i's element of C^m_1, clearly represents the maximum flow between the source and the node i by means of paths which have m branches or less. The multiplication process stops either when $c^m_{1n} \neq 0$ or when $m = n - 1$. Unlike the previously proposed sequential labeling methods [2] - [3] that the trace of the path has to be carried out along with the labeling process, the construction of a trace matrix $T = [t_{ij}]$ proposed here can be performed at the conclusion of the matrix multiplication. Matrix T is a zero-one matrix. $t_{ij} = 1$ if $c^m_{1j} = c^{m-1}_{1i} \cdot c_{ij} > 0$, and $i \neq j$; otherwise, $t_{ij} = 0$. Matrix T contains one or more paths. To select a single path matrix $P = [p_{ij}]$, backward trace technique can be used. The algorithm is designed to fully utilize the word-parallel and the fast search-retrieval capabilities of the associative processor to gain execution speed. A few transpose operations are required in this algorithm. Therefore, if a data manipulator [4] with the transpose function in it is provided will certainly help the execution speed. The multi-terminal network flow [5] - [6] is not considered. This algorithm has been coded in APL to emulate its execution in associative processor [7]. Results are compared with the algorithm proposed in [8]. Approximately two-to-one improvement in execution time is indicated.

(a) Node 1 is assumed to the source node and node n to be the sink node

References

[1] M. Pollack, "The maximum capacity through a network", OR 8(1960), pp. 733-736

[2] L.R. Ford, Jr. and D.R. Fulkerson, "Maximal flow through a network", Canadian J. Math 8(1956), pp. 399-404

[3] L.R. Ford, Jr. and D.R. Fulkerson, "A simple algorithm for finding maximal network flows and an application to the Hitchcock problem", Canadian J. Math 9(1957), pp. 210-218

[4] T. Feng, "A versatile data manipulator" Proceedings of the Sagamore Computer Conference, 1973

[5] R.E. Gomory, and T.C. Hu, "Multi-terminal network flows", J. SIAM Vol. 9, No. 4 (1961) pp. 551-570

[6] W. Mayeda, "Terminal and branch capacity matrices of a communcation net", IRE Trans. on Circuit Theory 7(1960), pp. 251-269

[7] Y.K. Chen, Ph.D. dessertation, in preparation

[8] V.A. Orlando, "Associative processors in the solution of network problems", Ph.D. dessertation, Syracuse University, 1972

PARALLEL - SEQUENTIAL PROCESSING OF FINITE PATTERNS*

William I. Grosky and Frank Tsui
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30341

Abstract -- The various basic conditions under which parallel, sequential and mixed parallel-sequential processing in tessellation structures, using the same local transformations, are equivalent in terms of pattern generation are studied. Various necessary conditions and sufficient conditions for equivalence are derived. We then illustrate a 'mutually destructive' condition where sequential and parallel processing cannot be made equivalent, and study this condition further. We finally relax some of our hypotheses and countenance the notion of simulation between parallel, sequential and mixed parallel-sequential processing, giving sufficient conditions for such simulations to exist.

Several recent research efforts are aimed at strengthening the theoretical understanding of parallel and sequential modes of picture and pattern processing [1, 2, 4, 5, 6, 9]. Rosenfeld and Pfaltz [7] have shown that any picture transformation that can be accomplished by a series of parallel local operations with Moore neighborhood index can also be accomplished by a series of sequential local operations with Moore neighborhood index, and conversely; but, the local operations may be different for the two types of processing.

In this paper, we first concentrate our investigation on the equivalence of parallel, sequential and mixed parallel-sequential local operations of arbitrary neighborhood index in arbitrary dimensions, where the local operator is the same for each mode of processing. We then relax this latter condition and explore the notion of simulation in general. The methodology used in this work is that of tessellation automata. We have generalized previously formulated definitions of these entities to take into account sequential processing, and we call our new entities the class of stratified mixed mode tessellation automata.

Stratified Mixed Mode Tessellation Automata

Definition 1: For $n \geq 1$, an n-dimensional stratified mixed mode tessellation automaton, TA, is a 4-tuple $\langle S, Z^n, NI, GT \rangle$, where,

1) S is a finite, non-empty set of states

2) Z^n is the set of n-tuples of integers. For $\underline{\zeta} \in Z^n$, we call $\underline{\zeta}$ a cell of TA. The set $CON = \{g \mid g: Z^n \to S\}$ is called the set of configurations of of TA

3) NI is an ordered q-tuple of elements of Z^n, for some $q \geq 1$, and is called the neighborhood index of TA. Suppose $NI = \langle \underline{\gamma}_1, \ldots, \underline{\gamma}_q \rangle$. Then, for $\underline{\zeta} \in Z^n$, $Ne(\underline{\zeta}) = \langle \underline{\zeta}+\underline{\gamma}_1, \ldots, \underline{\zeta}+\underline{\gamma}_q \rangle$ is called the neighborhood of $\underline{\zeta}$.

4) $GT \neq \emptyset$, called the set of global transformations, is a finite subset of CON^{CON} which is the union of GT_p, GT_s, $GT_{p,s}$ and $GT_{s,p}$, the sets of parallel, sequential, parallel-sequential and sequential-parallel global array transformations, defined as follows,

a) Suppose $\rho \in GT_p$ and let $c \in CON$. Then $\rho(c) = c' \in CON$, where, for some $\sigma_\rho : S^q \to S$, called a local transformation, we have, for each $\underline{\zeta} \in Z^n$, that $c'(\underline{\zeta}) = \sigma_\rho(c(\underline{\zeta}+\underline{\gamma}_1), \ldots, c(\underline{\zeta}+\underline{\gamma}_q))$. c' is called the successor configuration of c with respect to ρ. Thus, the state of a particular cell in a successor configuration of c depends on the states of the neighborhood of that cell in configuration c.

b) Suppose $\rho \in GT_s \cup GT_{p,s} \cup GT_{s,p}$. Then $\rho(c) = c' \in CON$, where, for some $\sigma_\rho : S^q \to S$ and $\tau_\rho \in \bigcup_{j \leq \omega} (Z^n)^j \cup (Z^n)^\omega$, for τ_ρ injective, called a trajectory, we now define $c'(\underline{\zeta})$ for $\underline{\zeta} \in Z^n$,

i) Suppose $\rho \in GT_s$. Then we require τ_ρ to be surjective as well as injective. For $0 \leq i \leq \tau_\rho^{-1}(\underline{\zeta})$, define $c_i^{(\underline{\zeta})} \in CON$ as follows,

$$c_0^{(\underline{\zeta})} = c$$

For $0 \leq k \leq \tau_\rho^{-1}(\underline{\zeta})-1$ and $\underline{\xi} \in Z^n$,

$$c_{k+1}^{(\underline{\zeta})}(\underline{\xi}) = \begin{cases} \sigma_\rho(c_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_1), \ldots, c_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_q)), & \text{if } \underline{\xi} = \tau_\rho(k) \\ c_k^{(\underline{\zeta})}(\underline{\xi}) & \text{otherwise} \end{cases}$$

Then, $c'(\underline{\zeta}) = c_{\tau_\rho^{-1}(\underline{\zeta})}^{(\underline{\zeta})}(\underline{\zeta})$

ii) Suppose $\rho \in GT_{p,s}$. Then we require τ_ρ to be non-surjective. Define $c^* \in CON$ by

$$c^*(\underline{\xi}) = \begin{cases} \sigma_\rho(c(\underline{\xi}+\underline{\gamma}_1), \ldots, c(\underline{\xi}+\underline{\gamma}_q)) & \text{if } \underline{\xi} \notin \text{range}(\tau_\rho) \\ c(\underline{\xi}) & \text{otherwise} \end{cases}$$

* This work was supported in part by NSF Grant GN-655

Then,

$$c'(\underline{\zeta}) = \begin{cases} c^*(\underline{\zeta}) & \text{if } \underline{\zeta} \notin \text{range}(\tau_\rho) \\ c^*(\underline{\zeta})\,(\tau_\rho|_{\text{range}(\tau_\rho)})^{-1}(\underline{\zeta}) & \text{if } \underline{\zeta} \in \text{range}(\tau_\rho) \end{cases}$$

iii) Suppose $\rho \in GT_{s,p}$. Then we require τ_ρ to be non-surjective. Define $c^{**} \in CON$ by,

$$c^{**}(\underline{\xi}) = \begin{cases} c(\underline{\xi})\,(\tau_\rho|_{\text{range}(\tau_\rho)})^{-1}(\underline{\xi}) & \text{if } \underline{\xi} \in \text{range}(\tau_\rho) \\ c(\underline{\xi}) & \text{if } \underline{\xi} \notin \text{range}(\tau_\rho) \end{cases}$$

Then,

$$c'(\underline{\zeta}) = \begin{cases} c^{**}(\underline{\zeta}) & \text{if } \underline{\zeta} \in \text{range}(\tau_\rho) \\ \sigma_\rho(c^{**}(\underline{\zeta}+\underline{\gamma}_1),\ldots,c^{**}(\underline{\zeta}+\underline{\gamma}_q)) & \text{if } \underline{\zeta} \notin \text{range}(\tau_\rho) \end{cases}$$

In the above three cases, the trajectory indicates the sequential order in which the cells of Z^n are processed. Case i) is pure sequential processing in which the state of a particular cell in a successor configuration of c is determined by the states of the neighborhood of that cell in configuration c#, where c# differs from c only in that we update the states of all cells processed before the given one. Case ii) results when some cells are first processed in parallel and then the remaining cells are processed sequentially. Case iii) results when some cells are first processed sequentially and then the remaining cells are processed in parallel.

Suppose $\sigma:S^q \to S$. We define $\underline{par(\sigma)} \in G_p$ to be the parallel global array transformation determined by σ, $\underline{seq_\tau(\sigma)} \in G_s$ to be the sequential global array transformation determined by σ and the surjective trajectory τ, $\underline{par\text{-}seq_\tau(\sigma)} \in G_{p,s}$ to be the parallel-sequential global array transformation determined by σ and the non-surjective trajectory τ, and $\underline{seq\text{-}par_\tau(\sigma)} \in G_{s,p}$ to be the sequential-parallel global array transformation determined by σ and the non-surjective trajectory τ.

We now define various concepts which will prove to be useful in the balance of this paper.

DEFINITION 2: For $\sigma:S^q \to S$, if σ is independent of its j-th argument, for $1 \leq j \leq q$, we call cell $\underline{\zeta}+\underline{\gamma}_j$ an independent neighbor of $\underline{\zeta}$ with respect to σ, for each $\underline{\zeta} \in Z^n$.

DEFINITION 3: For $\underline{\zeta} \in Z^n$ and τ a trajectory, we define the preprocessed set of $\underline{\zeta}$ with respect to τ for pure sequential, parallel-sequential and sequential-parallel processing,

a) Let τ be a surjective trajectory. Then the set $\{\tau(0),\ldots,\tau(\tau^{-1}(\underline{\zeta})-1)\} \cap Ne(\underline{\zeta})$ is called the preprocessed set of $\underline{\zeta}$ with respect to τ for pure sequential processing

b) Let τ be a non-surjective trajectory,

i) Suppose $\underline{\zeta} \in \text{range}(\tau)$. Then, the set $(\{\tau(0),\ldots,\tau((\tau|_{\text{range}(\tau)})^{-1}(\underline{\zeta})-1)\} \cup (Z^n - \text{range}(\tau))) \cap Ne(\underline{\zeta})$ is called the preprocessed set of $\underline{\zeta}$ with respect to τ for parallel sequential processing.

ii) Suppose $\underline{\zeta} \notin \text{range}(\tau)$. Then the preprocessed set of $\underline{\zeta}$ with respect to τ for parallel sequential processing is $\emptyset$.

c) Let τ be a non-surjective trajectory.

i) Suppose $\underline{\zeta} \in \text{range}(\tau)$. Then the set $\{\tau(0),\ldots,\tau((\tau|_{\text{range}(\tau)})^{-1}(\underline{\zeta})-1)\} \cap Ne(\underline{\zeta})$ is called the preprocessed set of $\underline{\zeta}$ with respect to τ for sequential-parallel processing.

ii) Suppose $\underline{\zeta} \notin \text{range}(\tau)$. Then the preprocessed set of $\underline{\zeta}$ with respect to τ for sequential-parallel processing is $\text{range}(\tau) \cap Ne(\underline{\zeta})$.

Briefly, the preprocessed set of $\underline{\zeta}$ with respect to τ in the various methods of processing is just the collection of neighbors of $\underline{\zeta}$ which were processed before $\underline{\zeta}$.

DEFINITION 4: We call a local transformation $\sigma:S^q \to S$ surjective of degree k/q, for $0 < k \leq q$, if, by varying the values of any k arguments of σ, we can produce as output every element of S, regardless of the values of the other q-k arguments. That is, letting $1 \leq i_1 < \ldots < i_k \leq q$, the function $\sigma\#:S^k \to S$ defined by $\sigma\#(x_{i_1},\ldots,x_{i_k}) = \sigma(y_1,\ldots,y_q)$, where, for $1 \leq j \leq q$, $y_j = x_j$ if $j \in \{i_1,\ldots,i_k\}$, while $y_j = s_j \in S$ if $j \notin \{i_1,\ldots,i_k\}$, is surjective.

DEFINITION 5: Cell $\underline{\zeta}$ is said to be a related neighbor of cell $\underline{\xi}$ if there exists a chain of cells $\underline{\delta}_1,\ldots,\underline{\delta}_m$, for $m \geq 2$, such that $\underline{\delta}_1 = \underline{\zeta}$, $\underline{\delta}_m = \underline{\xi}$, and, for $1 \leq i \leq m-1$, $\underline{\delta}_i \in Ne(\underline{\delta}_{i+1})$.

DEFINITION 6: Suppose $\rho \in G_p(G_s)(G_{p,s})(G_{s,p})$. The seed set of ρ, $SS(\rho)$, is that set of local transformations σ such that $\rho = par(\sigma)$ ($\rho = seq_\tau(\sigma)$ for some τ) ($\rho = par\text{-}seq_\tau(\sigma)$ for some τ) ($\rho = seq\text{-}par_\tau(\sigma)$ for some τ)

Various Notions of Simulation

In this section we examine numerous notions of the simulation of one tessellation automaton by another. One general definition of simulation which we will use is that of A.R. Smith [8]:

DEFINITION 7: Let $TA = \langle S, Z^n, NI, GT\rangle$ and $TA^* = \langle S^*, Z^n, NI^*, GT^*\rangle$ be two n-dimensional stratified mixed mode tessellation automata. For $t,r \geq 1$, we say that TA* simulates T in t/r times real time, if there are effectively computable injective mappings $\Delta:CON \to CON^*$ and $\Gamma:GT^r \to GT^{*t}$, such that, for any $c \in CON$ and $\langle \rho_1,\ldots,\rho_r\rangle \in GT^r$, we have,

$\Delta(\rho_r(\ldots\rho_1(c))\ldots)) = \rho_t^*(\ldots(\rho_1^*(\Delta(c)))\ldots)$, where $\langle\rho_1^*,\ldots,\rho_t^*\rangle = \Gamma(\langle\rho_1,\ldots,\rho_r\rangle)$. If $t = r = 1$, we say that TA* simulates TA in real time.

The type of simulation we examine first is called strict simulation,

DEFINITION 8: Let $TA = \langle S, Z^n, NI, GT\rangle$ and $TA^* = \langle S^*, Z^n, NI^*, GT^*\rangle$ be two n-dimensional stratified mixed mode tessellation automata which are such that $S^* = S$ and $NI^* = NI$. We say that TA* strictly simulates TA if, in Definition 6, $t = r = 1$, Δ is the identity map, and, for $\rho \in GT$, $SS(\rho) \cap SS(\Gamma(\rho)) \neq \emptyset$.

In exploring this notion of strict simulation, we are really just trying to determine when global transformations of the form $par(\sigma)$, $seq_{\tau_1}(\sigma)$, $par\text{-}seq_{\tau_2}(\sigma)$ and $seq\text{-}par_{\tau_3}(\sigma)$ are equal for various σ, τ_1, τ_2, τ_3. We thus concentrate on the latter formulation.

Our first result, being fairly obvious, is presented without proof,

THEOREM 1: For $\sigma: S^q \to S$ and τ_1, τ_2, τ_3 trajectories of the appropriate type, a sufficient condition for $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ to be pairwise equal is that for each cell $\underline{\zeta} \in Z^n$, the preprocessed set of $\underline{\zeta}$ with respect to τ_1 (τ_2) (τ_3) for sequential (parallel-sequential) (sequential-parallel) processing consist entirely of independent neighbors of $\underline{\zeta}$ with respect to σ.

The converse does not hold as the following example demonstrates,

Let $n = 1$, $NI = \langle -1, 0\rangle$, $S = \{1,2,3\}$, and $\sigma(1,1) = \sigma(2,1) = \sigma(3,1) = \sigma(3,2) = 1$, $\sigma(1,3) = \sigma(2,3) = \sigma(3,3) = 3$, and $\sigma(1,2) = \sigma(2,2) = 2$.

It is easily verified that $par(\sigma) = seq_{\tau_1}(\sigma) = par\text{-}seq_{\tau_2}(\sigma) = seq\text{-}par_{\tau_3}(\sigma)$ for all appropriate trajectories, while σ is neither independent of its first nor second argument.

We do have the following, though,

THEOREM 2: For $\sigma: S^q \to S$ and τ_1, τ_2, τ_3 trajectories of the appropriate type, suppose that $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ are pairwise equal. Letting SET_i be the preprocessed set of $\underline{\zeta}$ with respect to τ_i, for some $1 \leq i \leq 3$, suppose that for each $\underline{\xi} \in SET_i$, σ is surjective of degree $\alpha_{\underline{\xi}}/q$, where,

$$\alpha_{\underline{\xi}} = q - |Ne(\underline{\xi}) \cap Ne(\underline{\zeta})| - |Ne(\underline{\xi}) \cap \bigcup_{\underline{\delta} \in SET_i - \{\underline{\xi}\}} Ne(\underline{\delta})|.$$

Then, each element of SET_i is an independent neighbor of $\underline{\zeta}$.

PROOF OF THEOREM 2: It is easily seen that if $\alpha_{\underline{\xi}} > 0$, then there are at least $\alpha_{\underline{\xi}}$ cells in the neighborhood of $\underline{\xi}$ - not including $\underline{\xi}$ - which are neither in the neighborhood of $\underline{\zeta}$ nor in the neighborhood of any other cell in SET_i. Since σ is surjective of degree $\alpha_{\underline{\xi}}/q$, by varying the states in these $\alpha_{\underline{\xi}}$ cells, we can force cell $\underline{\xi}$ to be in any state in S before we process cell $\underline{\zeta}$. Consider any initial configuration c. Suppose the next state of cell $\underline{\zeta}$ is $s_{\underline{\zeta}}$ if c is processed in a parallel mode. Thus, the next state of cell $\underline{\zeta}$ is $s_{\underline{\zeta}}$ if c is processed in any of the other three modes, regardless of the states of the cells in SET_i. Since each cell in SET_i can be put in any state, our result follows.

QED

COROLLARY 1: Let $\underline{\zeta} \in Z^n$ and $|S|, |Ne(\underline{\zeta})| > 1$. Suppose that for each $\underline{\xi} \in Ne(\underline{\zeta}) - \{\underline{\zeta}\}$, the local transformation σ is surjective of degree $\alpha_{\underline{\xi}}/q > 0$ where $\alpha_{\underline{\xi}} = q - |Ne(\underline{\xi}) \cap Ne(\underline{\zeta})|$. Then, there exist appropriate trajectories τ_1, τ_2, τ_3 such that $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ are not pairwise equal.

PROOF OF COROLLARY 1: Suppose that for all appropriate trajectories τ_1, τ_2, τ_3, we have that $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ are pairwise equal. Choose trajectories τ_1^*, τ_2^*, and τ_3^* such that, for each $1 \leq i \leq q$, there is a cell $\underline{\zeta}_i$ such that $\underline{\zeta}_i + \underline{\gamma}_i$ is the only element in the preprocessed set of $\underline{\zeta}_i$ with respect to τ_1^*, τ_2^*, and τ_3^*. By Theorem 2, $\underline{\zeta}_i + \underline{\gamma}_i$, for $1 \leq i \leq q$, is an independent neighbor of $\underline{\zeta}_i$ with respect to σ. Thus, either σ is a constant function or the next state of any cell depends just on its previous state. Thus, σ is not surjective of degree $\alpha_{\underline{\xi}}/q > 0$.

QED

From Theorem 1, we see that if trajectories τ_1, τ_2, τ_3 can be found such that for each application of the local transformation σ to a cell $\underline{\xi}$, each cell in $Ne(\underline{\xi})$ is not in the preprocessed set of $\underline{\xi}$ with respect to τ_1, τ_2, τ_3, or, if it is, is either an independent neighbor of $\underline{\xi}$ with respect to σ or is in a state which never changes, then $par(\sigma) = seq_{\tau_1}(\sigma) = par\text{-}seq_{\tau_2}(\sigma) = seq\text{-}par_{\tau_3}(\sigma)$. We now look into the question of finding such trajectories. For this question, we restrict our attention to the finite configurations and a sub-set of them called the fixed configurations.

Our definition of strict simulation is likewise restricted to strict simulation over finite or fixed configurations by restricting the domain of Δ to these sub-sets.

DEFINITION 9: A state $s \in S$ is called quiescent with respect to $\sigma:S^q \to S$ if $\sigma(\underbrace{s,\ldots,s}_{q}) = s$. A configuration c is called finite with respect to $\sigma:S^q \to S$ if only a finite number of cells of c are non-quiescent with respect to σ.

DEFINITION 10: A state $s \in S$ is called null with respect to $\sigma:S^q \to S$ if $c(\underline{\zeta}) = s \leftrightarrow par(\sigma)(c)(\underline{\zeta}) = s$. A configuration all of whose cells are in a null state is of size 0. A configuration c is of size $m \geq 1$ if, when the absolute value of any coordinate of a cell $\underline{\zeta}$ is larger than m, then $c(\underline{\zeta})$ is null. A configuration is called fixed of degree m if m is the least integer such that c is of size m. The window set of a fixed configuration of degree $m > 0$ is the set consisting of those ceils which are such that the absolute value of each of their coordinates is less than or equal to m. We denote it by 'W_m'. The window set of a configuration all of whose cells are in a null state is $\emptyset$.

Of course, all configurations which are fixed of degree m are finite.

LEMMA 1: Let $\underline{\zeta}_1 R_n \underline{\zeta}_2 \leftrightarrow \underline{\zeta}_1, \underline{\zeta}_2 \in Z^n$ and $\underline{\zeta}_1$ is a related neighbor of $\underline{\zeta}_2$ and let $E_n = \{\langle\underline{\zeta},\underline{\zeta}\rangle \mid \underline{\zeta} \in Z^n\}$. Then $R_n \cup E_n$ is a transitive, reflexive relation.

THEOREM 3: Let $\sigma:S^q \to S$ and $NI = \langle\underline{\gamma}_1,\ldots,\underline{\gamma}_q\rangle$, for $\underline{\gamma}_1,\ldots,\underline{\gamma}_q \in Z^n$. A sufficient condition for $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ to be pair-wise equal over fixed configurations of degree $m > 0$ for some trajectories τ_1,τ_2,τ_3 is that no two different cells of Z^n are related neighbors of each other.

PROOF OF THEOREM 3: We claim that $R_n \cup E_n$ is a partial order over Z^n. From Lemma 1, $R_n \cup E_n$ is transitive and reflexive. Suppose $\underline{\zeta}_1 (R_n \cup E_n) \underline{\zeta}_2$ and $\underline{\zeta}_2 (R_n \cup E_n) \underline{\zeta}_1$. By our hypotheses, it is impossible to have $\underline{\zeta}_1 R_n \underline{\zeta}_2$ and $\underline{\zeta}_2 R_n \underline{\zeta}_1$. Thus, we must have that $\underline{\zeta}_1 = \underline{\zeta}_2$. Thus, we have our result. This, in turn implies that $(R_n \cup E_n) \cap (W_m \times W_m)$ is a partial order over W_m. We then embed $(R_n \cup E_n) \cap (W_m \times W_m)$ in a total order θ. Thus, for all $\underline{\zeta}_1, \underline{\zeta}_2 \in W_m$, $\underline{\zeta}_1 R_n \underline{\zeta}_2 \to \underline{\zeta}_1 \theta \underline{\zeta}_2$.

Let $\langle\underline{\zeta}_{i_1},\ldots,\underline{\zeta}_{i_{m^n}}\rangle$ be a listing of W_m such that for $1 \leq u < v \leq m^n$, $\underline{\zeta}_{i_u} \theta \underline{\zeta}_{i_v}$. Note that for $1 \leq u < v \leq m^n$, $\underline{\zeta}_{i_v} \notin Ne(\underline{\zeta}_{i_u})$. For, suppose $\underline{\zeta}_{i_v} \in Ne(\underline{\zeta}_{i_u})$. Thus, $\underline{\zeta}_{i_v}$ is a related neighbor of $\underline{\zeta}_{i_u}$ and hence $\underline{\zeta}_{i_v} R_n \underline{\zeta}_{i_u}$, which implies that $\underline{\zeta}_{i_v} \theta \underline{\zeta}_{i_u}$, which, in turn, implies that $j_2 \leq j_1$, which is a contradiction.

Thus, for pure sequential processing, we can choose any trajectory τ_1 such that, if $k_1 < k_2$, $\tau_1(k_1) = \underline{\zeta}_{i_{j_1}}$, $\tau_1(k_2) = \underline{\zeta}_{i_{j_2}}$, for $1 \leq j_1,j_2 \leq m^n$, then $j_2 < j_1$.

For parallel-sequential processing, we can choose any trajectory τ_2 which is such that either,

1) the range of τ_2 doesn't include any element $\underline{\zeta}_{i_j}$ for $1 \leq j \leq m^n$; or

2) the range of τ_2 includes each element of $\{\underline{\zeta}_{i_1},\underline{\zeta}_{i_2},\ldots,\underline{\zeta}_{i_j}\}$, for some $1 \leq j \leq m^n$, is disjoint from $\{\underline{\zeta}_{i_{j+1}},\ldots,\underline{\zeta}_{i_{m^n}}\}$, and is such that if $k_1 < k_2$, $\tau_2(k_1) = \underline{\zeta}_{i_{j_1}}$, $\tau_2(k_2) = \underline{\zeta}_{i_{j_2}}$, for $1 \leq j_1,j_2 \leq j$, then $j_2 < j_1$.

For sequential-parallel processing we can choose any trajectory τ_3 which is such that either,

1) the range of τ_3 doesn't include any element $\underline{\zeta}_{i_j}$ for $1 \leq j \leq m^n$; or

2) the range of τ_3 includes each element of $\{\underline{\zeta}_{i_j},\ldots,\underline{\zeta}_{i_{m^n}}\}$ for some $1 \leq j \leq m^n$, is disjoint from $\{\underline{\zeta}_{i_1},\ldots,\underline{\zeta}_{i_{j-1}}\}$ and is such that if $k_1 < k_2$, $\tau_3(k_1) = \underline{\zeta}_{i_{j_1}}$, $\tau_3(k_2) = \underline{\zeta}_{i_{j_2}}$, for $j \leq j_1,j_2 \leq m^n$, then $j_2 < j_1$.

QED

We now give a necessary and sufficient condition for two different cells of Z^n to be related neighbors of each other; but first we show,

LEMMA 2: Suppose NI = $\langle \underline{\gamma}_1,\ldots,\underline{\gamma}_q \rangle$ for $\underline{\gamma}_1,\ldots,\underline{\gamma}_q$ members of Z^n. Then cell $\underline{\zeta}$ is a related neighbor of cell $\underline{\theta}$, for $\underline{\zeta} \neq \underline{\theta}$, iff $\underline{\zeta} = \delta_{j_1}\underline{\gamma}_{j_1} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p} + \underline{\theta}$ for some $1 \leq p \leq q$, where $\delta_{j_k} > 0$ and $\underline{\gamma}_{j_k} \neq \underline{0}$ for $1 \leq k \leq p$.

PROOF OF LEMMA 2: Suppose $\underline{\zeta}$ is a related neighbor of $\underline{\theta}$. Thus, there is a chain of cells $\underline{\xi}_1,\ldots,\underline{\xi}_s$, for $s \geq 2$, such that $\underline{\xi}_1 = \underline{\zeta}$, $\underline{\xi}_s = \underline{\theta}$, and, for $1 \leq i \leq s-1$, $\underline{\xi}_i \in Ne(\underline{\xi}_{i+1})$. Thus, $\underline{\xi}_1 = \underline{\xi}_2 + \underline{\gamma}_{k_2}$, $\underline{\xi}_2 = \underline{\xi}_3 + \underline{\gamma}_{k_3},\ldots, \underline{\xi}_{s-1} = \underline{\xi}_s + \underline{\gamma}_{k_s}$, for $1 \leq k_2,\ldots, k_s \leq q$. We then get that $\underline{\xi}_1 = \underline{\gamma}_{k_2} + \underline{\gamma}_{k_3} + \ldots + \underline{\gamma}_{k_s} + \underline{\xi}_s = \delta_{j_1}\underline{\gamma}_{j_1} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p} + \underline{\xi}_s$, where $1 \leq p \leq q$, $\delta_{j_k} > 0$ and $\underline{\gamma}_{j_k} \neq \underline{0}$ for $1 \leq k \leq p$. Our result follows trivially since $\underline{\xi}_1 = \underline{\zeta}$ and $\underline{\xi}_s = \underline{\theta}$.

Suppose $\underline{\zeta} = \delta_{j_1}\underline{\gamma}_{j_1} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p} + \underline{\theta}$ for some $1 \leq p \leq q$, where $\delta_{j_k} > 0$ and $\underline{\gamma}_{j_k} \neq \underline{0}$ for $1 \leq k \leq p$. We thus get that $\underline{\theta} + \underline{\gamma}_{j_p} \in Ne(\underline{\theta})$, $\underline{\theta} + 2\underline{\gamma}_{j_p} \in Ne(\underline{\theta} + \underline{\gamma}_{j_p}),\ldots, \underline{\theta} + \delta_{j_p}\underline{\gamma}_{j_p} \in Ne(\underline{\theta} + (\delta_{j_p}-1)\underline{\gamma}_{j_p})$, $\underline{\theta} + \underline{\gamma}_{j_{p-1}} + \delta_{j_p}\underline{\gamma}_{j_p} \in Ne(\underline{\theta} + \delta_{j_p}\underline{\gamma}_{j_p})$, $\ldots, \underline{\theta} + \delta_{j_{p-1}}\underline{\gamma}_{j_{p-1}} + \delta_{j_p}\underline{\gamma}_{j_p} \in Ne(\underline{\theta} + (\delta_{j_{p-1}}-1)\cdot \underline{\gamma}_{j_{p-1}} + \delta_{j_p}\underline{\gamma}_{j_p}),\ldots, \underline{\zeta} \in Ne((\delta_{j_1}-1)\underline{\gamma}_{j_1} + \delta_{j_2}\underline{\gamma}_{j_2} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p})$. Hence, $\underline{\zeta}$ is a related neighbor of $\underline{\theta}$.

QED

We now present,

THEOREM 4: Suppose NI = $\langle \underline{\gamma}_1,\ldots,\underline{\gamma}_q \rangle$ for $\underline{\gamma}_1,\ldots, \underline{\gamma}_q \in Z^n$. Then there exists 2 different cells, $\underline{\zeta}$ and $\underline{\theta}$, which are related neighbors of each other iff $\alpha_{k_1}\underline{\gamma}_{k_1} + \ldots + \alpha_{k_s}\underline{\gamma}_{k_s} = \underline{0}$ for some $2 \leq s \leq q$, where $\alpha_{k_i} > 0$ and $\underline{\gamma}_{k_i} \neq \underline{0}$ for $1 \leq i \leq s$.

PROOF OF THEOREM 4: Suppose 2 different cells, $\underline{\zeta}$ and $\underline{\theta}$, are related neighbors of each other. By Lemma 2, $\underline{\zeta} = \delta_{j_1}\underline{\gamma}_{j_1} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p} + \underline{\theta}$ and $\underline{\theta} = \beta_{r_1}\underline{\gamma}_{r_1} + \ldots + \beta_{r_u}\underline{\gamma}_{r_u} + \underline{\zeta}$ for $1 \leq p,u \leq q$, where $\alpha_{j_i} > 0$ and $\underline{\gamma}_{j_i} \neq \underline{0}$ for $1 \leq i \leq p$, and $\beta_{r_i} > 0$ and $\underline{\gamma}_{r_i} \neq \underline{0}$ for $1 \leq i \leq u$. Thus, $\underline{\zeta} = \delta_{j_1}\underline{\gamma}_{j_1} + \ldots + \delta_{j_p}\underline{\gamma}_{j_p} + \beta_{r_1}\underline{\gamma}_{r_1} + \ldots + \beta_{r_u}\underline{\gamma}_{r_u} + \underline{\zeta}$, and our result follows.

Suppose $\alpha_{k_1}\underline{\gamma}_{k_1} + \ldots + \alpha_{k_s}\underline{\gamma}_{k_s} = \underline{0}$ for some $2 \leq s \leq q$, where $\alpha_{k_i} > 0$ and $\underline{\gamma}_{k_i} \neq \underline{0}$ for $1 \leq i \leq s$. Thus, letting $\underline{\psi} = \alpha_{k_2}\underline{\gamma}_{k_2} + \ldots + \alpha_{k_s}\underline{\gamma}_{k_s} \neq \underline{0}$, we see that $\underline{0} = \alpha_{k_1}\underline{\gamma}_{k_1} + \underline{\psi}$, and $\underline{\psi} = \alpha_{k_2}\underline{\gamma}_{k_2} + \ldots + \alpha_{k_s}\underline{\gamma}_{k_s} + \underline{0}$. Thus, by Lemma 2, $\underline{0}$ and $\underline{\psi}$ are related neighbors of each other.

Thus, we have,

COROLLARY 2: Let $\sigma:S^q \to S$ and NI = $\langle \underline{\gamma}_1,\ldots,\underline{\gamma}_q \rangle$ for $\underline{\gamma}_1,\ldots,\underline{\gamma}_q \in Z^n$. Then, a sufficient condition for $\{par(\sigma), seq_{\tau_1}(\sigma), par\text{-}seq_{\tau_2}(\sigma), seq\text{-}par_{\tau_3}(\sigma)\}$ to be pairwise equal over fixed configurations of degree $m > 0$, for some trajectories τ_1,τ_2,τ_3, is that,

$$|\alpha_{k_1}|\underline{\gamma}_{k_1} + \ldots + |\alpha_{k_s}|\underline{\gamma}_{k_s} = \underline{0}$$

for $2 \leq s \leq q$, where $\underline{\gamma}_{k_i} \neq \underline{0}$ for $1 \leq i \leq s$, implies that $\alpha_{k_1} = \ldots = \alpha_{k_s} = 0$.

We can generalize this in the following fashion,

DEFINITION 11: Let σ be a local transformation. We say that pure parallel processing using σ is weakly equivalent to pure sequential processing using σ, over a set of configurations C,

$$par(\sigma) \sim_C seq(\sigma),$$

if, for every $c \in C$, there exists an appropriate trajectory τ, such that $par(\sigma)(c) = seq_\tau(\sigma)(c)$. We have similar definitions for the other methods of processing.

We then have,

COROLLARY 3: Let $\sigma:S^q \to S$ and NI = $\langle \underline{\gamma}_1,\ldots,\underline{\gamma}_q \rangle$, for $\underline{\gamma}_1,\ldots,\underline{\gamma}_q \in Z^n$. Then, a sufficient condition for $\{par(\sigma), seq(\sigma), par\text{-}seq(\sigma), seq\text{-}par(\sigma)\}$ to be pairwise weakly equivalent with respect to the set of finite configurations is that,

$$|\alpha_{k_1}|\underline{\gamma}_{k_1} + \ldots + |\alpha_{k_s}|\underline{\gamma}_{k_s} = \underline{0}$$

for $2 \leq s \leq q$, where $\underline{\gamma}_{k_i} \neq \underline{0}$ for $1 \leq i \leq s$,

implies that $\alpha_{k_1} = \ldots = \alpha_{k_s} = 0$.

PROOF OF COROLLARY 3: This follows directly from Corollary 2 and Theorem 3. QED

Let us now present an example of a neighborhood index and local transformation σ such that $par(\sigma) \neq seq_\tau(\sigma)$ for all trajectories τ. We let NI = <-1,0,1> - each cell is in Z - S = {0,1} and $\sigma(1,0,0) = \sigma(0,0,1) = 1$, $\sigma(0,0,0) = \sigma(0,1,0) = \sigma(0,1,1) = \sigma(1,0,1) = \sigma(1,1,0) = \sigma(1,1,1) = 0$. Consider the configuration $c = \overline{0}10010\overline{0}$; that is, there is an $i_0 \in Z$ such that $c(i_0) = c(i_0+3) = 1$ and $c(i) = 0$ otherwise. It is easily verified that $par(\sigma) \neq seq_\tau(\sigma)$ for any trajectory τ. It is also easily verified that this example does not meet the sufficient conditions mentioned in the previous theorems. Here we see clearly what the concept of related neighbors portends; there are cells which are related neighbors of each other and which are such that whichever one is processed first destroys the possibility that the other will be in a state so that the resultant configuration is the same as if the original configuration were processed in parallel. We call this the mutually destructive condition. If this condition exists in a given situation, we have that $par(\sigma) \neq seq_\tau(\sigma)$ for any trajectory τ. The preceding theorems gave sufficient conditions for this condition not to exist.

We now present some theorems regarding the general notion of simulation. But first, we must define the following entities,

DEFINITION 12: A stratified mixed mode tessellation automaton is called pure parallel if and only if the union of its sets of sequential, parallel-sequential, and sequential-parallel global array transformations is empty.

DEFINITION 13: A stratified mixed mode tessellation automaton is called pure sequential if and only if the union of its sets of parallel, parallel-sequential, and sequential-parallel global array transformations is empty.

We now have the following,

THEOREM 5: Let $TA = \langle S, Z^n, \langle \underline{\gamma}_1, \ldots, \underline{\gamma}_q \rangle, GT \rangle$. Then, there is a pure sequential stratified mixed mode tessellation automaton, STA, which simulates TA in 2 times real time.

PROOF OF THEOREM 5:

Case 1: Suppose $\underline{\gamma}_j \neq \underline{0}$ for all $1 \leq j \leq q$

Let $GT_p = \{\rho_1, \ldots, \rho_r\}$, $GT_s = \{\rho_{r+1}, \ldots, \rho_{r+s}\}$, $GT_{p,s} = \{\rho_{r+s+1}, \ldots, \rho_{r+s+t}\}$, and $GT_{s,p} = \{\rho_{r+s+t+1}, \ldots, \rho_{r+s+t+u}\}$, $\theta_1 = \{a_1, a_2, a_3, a_4\}$ and $\theta_2 = \{b_1, b_2\}$, where $\theta_1 \cap S = \theta_2 \cap S = \emptyset$.

We then let $STA = \langle S \times S \times (\theta_1 \times \theta_2)^{r+s+t+u}, Z^n, \langle \underline{\gamma}_1, \ldots, \underline{\gamma}_q, \underline{0} \rangle, GT^* \rangle$, where GT* will be defined presently. Before we specify what Δ and Γ are, let us do the following,

We first define a map $M: GT_p \cup GT_s \cup GT_{p,s} \cup GT_{s,p} \to \{\tau \mid \tau: \omega \to Z^n, \tau \text{ is 1-1 and onto}\}$ as follows (this will be the set of trajectories used in STA):

a) For $\rho \in GT_p$, let $M(\rho)$ be arbitrary

b) For $\rho \in GT_s$, ρ is determined by some trajectory τ^*. Let $M(\rho) = \tau^*$

c) For $\rho \in GT_{p,s}$, ρ is determined by some trajectory τ^{**}. Let $M(\rho)$ be any trajectory such that,

i) Suppose $\underline{\zeta}_1, \underline{\zeta}_2$ are two different cells of Z^n in the range of τ^{**} and that $\tau^{**-1}(\underline{\zeta}_1) < \tau^{**-1}(\underline{\zeta}_2)$. Then,

$$(M(\rho))^{-1}(\underline{\zeta}_1) < (M(\rho))^{-1}(\underline{\zeta}_2).$$

(The order of sequential processing is preserved.)

ii) Suppose $\underline{\zeta} \in \text{range}(\tau^{**})$, $\underline{\xi} \notin \text{range}(\tau^{**})$ and $\underline{\xi} \in Ne(\underline{\zeta})$ or $\underline{\zeta} \in Ne(\underline{\xi})$. Then,

$$(M(\rho))^{-1}(\underline{\xi}) < (M(\rho))^{-1}(\underline{\zeta}).$$

(A cell processed sequentially in TA may be processed any time in STA as long as all cells in its neighborhood or which have it as a neighbor, which were processed in parallel in TA, are processed first.

d) For $\rho \in GT_{s,p}$, ρ is determined by some trajectory τ^{***}. Let $M(\rho)$ be any trajectory such that,

i) Suppose $\underline{\zeta}_1, \underline{\zeta}_2$ are two different cells of Z^n in the range of τ^{***} and that $\tau^{***-1}(\underline{\zeta}_1) < \tau^{***-1}(\underline{\zeta}_2)$. Then,

$$(M(\rho))^{-1}(\underline{\zeta}_1) < (M(\rho))^{-1}(\underline{\zeta}_2).$$

ii) Suppose $\underline{\zeta} \notin \text{range}(\tau^{***})$, $\underline{\xi} \in \text{range}(\tau^{***})$ and $\underline{\xi} \in Ne(\underline{\zeta})$ or $\underline{\zeta} \in Ne(\underline{\xi})$. Then,

$$(M(\rho))^{-1}(\underline{\xi}) < (M(\rho))^{-1}(\underline{\zeta}).$$

(A cell processed in parallel in TA may be processed any time in STA as long as all cells in its neighborhood or which have it as a neighbor, which were processed sequentially in TA, are processed first.

For $1 \leq j \leq r+s+t+u$, let $\tau_j = M(\rho_j)$.

We now specify the map $\Delta: CON_{TA} \to CON_{STA}$. Let $c \in CON_{TA}$. Then, $\Delta(c) = c^*$, where, for $\underline{\zeta} \in Z^n$, if $c(\underline{\zeta}) = s$, then $c^*(\underline{\zeta}) = \langle s, \hat{s}, \langle\langle a_{i_1}, b_{i_1} \rangle, \ldots, \langle a_{i_{r+s+t+u}}, b_{i_{r+s+t+u}} \rangle\rangle\rangle$, where,

a) $\hat{s}$ is a fixed element of S

b) For $1 \leq j \leq r$, we have $i_j = 1$, for $r+1 \leq j \leq r+s$, we have $i_j = 2$, for $r+s+1 \leq j \leq r+s+t$, we have $i_j = 3$, and for $r+s+t+1 \leq j \leq r+s+t+u$, we have $i_j = 4$. (This indicates the total range of processing: parallel, sequential, parallel-sequential and sequential-parallel.)

c) For $1 \leq j \leq r+s+t+u$, b_{i_j} indicates whether or not cell $\underline{\zeta}$ is in the range of the trajectory corresponding to ρ_j. That is, for $1 \leq j \leq r$, we let $i_j = 1$, while for $r+1 \leq j \leq r+s+t+u$, $i_j = 1$ if and only if cell $\underline{\zeta}$ is not in the trajectory which determines ρ_j.

As a notational convenience, for

$\alpha = \langle\langle a_{i_1}, b_{i_1}\rangle, \ldots, \langle a_{i_{r+s+t+u}}, b_{i_{r+s+t+u}}\rangle\rangle \in (\theta_1 \times \theta_2)^{r+s+t+u}$, let $\alpha[k] = \langle a_{i_k}, b_{i_k}\rangle$, for $1 \leq k \leq r+s+t+u$, and $\alpha[k][1] = a_{i_k}$ and $\alpha[k][2] = b_{i_k}$.

Now, the map $\Gamma: GT \to GT^{*2}$ is defined by $\Gamma(\rho) = \langle \rho_1^*, \rho_2^* \rangle$, where we now define ρ_1^* and ρ_2^*.

Suppose $\rho = \rho_i$, for $1 \leq i \leq r+s+t+u$. Now, ρ is defined via some local transformation σ_ρ. We have that ρ_1^* is defined via the trajectory $\tau_i = M(\rho)$ and local transformation $\sigma^*_{\rho_1}$, where,

$$\sigma^*_{\rho_1}(\langle s_1, \hat{s}_1, \alpha_1\rangle, \ldots, \langle s_{q+1}, \hat{s}_{q+1}, \alpha_{q+1}\rangle) =$$

$$\begin{cases} \langle s_{q+1}, \sigma_\rho(s_1, \ldots, s_{q+1}), \alpha_{q+1}\rangle, \text{ if either} \\ \quad 1 \leq i \leq r, \text{ or } r+s+1 \leq i \leq r+s+t \text{ and} \\ \quad \alpha_{q+1}[i][2] = b_1, \text{ or } r+s+t+1 \leq i \leq \\ \quad r+s+t+u \text{ and } \alpha_{q+1}[i][2] = b_1 \\ \langle \sigma_\rho(s_1, \ldots, s_{q+1}), \hat{s}_{q+1}, \alpha_{q+1}\rangle, \text{ if either} \\ \quad r+1 \leq i \leq r+s, \text{ or } r+s+t+1 \leq i \leq \\ \quad r+s+t+u \text{ and } \alpha_{q+1}[i][2] = b_2 \\ \langle \sigma_\rho(s_1^*, \ldots, s_{q+1}^*), \hat{s}_{q+1}, \alpha_{q+1}\rangle, \text{ if} \\ \quad r+s+1 \leq i \leq r+s+t \text{ and } \alpha_{q+1}[i][2] = b_2, \\ \quad \text{where, for } 1 \leq k \leq q+1, \\ \qquad s_k^* = \begin{cases} s_k \text{ if } \alpha_k[i][2] = b_2 \\ \hat{s}_k \text{ if } \alpha_k[i][2] = b_1 \end{cases} \end{cases}$$

We also have that ρ_2^* is defined via local transformation $\sigma^*_{\rho_2}$ and an arbitrary trajectory, where,

$$\sigma^*_{\rho_2}(\langle s_1, \hat{s}_1, \alpha_1\rangle, \ldots, \langle s_{q+1}, \hat{s}_{q+1}, \alpha_{q+1}\rangle) =$$

$$\begin{cases} \langle s_{q+1}, \hat{s}_{q+1}, \alpha_{q+1}\rangle \text{ if } \alpha_{q+1}[i][2] = b_2 \\ \langle \hat{s}_{q+1}, s_{q+1}, \alpha_{q+1}\rangle \text{ if } \alpha_{q+1}[i][2] = b_1 \end{cases}$$

Case 2 $\gamma_{j'} = \underline{0}$ for some $1 \leq j' \leq q$

This case is similar to the above.

It is straightforward to show that STA simulates TA in 2 times real time.

QED

Before we present our last theorem, we define the following concept,

DEFINITION 14: Let $TA = \langle S, Z^n, \langle \underline{\gamma}_1, \ldots, \underline{\gamma}_q\rangle, GT\rangle$. Suppose $\underline{\zeta} \in Z^n$ and $\rho \in GT_s \cup GT_{p,s} \cup GT_{s,p}$. We now define what we call the type of cell $\underline{\zeta}$ with respect to ρ, denoted by,

$$type^{TA}_{\underline{\zeta},\rho},$$

as follows,

a) Suppose $\rho \in GT_s$. Let τ_ρ be the trajectory which determines ρ. Set $S_0 = \{*\}$, and, for $i \geq 0$, let $S_{i+1} = S_i \cup S_i^q$. For $0 \leq j \leq \tau_\rho^{-1}(\underline{\zeta})$, we define $\varepsilon_j^{(\underline{\zeta})}: Z^n \to S_j$. (Thus, $\varepsilon_0^{(\underline{\zeta})}$ is the constant function which maps Z^n into $\{*\}$.) For $0 \leq k \leq \tau_\rho^{-1}(\underline{\zeta})-1$ and $\underline{\xi} \in Z^n$, we define,

$$\varepsilon_{k+1}^{(\underline{\zeta})}(\underline{\xi}) = \begin{cases} \langle \varepsilon_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_1), \ldots, \varepsilon_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_q)\rangle \text{ if} \\ \qquad \underline{\xi} = \tau_\rho(k) \\ \varepsilon_k^{(\underline{\zeta})}(\underline{\xi}) \text{ otherwise} \end{cases}$$

Then, $type^{TA}_{\underline{\zeta},\rho} = \varepsilon^{(\underline{\zeta})}_{\tau_\rho^{-1}(\underline{\zeta})}(\underline{\zeta})$

b) Suppose $\rho \in GT_{p,s}$. Let τ_ρ be the trajectory which determines ρ. Set $T_0 = \{*, \langle \underbrace{*, \ldots, *}_{q} \rangle\}$, and, for $i \geq 0$, let $T_{i+1} = T_i \cup T_i^q$.

Suppose that $\underline{\zeta} \notin range(\tau_\rho)$. Then, $type^{TA}_{\underline{\zeta},\rho} = \langle \underbrace{*, \ldots, *}_{q} \rangle$.

Suppose that $\underline{\zeta} \in range(\tau_\rho)$. Let $\omega_\rho = \tau_\rho|_{range(\tau_\rho)}$. For $0 \leq j \leq \omega_\rho^{-1}(\underline{\zeta})$, we define

$\alpha_j^{(\underline{\zeta})}: Z^n \to T_j$,

$$\alpha_0^{(\underline{\zeta})}(\underline{\xi}) = \begin{cases} * \text{ if } \underline{\xi} \in \text{range}(\tau_\rho) \\ \langle \underbrace{*,\ldots,*}_{q} \rangle \text{ otherwise} \end{cases}$$

$$\alpha_{k+1}^{(\underline{\zeta})}(\underline{\xi}) = \begin{cases} \langle \alpha_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_1), \ldots, \alpha_k^{(\underline{\zeta})}(\underline{\xi}+\underline{\gamma}_q) \rangle \\ \qquad \text{if } \underline{\xi} = \tau_\rho(k) \\ \alpha_k^{(\underline{\zeta})}(\underline{\xi}) \text{ otherwise} \end{cases}$$

Then, $\text{type}_{\underline{\zeta},\rho}^{TA} = \alpha_{\omega_\rho^{-1}(\underline{\zeta})}^{(\underline{\zeta})}(\underline{\zeta})$

c) Suppose $\rho \in GT_{s,p}$. Let $\eta_\rho = \tau_\rho|_{\text{range}(\tau_\rho)}$.

We then let,

$$\theta\text{-type}_{\underline{\zeta},\rho}^{TA} = \begin{cases} * \text{ if } \underline{\zeta} \notin \text{range}(\tau_\rho) \\ \varepsilon_{\eta_\rho^{-1}(\underline{\zeta})}^{(\underline{\zeta})}(\underline{\zeta}) \quad \text{otherwise} \end{cases}$$

We then have that,

$$\text{type}_{\underline{\zeta},\rho}^{TA} = \begin{cases} \theta\text{-type}_{\underline{\zeta},\rho}^{TA} \text{ if } \underline{\zeta} \in \text{range}(\tau_\rho) \\ \langle \theta\text{-type}_{\underline{\zeta}+\underline{\gamma}_1,\rho}^{TA}, \ldots, \theta\text{-type}_{\underline{\zeta}+\underline{\gamma}_q,\rho}^{TA} \rangle \\ \qquad \text{otherwise} \end{cases}$$

We then have the following,

THEOREM 6: Let $TA = \langle S, Z^n, \langle \underline{\gamma}_1, \ldots, \underline{\gamma}_q \rangle, GT \rangle$. For each $\rho \in GT_s \cup GT_{p,s} \cup GT_{s,p}$, suppose that

$$|\{\text{type}_{\underline{\zeta},\rho}^{TA} \mid \underline{\zeta} \in Z^n\}| < \aleph_o.$$

Then, there is a pure parallel stratified mixed mode tessellation automaton which simulates TA in real time.

PROOF OF THEOREM 6: This is similar to the proof of Theorem 1 in Grosky and Tsui [3].

QED

REFERENCES

[1] S. Amoroso and G. Cooper, "Tessellation Structures for Reproduction of Arbitrary Patterns," Journal of Computer and System Sciences (v. 5, 1971), pp. 455-464

[2] S.K. Chang, "On the Parallel Computation of Local Operations," Third Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio (1971), pp. 101-115

[3] W.I. Grosky and F. Tsui, "Pattern Generation in Non-Standard Tessellation Automata," Proceedings of the ACM 1973 National Conference, Atlanta, Georgia, pp. 345-348

[4] E. Lieblein, A Theory of Patterns in Two-Dimensional Tessellation Space, Ph.D. Thesis Department of Electrical Engineering, University of Pennsylvania, 1968

[5] M. Minsky and S. Papert, Perceptrons, MIT Press, Cambridge, Massachusetts, 1969

[6] A. Rosenfeld, "Isotonic Grammars, Parallel Grammars, and Picture Grammars," Machine Intelligence 6, (eds.) B. Meltzer and D. Michie, Edinburgh University Press (1970) pp. 281-294

[7] ——— and J.L. Pfaltz, "Sequential Operations in Digital Picture Processing," Journal of the ACM, (v. 13, 1966), pp. 471-494

[8] A.R. Smith III, "Cellular Automata Complexity Trade-Offs," Information and Control, (v. 18, 1971), pp. 466-482

[9] H. Yamada and S. Amoroso, "A Completeness Problem for Pattern Generation in Tessellation Automata," Journal of Computer and System Sciences, (v.4, 1970), pp. 137-176

PARALLEL IMPLEMENTATION OF A TWO-DIMENSIONAL MODEL[a]

Valere J. Kransky, E. Dick Giroux, and Gary A. Long
Lawrence Livermore Laboratory, University of California
Livermore, California 94550

Abstract—A large, serially programmed, two-dimensional mathematical model has been reprogrammed for the CDC STAR-100 and the CDC 7600 computers using parallel programming techniques. The parallel program is currently running on the CDC 7600. The concepts, techniques, and the results of its use are discussed. The parallel program executes efficiently, can be modified easily, and requires no major redesign or reprogramming for conversion to other large-scale parallel machines.

Introduction

The Lawrence Livermore Laboratory began seriously investigating the programming of "parallel" machines in 1969. Our group was assigned the task of reprogramming a large, two-dimensional physical simulation model called HEMP [1]. The equations are Lagrangian and the difference scheme is explicit. Included in the model are hydrodynamics, elastic-plastic flow, multiple sliding, multiple materials, and fracturing. We established the following programmatic goals:

(1) To formulate parallel programming techniques and methods for general use.

(2) To develop a program that would execute with the same source deck on different types of computers. (This is particularly important at LLL because of our history of acquiring new types of large-scale computers.)

(3) To achieve optimum execution rates on parallel computers.

(4) To design the program in a manner that would provide maximum flexibility for frequent modifications.

Vector Programming

After analyzing several different large-scale parallel computers (see Appendix A), we decided that vector programming techniques would satisfy our needs. We define a vector to be a contiguous array of data whose boundaries are specified by a descriptor word. The data contained in a vector may be:

(1) floating point
(2) integer
(3) bits
(4) bytes
(5) characters

A descriptor is a pointer whose low-order bits are a bit-base address that points to the data and whose high-order bits contain the item count of the data set.

The ease with which one can manipulate data is the essential feature of vector programming. We can manipulate vectors with such operations as:

(1) Compress - selects a subset of a vector under the control of a bit vector.
(2) Merge - puts together two vectors under the control of a bit vector.
(3) Compare - generates bits in a bit vector as a results of comparing two vectors.
(4) Transmit index list - collects into a contiguous result vector, discontiguous elements from another vector by using an index vector.
(5) Transmit index destination - stores into discontiguous locations the contiguous elements of another vector by using an index vector.

Such instructions as these permit the "massaging" of data for the various equations found in large-scale scientific programs.

Vectorization of the HEMP Equations

The HEMP problem-solving procedure consists of repeated solutions of explicit equations over a large, two-dimensional grid. Each complete pass through the equations for all grid points (nodes) and zones is a "problem cycle."

Nongeneral Calculations

Certain parts of the two-dimensional mesh (see Fig. 1) must be treated in special, nongeneral ways in the solution of practical problems. Three of the more important of these are described below: (The problem shown in Fig. 1 is not a typical HEMP problem; most problems are more complex and much larger.)

(1) Most of the physical system calculated by the HEMP program include more than one type of material. The materials are in contact with each other at interior boundaries. Often, large displacements along these surfaces take place as the system is solved on the computer. In the program this necessitates the inclusion of special "slide-line" calculations and logic to simulate the surfaces with a decoupled grid.

(2) There are usually two or more materials in a HEMP problem. The behavior of these materials is modeled by equations-of-state. The program must associate the proper equation-of-state with the appropriate grid zone, and calculate material behavior.

[a]Work performed under the auspices of the U.S. Atomic Energy Commission.

(3) Various boundary conditions are associated with the exterior boundaries of the system. These require that the program do selective calculations for certain boundary points.

HEMP Difference Scheme

The HEMP equations-of-motion require the calculation of a line integral at each node. This is represented by the dashed line shown connecting nodes I, II, III, and IV in Fig. 2. In addition, zonal data must be accessed at zones (1), (2), (3), and (4). The exterior boundary-line integrals are calculated in a manner similar to that of the four-zone case, except that coordinates of the node being accelerated are also assigned to one of the surrounding nodes (see Fig. 3).

It was determined that the movement of the boundary points, while subject to various non-general conditions, could be substantially calculated with the same equations (and therefore in the same vector) as the interior grid points. For the purpose of describing the vector techniques used in doing some of the calculations, a tiny grid with a slide-line is shown in Fig. 4. In order that we may treat all nodes with the same equations to obtain a "tentative" acceleration, we expand the nodal vectors with a "geometric bit string." By geometric bit string, we mean a bit string whose bit pattern is dictated by the grid's shape and size. This expansion creates a vector that has vacant elements for the insertion of "phony node" values. The expanded grid is shown in Fig. 5. Through the use of compression, expansion, and controlled-store operations, the phony nodes are assigned the values of the adjoining real boundary nodes. The zonal quantities are expanded out in a similar manner. Now we have a grid that includes phony nodes and phony zones.

Compression with appropriate geometric bit strings is done to isolate the diagonal end points. The diagonal differences (which are zonal-centered quantities) are calculated. These diagonal differences are compressed with another set of geometric bit strings to produce nodal-centered values. These are used to calculate the acceleration terms. New velocities are calculated that are used to reposition the nodes.

The acceleration terms are needed for the boundary calculations (including slide-lines); therefore, it is efficient to calculate acceleration

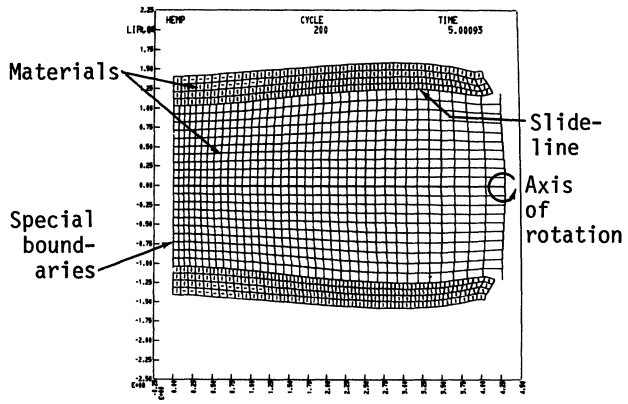


Fig. 1. A HEMP problem.

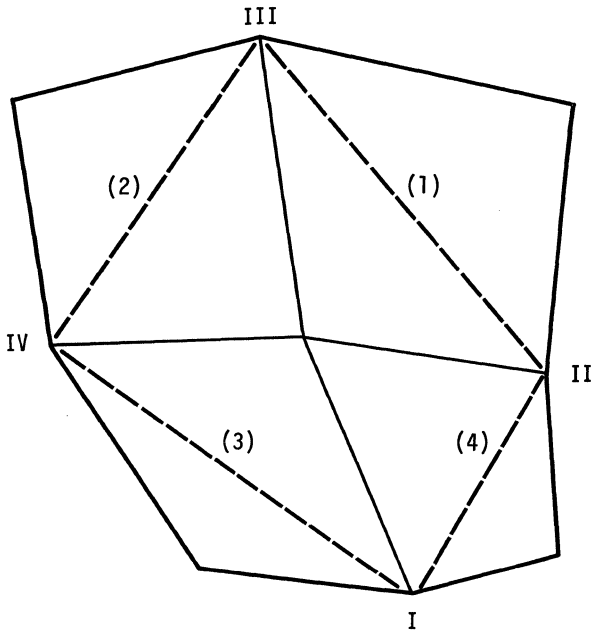


Fig. 2. HEMP acceleration arms.

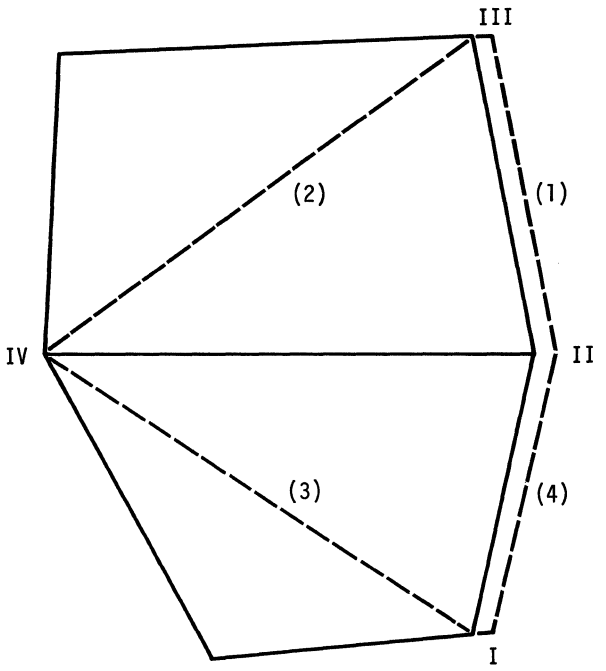


Fig. 3. HEMP boundary arms.

terms in one vector pass. For boundary points, the position is only tentative and may be overridden by subsequent calculations.

The Slide-Line Calculation Logic

Slide-line calculations are complex. They require that nodes and zones on each side of the slide-line be associated with nearby nodes and zones on the opposite side of the slide-line. Figure 6 shows how zones must be mapped across a slide-line. This relationship can change from problem cycle to problem cycle. A search procedure is require to determine this relationship. This was at first thought to be an inherently serial process, and therefore not amenable to vector programming procedures. We vectorized this procedure so that it is done in a few iterations, through the use of cascading compare and compress operations.

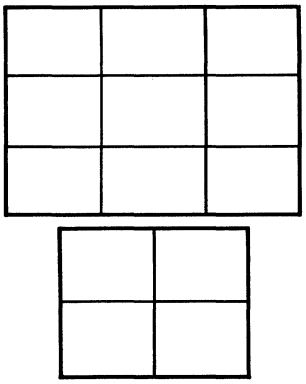


Fig. 4. A simple HEMP grid.

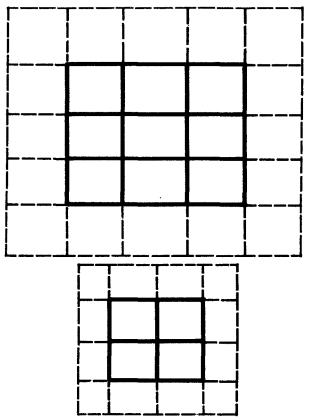


Fig. 5. HEMP grid with phony zones and nodes.

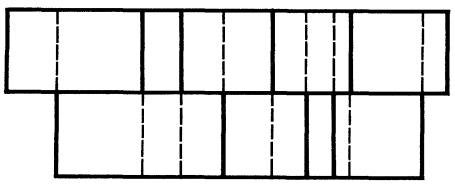


Fig. 6. Slide-line mapping.

An "ordering index" vector is calculated and saved from cycle to cycle. This vector describes the relative nodal positions at that cycle. During each cycle, the ordering index vectors are updated to reflect positional changes. To update the ordering index numbers, all nodes on one side of the line are checked against their previously known solution points on the other side to determine if those solutions are currently correct. The currently correct nodes are compressed out of the vector. A trial ordering adjustment is made with the reduced vector. If found to be satisfactory, these solutions are compressed out. This iterative procedure is continued until all solutions are found. The relative positions of the slide-line nodes change little from cycle to cycle. Ordinarily, one to three iterations are required to update all the ordering index numbers. This process quickly cascades from full-length slide-line vectors to much shorter vectors. Although they are more involved, subsequent slide-line calculations that use these ordering numbers cascade in a similar manner.

Slide-line manipulation includes the building and use of dynamic bit strings. These conditional bit strings are used to compress a sequential index set that is used to fetch or store elements of data within the slide-line vectors. Slide-lines are relatively short, and we may have several slide lines in a problem. Therefore, they are catenated together so that all slide-lines car be calculated in one vector pass. Since the acceleration equations are the same for both sides of the slide-line, alternate sides are catenated together so that all common parts of the calculation can be done in one vector pass.

Equation-of-State Handling

Each problem can have associated with it a number of equations-of-state. In practice, the same equation-of-state is associated with many contiguous zones. This enabled us to:

(1) select zones with like material properties;

(2) arrange the zonal variables into material-related vectors, and

(3) calculate similar zones in one series of vector operations.

A particular zonal grid vector is composed of packed integer fields. One field is a group of numbers that are associated with a particular equation-of-state form. Another field is a material number within that form. The material number within the form is used as an index to access equations-of-state coefficients within that form. When the material properties are to be calculated within a problem cycle, this vector is unpacked (using vector operators) into a number of full-word vectors. A vector compare is done

to determine which zones are associated with each equation-of-state form. The appropriate variables are then compressed out using the resulting bit string. The corresponding material within the form numbers is also compressed out. The form number is used to control a branch to the appropriate equation-of-state coding. The material within the form number is used as an index to select the appropriate equation-of-state coefficients for the material. The program makes repeated passes through this procedure until all zonal material properties are calculated. Because the program is provided with a list of forms for a given problem, only as many s passes are made as there are forms in the problem.

Forking

The equation-of-state calculations are examples of program fork handling. (By forks we mean the selection of various calculational operations. In a serial program this would be done by conditional branching.) Many forks in the program are done dynamically (dynamic forking). In general, a particular vector compare produces a different bit string each problem cycle. The bit string is used to control the calculations. We use two methods of control logic. One method is the previously mentioned vector compare-compress-calculate-expand-and-store series of operations. There is overhead in doing the compressions and expansions in this method. A second method is to use full-length (uncompressed) vectors through both sides of the fork, and then use a bit string(s) to control the storing of results. Here we are calculating many results that are going to be unused, and therefore wasted. Whether to use the compress-expand method or the controlled-store method depends on the bit density of the fork bit string and the amount of calculation on each side of the fork.

When the bit string is relatively sparse on the long side of the fork, it may be more efficient to compress, calculate, expand, and store. When the bit string is relatively dense on the long side of the fork, it may be more efficient to calculate the entire vector both ways and use the controlled store. The method to use is determined through the use of an equation that has in it the vector lengths, the operation types, and the number of operations on each side of the fork [2]. The decision is made dynamically each problem cycle. (This calculation is practical because our vectors are long, and some forks require many operations on one or both sides of the fork.)

Operation Skipping

The issuance of one vector instruction produces a large number of results. This has introduced another time-saving flow-control technique that is not available in serial programming—operation skipping [2]. Some of the HEMP equations contain terms that are not used in a particular problem. In serial programming, it is more expensive to check a flag and possibly skip an operation each time through a loop than it is to issue the unnecessary instruction(s). In vector programming, a single flag check can cause a sequence of vector instructions to be skipped, saving hundreds or thousands of unrequired operations. This test can also be done on the length field of a vector descriptor.

Character Vector Techniques

The HEMP program produces large quantities of printer output. To make this an efficient process, we have used character vector operations to convert binary data to BCD (binary coded decimal). This is one application of vector techniques to areas other than arithmetic number crunching.[a]

Tree Structures

Some index sets, bit strings, and other data sets are constructed at generation time; others are built dynamically during execution. Because of the wide divergence of HEMP problem sizes, shapes, and options, the use of fixed blocks of memory to store this data would be a waste of storage space. To conserve core, we pack this data in memory. We access this data through a series of linked descriptors or "tree structures" that point to the data. The top descriptor points to a vector of descriptors, each of which points to another vector of descriptors or data. Each descriptor tree eventually points to data. If an unusually large data set is required, it takes the needed space for that problem only. If a data set is not required for a particular problem run, it consumes no memory. Tree structures are used in the slide-line, the boundary condition, and other sections of the program. A simple example is shown in Fig. 7.

Core Allocation

Allocation of storage for all vectors needed by the HEMP program is done dynamically, at execution time [4]. The program never allocates more vector space than it needs and/or is physically available in core.[b] The allocation of core is based on the contents of a HEMP data file (Appendix C).

Temporary Results Vectors

The evaluation of a typical vector arithmetic expression requires temporary

---

[a] Charcter vector operations facilitate the writing of interactive, timeshared, and interpretive routines. Character vector techniques can be applied to compilers, loaders, and other system software packages [3].

[b] For some problems the entire grid cannot be held in memory. The program explicitly handles the transfer of data between core memory and disk. Even though a computer (like the CDC STAR-100) may have virtual memory, the overhead associated with page faulting is too costly.

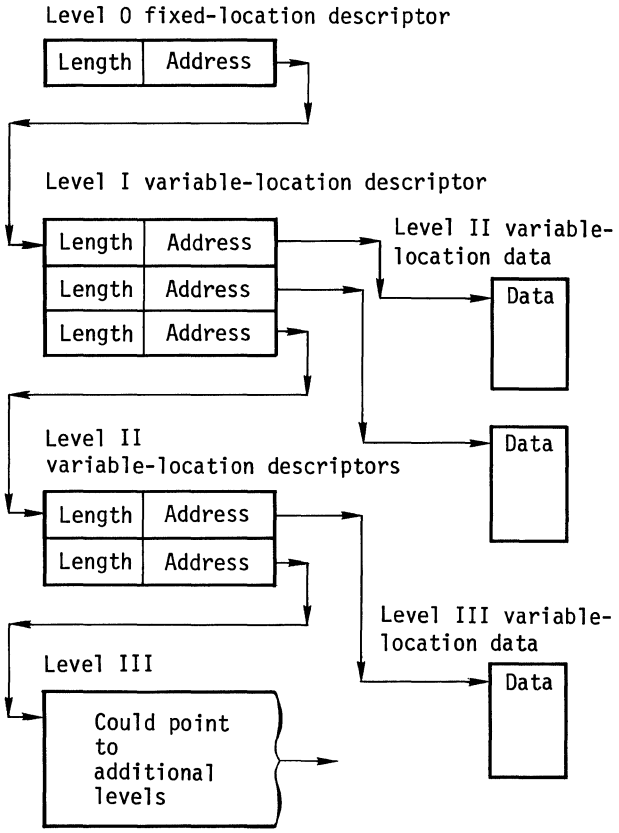


Fig. 7. A tree structure.

vectors of the same length as the result vector. The length of result vectors in a typical HEMP problem is about 1500 words long. As in serial programming, certain calculational results must be saved for later use. In serial programs, this does not present a memory management problem, since each saved result only needs one word of core. In vector programming, this is a serious problem. Each saved result is a vector that requires a large amount of core memory. To alleviate this problem, we reuse the same dynamic vector space as much as possible. This is done through the use of a simple "saved vector" allocation scheme.

The base addresses of saved vectors are kept in a stack. The base addresses of "saved bit vectors" are kept in another stack. Initially, the base addresses in a stack are in ascending order. The number of words between any two adjacent entries in a stack is the same (the length of the longest result vector needed). When a calculation needs a result vector, it takes the next entry (address) from a stack. When a result vector is no longer needed, the address is returned to the stack.

Multiple Vector Passes

When discussing the equations, it was assumed that all vectors were full grid size. This was a simplistic view, taken to make the discussion easier to understand. In actual practice, a problem must be calculated by making multiple passes through the equations. This is necessary because current computers do not have enough core memory available for the save-vectors to be the length of a full grid vector. The number of passes through the equations is a function of the maximum size of a saved result vector and the size of a grid vector.

Prior to each pass through the equations, the grid variable descriptors are adjusted to point to that part of the grid that is to be calculated. If a slide-line(s) is included in a pass, the necessary data vectors and vectors of descriptors for the slide-line equations are constructed. The coordinate vectors and the velocity vectors are merged with phony nodes. The geometric bit vectors are also dynamically constructed each pass.[a]

Vector Programming Aids

The implementation of our vector program has been facilitated by the use of:

(1) an APL interpreter (Appendix D),
(2) programming language extensions (Appendix E), and
(3) new debugging routines (Appendix F).

Current Status

The vector HEMP program is currently running on the CDC 7600 through the use of vector software kernels (Appendix G). Vector HEMP demonstrates marked improvement in execution rate over the serial FORTRAN program (Appendix H). The same vector HEMP source deck that is in use on the CDC 7600 will be used on the CDC STAR-100 computer (Appendix I).

Summary and Conclusion

The following vector techniques were developed and used:

(1) geometric bit strings
(2) phony nodal and zonal elements
(3) dynamic bit strings
(4) static forking
(5) dynamic forking
(6) operation skipping
(7) cascading vector solutions
(8) character vectors for printing results
(9) descriptor tree structures

The vectorization of large scientific computer programs is accomplished by complete redesign and reprogramming.[b] In general, improvements in execution rates will not be achieved by

[a] We have a full set of Boolean bit vector operations to facilitate the construction of the geometric bit strings [5].

simply "vectorizing" a few subroutines. Vector programming techniques can be successfully applied to a wide variety of large-scale parallel computers.

Appendix A. Parallel Computer Analysis

The pursuit of our goals necessitated a detailed analysis of parallel computers. By parallel we mean any computer on which a single operator at the source level will cause multiple, identical machine operations to occur. The operators may invoke a single hardware instruction or a sequence of instructions.

The parallel computers studied included multiprocessor computers, array computers, pipeline computers, and associative computers. Our attention was focussed mainly on large-scale computers in existence or in the planning stage. The computers we investigated were:

(1) the CDC STAR-100 [6]
(2) the Burroughs ILLIAC IV [7]
(3) the Texas Instrument ASC [8], [9]
(4) the CDC 7600 [10]

The STAR-100 and ASC computers use "pipes" through which operands from contiguous memory locations are streamed. The ILLIAC IV uses 64 separate processing elements (PE's) that can all execute the same instruction simultaneously. The STAR-100, ASC, and ILLIAC IV all use "bit logic" to control the storing of operands to memory. For bit logic operations, the STAR-100 has a much more complete set of instructions than either the ASC or the ILLIAC IV. In parallel computation, bit logic replaces the indexes used in serial programming and is the most important nonarithmetic capability of the computers.

The CDC 7600, while not a pipeline or multiprocessor computer, can be an efficient vector machine through the implementation of software kernels (Appendix G).

Appendix B. Vector I/O Library

We used character vector techniques extensively in writing a vector I/O library. All I/O is done by subroutine calls to this library. The HEMP source deck contains no READ, WRITE, etc. type of statement. Only a small part of this library (that part which interfaces directly with the operating system) has been written in a machine-dependent manner. One subroutine in this library is used for printing vectors of numbers. This vector "write" routine can execute up to six times faster than serial FORTRAN "write" routines on the CDC 7600.

(b) Because of the extreme disparity in the calculational speed between a truly vectorized algorithm and a calculation done in a loop on a parallel machine, it does little good to vectorize just part of a program and leave the rest in serial mode. If parallel machines are to perform at anywhere near their capability, all array-type calculations must be vectorized. If arrays are calculated serially, the performance of parallel computers will be degraded by factors of 10 to 30.

Appendix C. HEMP Data File

A HEMP data file is composed of three parts (Fig. 8):

Part I contains various scalar information about the problem and the size (number of words) of Part II (this size changes from problem to problem).

Part II contains descriptor tree structures and data vectors. The data vectors in Part II contain information about:

(1) the size of the grid,
(2) the number of grid variables (the number of variables varies from problem to problem),
(3) the order of the grid variables,
(4) the attributes of the grid variables (i.e., nodal, zonal, etc.),
(5) the boundary conditions,
(6) the slide-line surfaces, and
(7) the equations-of-state.

Part III contains the grid variables, themselves.

Appendix D. APL Interpreter

LLL's APL interpreter was used heavily during the design and algorithmic development phases of vector HEMP programming. Many data manipulation concepts were checked out in APL. The design of the very complex slide-line algorithms was particularly aided through its use. Without APL this would have been a much more difficult task. The value of APL is due primarily to three things:

(1) its interactivity,

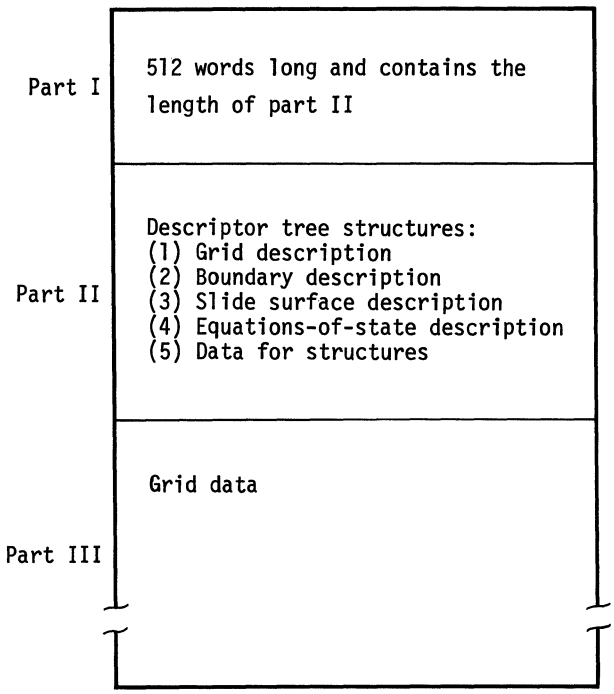


Fig. 8. HEMP data file.

(2) the fact that it includes in its operation set most of the vector operators, and
(3) its extensive debug features [11] - [13].

Appendix E. Programming Language Extensions

The source deck for the HEMP program is written in LRLTRAN [14] and [15]. LRLTRAN is a super-set of FORTRAN IV. LRLTRAN has scalar and vector extensions to FORTRAN IV. The scalar extensions most used were:
(1) The .LOC. statement. Given: J = .LOC.X, then J would contain the absolute core location of the variable, X.
(2) The PARAMETER statement. Given: PARAMETER (LWDB = 60), then all occurrences of the name, LWDB, in the source deck would be replaced by the literal, 60.
(3) A MACRO processor. We only use the character substitution part of the MACRO processor.

Vector Language Extensions

We used the following vector extensions in LRLTRAN:
(1) VECTOR (DV1, V1)— declares V1 to be a vector and DV1 to be the descriptor of vector V1.
(2) BIT B1 VECTOR (DB1, B1)— declares B1 to be a bit vector and DB1 to be the descriptor of B1.
(3) CALL Q8CMPRS—generates code for the vector compress instruction.
CALL Q8MERGE—generates code for the vector merge instruction.
CALL Q8XPND—generates code for the vector expand instruction.
CALL Q8MASK—generates code for the vector mask instruction.
(4) The .CTRL. operator. V1 = B1.CTRL. V2 says store V2 into V1, under the control of bit vector B1.
(5) CALL Q8INLINE(op-code, argument list for the op-code). Op-code is the STAR-100 hexadecimal operation code, and the argument list must match the fields for the operation as defined in the STAR-100 reference manual [6].
The compiler generates inline coding for the STAR-100 for the vector operations. For the 7600, the compiler produces calls to software kernels for vector operations. The source deck for the HEMP program contains only dyadic expressions. This was done primarily to minimize allocation of scratch vector space for complicated equations.

Appendix F. Vector Debugging Routines

When debugging serial programs, octal (or hexadecimal) and/or decimal dumps are sufficient. Vector programs require more sophisticated dumping procedures. We wrote a subroutine, VDUMP, to print "snapshots" of core while running a problem and a utility routine, VDUMP, to do post-mortem dumps. Both routines will dump in the following formats:
(1) bit (pure binary, ones and zeroes)
(2) ASCII
(3) descriptors (on the 7600, they print the octal word address as well as the octal bit base address, and the length field in base 10).
(4) floating point
(5) hexadecimal
(6) integer (base 10)
(7) octal
The routines will also dump vectors of all of the above formats. When printing a vector, the routines always print the descriptor first.
Subroutine VDUMP will also trace a descriptor tree structure, printing all intermediate descriptors and the data vector at the end of each branch. The type of data at the end of a branch is determined by the subroutine and formatted accordingly.
Utility routine VDUMP was written using character vector techniques. It executes about three times faster than our serial dump routine.

Appendix G. Vector Kernels

In evaluating large-scale parallel computers we reached the conclusion that they all could be considered to be "pipe-line" computers. This makes it possible to emulate a sequence of arithmetic and/or logical computer operations. Efficient subprograms called kernels can be written if a computer has:
(1) a reasonable number of registers,
(2) several arithmetic units that can be run in parallel, and
(3) partitioned memory, so that multiple memory references can be made at the same time.
The 7600 lends itself to the vector kernel concept. During the design phase of the HEMP program, two coworkers (Frank McMahon and Lansing Sloan) were programming 7600 vector routines to improve execution speed of FORTRAN programs [16], [17]. We had already developed and simulated similar vector kernels for the ILLIAC IV in 1970. Coordinating with McMahon, we decided to emulate a subset of the STAR-100's arithmetic and bit-byte instruction set for the

Table I

Process	Results per Microsecond 7600	STAR-100
Unoptimized FORTRAN	1.2 - 1.9	?
Optimized FORTRAN	1.6 - 3.3	?
Vector Operations (Dyads)		
Transmit	15	50
Arithmetic		
(+, -, *, /)	2 - 10	12.5 - 50
Compress	5 - 100	25
Merge	4	25
Boolean string	100 - 400	400
Transmit index list	7	4
Vector Operations (Triads)		
	Products per Microsecond	
(V1 * V2 * V3)	10	25

7600. When using these vector kernels (labeled "in-stack loops" or simply "stack-loops") exclusively, we have what we call a vector 7600. These stack loops are mostly dyadic operations (V1 = V2.op.V3), but some are triadic (V1 = [V2.op.V3].op.V4), where V1, V2, V3, V4 are all vectors. Dyadic operations on the 7600 achieve around seven floating-point results per microsecond, while triadic operations attain around ten floating point results per microsecond. Vector execution rates are a function of the item count of the vector operations and the look-ahead techniques used to achieve complete concurrent CPU utilization. The stack-loops, like the STAR-100 vector instructions, require a fixed amount of start-up time. This start-up time becomes negligible for vectors of lengths greater than 400 operands.

Table I compares the results per microsecond of the 7600 stack-loops, normal FORTRAN, and the STAR-100.

Appendix H. Timing of Vector HEMP vs. Serial HEMP

At present the HEMP program is running on the 7600 using the vector stack-loops. To date, timing comparisons show that the vector HEMP program executes 2.2 times faster than the serial FORTRAN HEMP program (Fig. 9). With additional programming improvements and the use of the vectorized editing routines, throughput factors of three are predicted. The approximate number of vector operations performed per pass through the HEMP equations are:

(1) 950 arithmetic operations (including simple data transfers),

(2) 200 full-word logic operations (compare, compress, merge, etc.), and

(3) 100 bit-string operations (bit and byte).

Appendix I. Spanning Computers

None of the vector language extensions appear in the HEMP source deck. All vector operations and descriptor manipulations are buried in simple macros. We have different macro files for the STAR-100 and the 7600. Separate macro files are needed because:

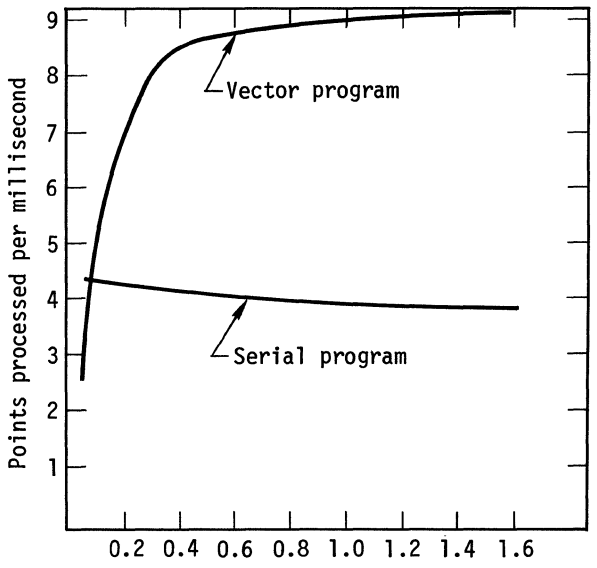
(1) there are differences between the formats of STAR-100 and 7600 descriptors;

(2) different PARAMETERS are used;

(3) some operations require multiple vector instructions on the STAR-100, whereas on the 7600 a subroutine is called.

Another reason for limiting ourselves to dyadic vector expressions is the simplicity of moving the HEMP program to computers other than the STAR-100 and the 7600. A relatively simple preprocessor would handle the macro expansions and the parameter substitutions. The resulting deck would then be FORTRAN IV-compatible (i.e., calls to software kernels would be done during preprocessing).

The use of vectors on a machine like the ILLIAC IV eliminates the necessity of memory management techniques such as "skewing" for optimum PE usage and boundary condition calculations. The use of vectors also results in very little wasted memory, since the memory is packed. For someone who is accustomed to sequential (serial) programming, vector programming presents new challenges. However, our experience at LLL shows that if the equations of a model are appropriate to the use of vectors, they can be programmed in a straightforward manner.



	Points per millisecond	Time for cycle	Number of grid points
Serial	4.45	11	50
	4.25	94	400
	4.20	399	1600
Vector	2.75	18	50
	8.50	47	400
	9.30	180	1600

Fig. 9. Timing comparison of vector-vs.-serial 7600 HEMP program.

Acknowledgment

We would like to acknowledge the contribution of Jerry L. Owens during the initial design phases of the vector HEMP program.

References

[1] M. L. Wilkins, Calculation of Elastic-Plastic Flow, Lawrence Livermore Laboratory, UCRL-7322, Rev. 1 (Jan. 1969), 101 pp.

[2] J. L. Owens, The Influence of Machine Organization on Algorithms, Lawrence Livermore Laboratory, UCRL-74795 (May 1973), 20 pp.

[3] Mary Zosel, A Parallel Approach to Compilation, to be published.

[4] R.E. von Holdt, An Almost Optimum STAR-100 Memory Allocation Scheme, Lawrence Livermore Laboratory, UCID-30042 (March 1972), 21 pp.

[5] G.A. Long and J.L. Owens, Bit and Byte Vector Operations and an Introduction to STARTRAN, Lawrence Livermore Laboratory, UCID-30005 (April 1971), 58 pp.

[6] Control Data Corporation, STAR-100 Computer System Hardware Reference Manual (1971), 125 pp.

[7] Burroughs Corporation, Illiac IV System Characteristics and Programming Manual (June 1969), 120 pp.

[8] Texas Instruments, Inc., Description of the ASC System, M1001P (July 1972), 60 pp.

[9] Texas Instruments, Inc., The ASC Central Processor, H1005P-7112 (Dec. 1971), 50 pp.

[10] Control Data Corporation, 7600 Computer System Reference Manual (1971), 110 pp.

[11] K.E. Iverson, A Programming Language, Wiley (1962), 286 pp.

[12] K.E. Iverson, "A Common Language of Hardware, Software, and Applications," FJCC (1962), pp. 121-129.

[13] H.W. Bingham, Use of APL in Microprogrammable Machine Modeling, Burroughs Corp., pp. 105-109.

[14] R. Zwakenberg, LRLTRAN Extensions, Lawrence Livermore Laboratory, UCID-30019 (July 1971) 17 pp.

[15] R. Zwakenberg, CHAT—A Conversational Compiler, LTSS, Part III, Ch. 207 (Nov. 1968), 68 pp.

[16] L.J. Sloan, private communication.

[17] F.H. McMahon, private communication.

A PARALLEL ASSEMBLER FOR ILLIAC IV

J. M. Randal
Computing Services Office
University of Illinois
Urbana, Illinois 61801

Summary

One of the difficulties envisioned in running a computer of the power of ILLIAC IV, is that of keeping it adequately supplied with a stream of ready-to-run jobs. This paper reports on the progress made in providing an assembler, compatable with one already provided and running on a Burroughs B6700, that runs on ILLIAC IV. Through detailed timing and functional simuation an assembler has been produced which assembles correctly executable object code at, at least 300,000 cards a minute, virtually replacing the need for an "assemble-load-and-go" phase by an "assemble-and-go" phase 100 times faster. From the users point of view the parallel assembler is indistinguishable from the existing serial one, even though its functions are spread over two machines. The paper catalogues other reasons for undertaking the project. Two principal approaches that enhance parallelism in an assembly process, that of arranging the source code in the machine so that it is most amenable to parallel attack, and the delaying of as much semantic analysis as possible as long as possible are outlined. The paper goes on to describe how parallelism is achieved for each stage of the assembly process, an the measured amounts of parallelism are compared and discussed. The paper concludes with a few observations on the practicality of parallel compilation of higher level languages and other so called "inherently serial" processes.

PROCESS COMMUNICATION PREREQUISITES OR THE IPC-SETUP REVISITED

Michael J. Spier
Software Engineering Department,
Digital Equipment Corporation,
146 Main Street,
Maynard, Massachusetts 01754, USA

Abstract -- A careful examination of any existing inter-process communication (IPC) mechanism invariably uncovers the underlying existence of a more fundamental IPC mechanism, which in turn is built on a yet more fundamental IPC mechanism... etc.

This study resolves this indefinite recursion of a self defining mechanism by proposing a certain causality, expressed in terms of a finite list of process communication prerequisites, and based on a non-mechanistic postulate which calls for an area of communication (or *mailbox*) that is by its very nature impervious to mutual interference by the communicating processes.

Given arbitrary processes for which these prerequisites hold, we may logically construct the "very first" *elementary IPC mechanism*, i.e., the one which is not dependent upon its own pre-existence. Such a mechanism is developed in this paper; it is capable of transmitting a single, one-way, one-bit message among processes.

It is suggested that the proposed causality, although arbitrary in many ways (and openly admitted as such) may serve as a convenient intellectual tool with which autonomous sequential processes may be observed and studied.

Keywords: inter-process communication, IPC, IPC-Setup, mailbox, mutual exclusion, process, synchronization.

CR Categories: 1.3 4.32

INTRODUCTION

The *Inter-Process Communication Setup* (or *IPC-Setup* for short [a]) is an initial communication which establishes the conventions by which two or more asynchronous sequential processes agree upon a pattern of harmoniously cooperative behavior.

The concept has been introduced in a previous paper [11] where it was incidental to the main subject. Subsequent reflection has convinced me that this concept deserves a much more thorough investigation. I have observed that the implementability of any given inter-process communication (IPC) mechanism is contingent on the previous availability of a more fundamental IPC mechanism (e.g., in order to implement a producer/consumer buffered communication mechanism [5] we need some mutual exclusion functions such as P and V [7], which in turn require a mechanism to guarantee their internal indivisibility in time, which in turn.....etc.) The recursion seems indefinite.

Consider the following causality, which displays a most perplexing dilemma. In order for two processes to synchronize themselves (e.g., using Dijkstra's P/V functions) they must have had some previous communication to establish the semaphore's identity as well as their agreement to make proper use of the synchronizing primitives. Thus,

- In order for processes to communicate, they must synchronize themselves,
- In order to synchronize themselves, they must have had an earlier communication,
- Which implies a yet earlier act of synchronization,
- Which had to be based on a yet earlier act of communication,

  Which....

That the dilemma is not practically insurmountable is amply demonstrated by the various functional IPC mechanisms that we know of. Evidently, at some basic level (typically the hardware level) the dilemma was resolved through an arbitrary act of Gordian-knot cutting (typically hardware-provided mutual exclusion). Experience has shown, however, that whenever the nature of processes changes (e.g., by the transition into virtual time) lower level synchronization machinery may no longer be valid. When we attempt to design a multi-level processing system, with nested levels of (virtual) parallelism where each successive act of (virtual) processor multiplexing increasingly removes us from our hardware base, it is we who have to provide the Gordian-knot cutting service at appropriate levels of implementation. As we implement successive layers of abstraction, the complexity of our underlying machinery increases. Whereas at the hardware level of a uni-processing computer we achieve the desired mutual exclusion through the simple act of interrupt inhibition, at a much higher level of implementation we may have to consider the properties of virtual processors, the effects of invisible page fault interruptions, the effects of an externally generated user interrupt (when an interactive user presses the *attention* key), etc.

Also, to deal with two mechanisms which are defined in terms of one another is intellectually very frustrating. We may have to accept the "chicken or egg" dilemma when confronted with the real universe, we may wish however to have firmer intellectual control over our artifacts (e.g., computers, computer processes), at least in the sense of establishing a certain causality whose fundamental postulates are external to the observed mechanisms. It is the purpose of this study to suggest such a causality, in the form of a list of conditions which have to be true in order to guarantee n arbitrary processes (m senders and $n-m$ receivers) the ability to exchange a single, one-way, one-bit communication. This intellectual exercise has one ground rule: no pre-existing underlying mechanism is admitted, lest it contain a hidden IPC mechanism and thus leave us no further advanced than before. Therefore, I shall discuss implementation-independent abstract processes.

(a) The term *IPC-Setup* was originally coined by Elliott I. Organick.

A valid question to be raised is: "..why worry about processes which are external to the computer?" As Naur [10] points out, we are creatures of habit and have the inherent tendency to visualize concepts in those terms with which we are most familiar. Being computer professionals, we intuitively think of *process* in the context of *executing computer program*, it being implicitly understood that *computer* translates to "hardware level machine". As operating systems become more sophisticated and the hardware base hidden by intermediary levels of abstraction, our earlier simplistic notion of "process" may no longer hold, indeed become an intellectual impediment. Any insight gained into the properties of the implementation-independent abstract process will however hold true.

In the following unconventional view of non-computer processes, I have guided (indeed biased) the development towards those kinds of processes with which we deal within the confines of the sequential digital computer, and added computer-derived examples to illustrate specific points.

WHAT IS A PROCESS?

Webster's Dictionary succinctly defines the term "process" as "something going on". By selectively narrowing down our choices from this initial vague definition, we can derive an acceptable definition of "process" as it applies to our field of interest.

Let us think of *process* as being *the manifestation of Time, in Space*. The universe in which we exist is subject to the Flow of Time so that it presents itself under different configurations at different points in Time. I apply the term "process" to some time-dependent evolution from one configuration to another. We might visualize the universal set of processes as "threads" of "control" indefinitely stretching from the past into the future, hopelessly intertwined beyond human comprehension.

In order to make sense out of them, study and even manipulate them (e.g., within the confines of a sequential digital computer), we must selectively choose -- among the universal processes -- those specific evolutions in Time ("threads") which we deem worthy of consideration. Thus, I choose to declare "process" to be a *subjective* quality, existing only in the eyes of the observer, who explicitly ignores all other peripheral "threads" in order to avoid confusion.

Examples of such humanly selective observations may range from the macroscopic level, exemplified by the Astronomer contemplating the birth-and-death process of suns and galaxies (or even the, to us, ultimate process of the universe's expansion and contraction), through the Historian tracing the evolution of Mankind or the lifecycle of nations, down to the microscopic level of the Quantum Physicist observing the incredibly short lifespan of some sub-atomic particle.

An observer has to choose for himself not only that one specific "thread" which is of interest to him, but also the *interval* in Time between two successive observations (named "grain of time" in [8],[11]). I attribute the necessity for a subjective choice of intervals to the human brain's limited capacity to assimilate details, and I suggest that there exists a certain "Subjective Time Flow" within our minds, in terms of which sequential processes are best visualized.

We may assume that the human brain cannot make sense out of a visualized process if that process consists of too many discrete details, and that for the sake of coherency the subjective process contains only a limited number of them. Thus, when a human observer translates an evolution in Real Time into an evolution in Subjective Time, he typically chooses intervals between observations which are proportionate to the observed process's period of existence.

And effectively, the Astronomer chooses his interval in terms of billions of years, while the Physicist's interval may be expressed in terms of billionths of a billionth of a second. Yet, within the minds of both these observers, their respective processes may unwind at the same subjective rate of speed, covering a similar number of discrete observations, and may abstractly be related to one another.

Given the periodic nature of observations, the process can no longer be made literally analogous to a continuous thread; rather, it is better represented as a discrete sequence of *dots* which are laid along the axis of the imaginary thread, where each dot corresponds to an observation and where spaces separating the dots correspond to the time intervals between successive observations.

The human observer typically chooses to ignore the existence of the intervals, which to him are irrelevant, and to pretend that the dots are effectively adjacent to one another. Consequently, the discrete sequence of observations may artificially coalesce once more into a humanly coherent, subjectively unbroken thread. By eliminating the real time intervals, we effect the translation of the process's evolution into the flow of Subjective Time. Compare this to Dijkstra's notion of "ordered markers on a scaleless time axis" [5].

Relating the above to our area of interest, namely the study of those processes which exist within digital computers, we see that the notion of "process" is still a subjective quality, dependent on the human observer's choice. A process may, for example, consist of some high level language (e.g., *FORTRAN, ALGOL*) program where observations relate to source language variables and where the interval of time between two successive observations is known to span one or more hardware *cycles*, while from the (interactive or batch) user's point of view a process may consist of a series of system commands, the interval between which may comprise one or more high-level language *programs*.

Lastly, a process may be *sequential* or *non-sequential* [(b)]. Briefly, the former denotes a

(b) In "Cooperative Sequential Processes" [5], Dijkstra illustrates the distinction between sequential and non-sequential processes.

process in which the interval between two successive observations is assumed to consist of a single logical evolutionary step, while the latter denotes a process in which the interval between two successive observations is assumed to consist of a compound logical evolution. The difference between the two is largely subjective and I believe it is safe to state that the non-sequential process has the property that any of its changes of state may be decomposed into a number of parallel sequential processes.

In this paper I adopt the point of view that the sequential process is the "elementary" kind of process. I shall henceforth ignore the non-sequential one by simply choosing to observe my processes at those points of their evolution where they display a single logical change of state (concerning this arbitrary choice of perspective, the reader is referred to observation #1 further on). This choice coincides with our professional custom to consider the computer's *fetch-decode-execute* cycle as a truly sequential progression, even though they might consist internally of two parallel overlapping *execute current instruction while fetching the next one* operations, or even though at a more elementary level the entire computer is known to be implemented as a highly complex parallel hardware logic.

This paper, then, restricts itself to the study of *observably* sequential processes.

PROCESS DEFINITION PARAMETERS

For the purpose of this intellectual exercise, I wish to study processes which are known to exist. The following definitions apply to subjectively observed processes which may or may not inhabit the insides of a digital computer. Therefore I have chosen intentionally to ignore the *processor stateword* concept whose hardware-level definition is clear whereas its implementation-independent definition would not substantially add to our present abstract discussion. The following definition of the term *process* as it applies to a selectively observed "thread" is borrowed from a previous publication [11].

A process is a discrete progression, in Time, of discernible changing states.

Though correct from the abstract point of view, this definition may not prove of great practicality when it comes to the consideration of computer processes. In order to relate it more closely to professional terminology, I introduce the term *memory space* (named "state variable set" in [8]) to denote the set of variables whose changing states may be observed:

A memory space, subjected to the Flow of Time, presents ever changing configurations of discernible states.

An additional helpful concept which allows us to attach somewhat more of a tangible substance to the "time" abstraction is that of the *processor* [c].

[c] The term is used in its most abstract connotation, and must not be taken literally in the meaning of "hardware CPU".

The processor is an abstract "execution agent" (comparable to Johnston's "clerk" [9], and to Dennis' & Van Horn's "locus of control" [4]) which activates an ordered sequence of modifications on the various components of the memory space. If we can hypothesize a memory space which is unaffected by the Flow of Time (e.g., the memory of an inactive computer), we can define the processor as being:

A catalyst capable of subjecting a memory space to the Flow of Time.

With the help of these two terms, we may now devise a definition of "process" which is much closer to our professional terminology:

A process is the activity of a processor within a memory space.

The memory space may assume various aspects, and depending upon its nature the contained variables may be discretely identifiable, or not. By intentionally biasing the discussion towards the kind of processes in which we are interested, I shall postulate for our convenience that the variables with which we deal are discretely addressable by means of universally unambiguous identifiers, or *names*. In the following, the term *mailbox name* is used in the connotation of "universally unambiguous identifier of a memory space component of the type mailbox".

Returning once more to the universal processes, we can envision their flow in Time as individual intertwined threads, where one particular thread represents our process of choice. This thread reaches both backwards and forwards into infinity. It would be useful to delimit the *extremities* of that *portion* of the thread which we actually hold under observation. I would thus add the following two parameters to the definition of a process, these being its *creation* time and its *termination* time, corresponding to the extremities of the thread-portion pointing towards "past" and "future", respectively.

For example, we may consider a human being, going through his daily routine, to be a sequential process. Evidently, his dates of birth and death are relevant parameters in the definition of such a process (if only to preclude any notion of the feasibility of communicating present-day computer science concepts to the late Charles Babbage).

Following is a list of parameters applicable to a single observably sequential process. By assigning values to these parameters, we may talk more precisely about some specific process:

Parameter #1: The *intervals* between successive change-of-state observations [d].

Parameter #2: A *memory space* comprised of all the variables which may be affected by the processor.

Parameter #3: A process *creation* time at which the combination processor/memory space becomes meaningful.

[d] Note that while the intervals need not be of uniform size, their rough order of magnitude is a relevant parameter.

Parameter #4: A process *termination* time at which the combination processor/memory space ceases to be meaningful [e].

I shall also refer to the combined parameters 3,4 as the *process's lifespan*.

Observation #1: I wish to emphasize the fact that the previous definitions and parameters are arbitrarily chosen in order to provide a useful handle on the kind of processes in which we are interested. From the absolute point of view, both definition and parameters are highly ambiguous. Consider: the parameters relate to a *portion* of a *thread* which is our chosen process. From the larger thread's perspective, the above "creation" and "termination" may be considered to be *changes of state* where the "lifespan" in between is considered to be the *interval*. Thus, we may state that *a process is a change of state* and that consequently *a process is a discrete progression, in Time, of processes*. This phenomenon of recursive self-definition is a marked property of the general area of discussion. By considering the process from a conveniently chosen subjective point of view, and by making some well placed arbitrary definitions and postulates, we may gracefully extricate ourselves from this "chicken or egg" situation, which persistently manifests itself in the study of IPC mechanisms. Compare this to the discussion of "image processes" in [8].

INTER-PROCESS COMMUNICATION

Returning again to the universal processes, we may intuitively think of inter-process communication as being an interaction of sorts between two or more threads. The term "communication" conveys the meaning of commonality, or togetherness. I postulate that processes cannot engage is communication unless they *already* have something in common. I further postulate that such commonality must relate to the *memory space* component of the process.

While arbitrary, the postulates make sense when we consider that the process consists of only two components, 1) the *processor*, and 2) the *memory space*. While commonality in Time (such as the co-existence of otherwise unrelated computers) does not -- in itself -- provide us with the ability to communicate, commonality in Space (such as connecting those computers to a common memory bank) definitely does. We can visualize the processes, communicating with one another by depositing messages in the common memory space and/or extracting messages from it. Supposing that the memory space consists of a *medium* which lends itself well to the exchange of messages, we may state that:

Processes communicate by exchanging messages in a commonly accessible medium.

Even though I shall henceforth employ the term *mailbox* (as suggested in [11]) to designate the locality in which messages are exchanged, I have expressly used the term "medium" in the definition, in order to emphasize the rather large variety of possible overlapping memory spaces. While the point is very obvious in non-computer communications, e.g., one person talking to another (the medium being the surrounding air), it applies as well to certain less conventional instances of communication in the computer world, such as the radio link connecting the remote components of the ALOHA system [1], or the IMP's and transcontinental lines of the ARPA network [3], or simply the tapes or disk-packs which may be manually shuttled between independent computer installations.

Of the two process components, *processor* and *memory space*, I have chosen to dismiss the processor as a possible vehicle for the elementary commonality. Is such a dismissal justified? Would an exactly synchronized rate of progression not provide a suitable basis for the communication of two spatially-independent processes? My answer is an emphatic no! Two such processes which knowingly tick along in an exactly synchronized rate may each perform a function based upon the *assumed* concurrent activity of the other, however they do not communicate because each acts *independently of the other's existence* (i.e., one such process may be terminated without affecting the other's behavior, the survivor's activity continues even though its premise of concurrency no longer holds). Still, while a synchronous rate of progression is not in itself sufficient to form the basis for a communication mechanism, it may be usefully applied to an IPC mechanism based on memory space commonality, as shown in the last section of this paper.

I therefore consider that commonality of memory space is *the* essential condition which has to be satisfied if processes are to communicate at all [f]. Processes whose memory spaces are exclusive are by definition incapable of mutual communication, in fact are said to be *protected* from one another [12]!

COHERENT COMMUNICATION

Our processes communicate by exchanging messages in a commonly accessible mailbox. Depending on whether or not message depositions and extractions happen concurrently (remember, at this point

[e] Note that it is the meaningful association which determines the process's existence, and that the disassociation implies the termination of neither processor nor memory space.

[f] A question was raised about this last statement, and critics argued that in systems such as the RC4000 [2] processes communicate not through a shared data base but rather through the intermediary services of the system monitor. I wish to point out that the present study does not concern itself with the more complex ways of building IPC mechanisms, but only with the prerequisites for the most primitive "first" communication. Moreover, the fact that a user process's memory space consists in part of a protected portion of the monitor does not invalidate the fact that within that monitor there is a shared data base in which messages are exchanged; remove that message buffer from the RC4000 system, and inter-process communication is guaranteed to work no more.

we know nothing specific about these processes and their pattern of behavior, excepting the fact that they share a commonly accessible mailbox), such communication may be *coherent* or *interfering*. The communication is said to be interfering if the value extracted from the mailbox by some receiving process *B* is not identical to the same value previously deposited in the mailbox by some sending process *A*. Interference (and the resulting message incongruity) may occur when several depositions are made concurrently, or when extraction is attempted while deposition is still under way. The communication is said to be coherent when it is not interfering.

Coherent communication is characterized by the fact that a message extracted from a mailbox is guaranteed identical to the same message previously deposited in the mailbox. We may not know who deposited that message in the mailbox, nor what it means, but we are assured of the congruity of that message.

It is the coherent message which is of interest to us. A way to assure coherency must definitely be an important constituent of the process communication prerequisites that we seek. I shall therefore further postulate that the mailbox itself, by its nature, possesses a property of guaranteed message congruity such that whenever two or more processes simultaneously attempt to either deposit a message in it, or extract a message from it, only one process at a time will be allowed to do so; the exact succession into which this enforced sequentiality will be resolved is undefined and immaterial [g]. Not knowing who created the mailbox with its magical property, nor how this property is functionally enforced, we can only surmise that it is the handiwork of some benevolent instrumentality. Still, assuming the mailbox's availability, we may state that:

Coherent inter-process communication is an interference free exchange of messages in a commonly accessible mailbox.

In the following, I shall refer to the mailbox as being "interference-proof". The interested reader may wish to study the details of the ALOHA system [1], whose mailbox (i.e., a certain bandwidth of the electromagnetic spectrum) is not guaranteed to be coherent; ingenious encoding techniques reduce the probability of interference to a very low factor, but the fact remains that coherency is not guaranteed.

Observation #2: Let us for the time being accept the premise of a mailbox which allows only exchanges of coherent information, even though it is unclear how such a mailbox might be constructed. Later on I shall 1) postulate a very elementary two-state mailbox whose implementability will not be subject to doubt, and 2) suggest that more elaborate mailboxes may be constructed with the help of the elementary one.

[g] The necessity for such a mailbox (and its magical property of coherency) is a fundamental postulate of any multiple processor computer system; e.g., at any given time, memory is interlocked to all but a single processor.

MEANINGFUL COMMUNICATION

Let us attempt to construct an initial model of communicating processes. For the sake of simplicity, I shall deal with two processes only, a sender and a receiver. The following is, however, valid for any number *m* of sending processes, and *n* of receiving processes.

At this point, all that we may assume about our processes are the characteristics discussed earlier; namely, their sequentiality and their memory spaces which overlap a commonly-accessible, interference-proof mailbox. Two processes named *A* and *B* communicate as follows: 1) the sender, process *A*, deposits a message *Msg* in the mailbox; 2) the receiver, process *B*, copies the contents of the mailbox into some private locality *L*. The sender would perform

mailbox := Msg;

and the receiving operation would be

L := mailbox;

Even though the communication is coherent, it is completely meaningless. Consider the following:

1) By what right can it be assumed that process *A* has ever had the intention of depositing anything whatsoever in the mailbox? Assuming that it did have such an intention,
2) Are processes *A* and *B* actually referring to the same mailbox? Is it not possible that process *A* innocently deposits its message in some $mailbox_1$ while process *B* persists on extracting an assumed message from some other $mailbox_2$? We may graciously submit that the mailbox is one and the same, still
3) Process *B* may be the speedy one, extracting an assumed message from the mailbox before the slower process *A* has ever had the chance to perform the intended deposition. And if we agree to discard this possibility as well, then
4) Having received its coherent message, process *B* is no further advanced because it has no way of knowing what the message is supposed to mean [h].

If we wish to engage in meaningful communication, we have to make sure that the above uncertainties are satisfactorily resolved. We may not, at this point, have any specific remedy; this need not deter us from describing the effect of such a solution by establishing a list of conditions which are essential to meaningful communication:

Condition #1: The processes have to agree in *advance* (and that means prior to the creation

[h] We may better appreciate this fact if we consider the task of the military cryptographer, faced with the decoding of an intercepted coherent enemy message; he is capable of success because he knows the other guy's language. Process *B*'s task is hopeless, it knows absolutely nothing about process *A*. Consider the hopelessness of deciphering Egyptian hieroglyphs prior to the Rosetta Stone Discovery.

time of any of the communicants) on their intention to communicate some time in the future. Remember, we still choose to remain in a state of blissful ignorance concerning these processes, thus the *pre-natal* instance is the only logically-safe point in time.

Condition #2: They must agree on the exact identity of the single mailbox in which messages will be exchanged.

Condition #3: They must agree on their respective sender/receiver roles.

Condition #4: Mandatory sequentiality has to be imposed on the act of communication. First the sender has to deposit his message, and only then may the receiver extract it from the mailbox.

Condition #5: The communicating processes have to have agreed, in advance, upon the way in which messages are to be interpreted and understood.

PROCESS SYNCHRONIZATION

In the above list, condition #4 requires that the communicating processes adjust their relative speeds; as they progress independently in Time, when their respective instances of communication arrive, these instances have to become *aligned in Time* in a predetermined way. We use the term "synchronization" to denote such an alignment.

We still know nothing specific about these processes, hence cannot trivially choose between alternate schemes of synchronization which may all seem *a priori* to be equally attractive. Possibilities may include 1) the sender having the ability to slow down the receiver's progression in Time, 2) the receiver having the ability to cause the sender's speed to be accelerated, etc.

A simple, though arbitrarily chosen, scheme to assure that message extraction will happen later in Time than message deposition would have the receiver process *voluntarily* enter a waiting state if the message has not yet arrived. This method is chosen because it lends itself best to the kind of process synchronization practiced within digital computers, and is hence typical of existing computer program IPC mechanisms. Its adoption requires that we add two more conditions to our list.

Condition #6: The receiving process is capable of determining at any given moment whether or not a message had actually been deposited in the mailbox.

Condition #7: If a message had not yet been deposited, the receiver must be willing, and capable (!), of suspending its progression until such time when the message has arrived.

This introduces one last complication. Condition #6 calls for the process's ability to inspect the mailbox's contents and determine whether or not a message had arrived. Presumably it will do so by testing the mailbox for some specific value which may be either a *non-message* or a *message* value. What can that value be? If the receiver tests for a non-message value, it is not possible that the sender has innocently used that very same value for its message and thereby mislead the receiver? Or if the receiver tests for a message value, is it not possible that the mailbox might -- by some unfortunate chance -- have been pre-initialized to that very same value thus misleading the speedier receiver into acceptance of a supposed communication, when in fact no such transaction has yet taken place? We must therefore complete our list of conditions with the following two:

Condition #8: The communicating processes have to have agreed on a single non-message value *Vinit* to be interpreted as "no message has yet arrived" (by agreeing on a non-message value, we leave the door open for a possible variety of meaningful message values).

Condition #9: The mailbox is guaranteed to have been initialized to the non-message value *Vinit* prior to the creation time of any of the communicating processes (again, within the present context of discussion, this is the only logically defensible point in time).

PROCESS COMPATIBILITY

Having established the need for process synchronization, we must preclude from our consideration those processes which are -- by virtue of their temporal characteristics -- inherently incompatible with one another from a synchronization point of view. Of the process definition parameters, the *interval* and the *lifespan* may assume values which would make the processes incapable of meaningful synchronization. I wish to remind the reader that this paper does not engage in the exercise of process construction, but in the observation of already existing processes. Thus, the three incompatibilities listed below are valid so long as we recognize our inability to influence the processes' temporal parameters (i.e., we preclude from our consideration artifacts such as "clocking processes" [8]).

Incompatibility #1: The processes' lifespans may be exclusive; one process's termination time may have passed well before the other process's creation time has yet arrived. This case was exemplified by the earlier mentioning of Charles Babbage. This condition is asymmetrical in that the expiration of the sending process might still be acceptable (i.e., Charles Babbage effectively did leave a message for posterity) whereas the premature expiration of the receiver is obviously inadmissible. If we postulate that the elementary communication mechanism that we seek should be indifferently functional for any sender/receiver configuration (i.e., should allow any two or more processes to adopt either role) then we have to insist that the processes' lifespan overlap the period of communication.

Incompatibility #2: The processes' lifespans may overlap the period of communication, but only partially so that the sender's termination time arrives before it has had the opportunity to properly conclude its part of the transaction. This may cause the receiving process indefinitely to suspend its progression in

anticipation of a message whose deposition was never satisfactorily carried out. For processes to engage in guaranteed non-fatal meaningful communication, the sender's termination time must lie well outside the period of communication, known as the *critical section* in the process' lifespan [5].

Incompatibility #3: The *size* or *relative order of magnitude* of the processes' respective intervals must be compatible. It is difficult to be very specific about the exact kind of interval compatibility that is desired; the reader must have noticed by now that the main thesis of this paper consists of emphasizing the very nebulous nature of the overall subject.

Nonetheless, this is a very real problem best exemplified by the inability of a virtual processor, executing within a paged virtual memory, to correctly service real-time applications. The interval between two successive virtual machine cycles is undetermined, while the correspondent real-time process requires guaranteed service within specific time bounds.

ELEMENTARY COMMUNICATION MECHANISM

Let us now construct the "first" and most elementary communication mechanism which would satisfy all of the requirements mentioned earlier. The processes are assumed to be inherently suitable for mutual communication in the dual sense of overlapping memory space and temporal compatibility. Concentrating on the communication mechanics alone, we are faced with one major difficulty which is the creation of the interference-proof mailbox.

It is possible to construct a very primitive mailbox which has the capacity for a single bit of information only. The domain of the mailbox is thus restricted to two possible values which we shall name *TRUE* and *FALSE*. By nature of its definition, the mailbox could never be found in a state which is neither *TRUE* nor *FALSE* and it therefore fulfills our requirement of inherent coherency.

If we assume that such a mailbox was originally created by some benevolent *instrumentality* (i), placed in the common memory space and thoughtfully initialized by the instrumentality to the *FALSE* state, then we may establish the following scheme for communication, where a sender process sets the mailbox to *TRUE*, and where the receiver process interprets the *TRUE* state as meaning "a message has arrived".

Also, the receiver process would interpret the *FALSE* state of the mailbox as meaning "a message has not yet arrived". The receiver may now suspend its *logical* progression by insistently testing the mailbox for a *TRUE* state. The mechanism would work as follows: the sending operation corresponds to

mailbox := TRUE;

(i) The computer hardware designer who provides us with an interlocked memory, or even with hardware implemented semaphores [7], is a good example of what I would call "instrumentality".

while the receiving operation is of the form

busyloop:
IF mailbox = FALSE THEN GOTO busyloop;

Observation #3: It is important to note that while the receiving process's progression in Time is by no means affected, we have achieved the functionally desired effect by imposing on that process a rule of behavior which guarantees that its memory space is subjected to no further modification while the *mailbox = FALSE* condition prevails. As mentioned in the last section of this paper, computer processes have the highly interesting property in that their Flow of Time may be literally stopped and restarted.

The above mechanism is the most rudimentary imaginable, capable only of a single one-bit one-way (or "simplex") communication. By reciprocally using two mailboxes and by inverting the processes' sender/receiver roles, we may construct a mechanism capable of sending two single one-bit messages in opposite directions (known as "duplex" communication channel). Combinatorial usage of many such mechanisms allows us to construct a "multiplex" channel, or a "bus" (parallel simplex channels) as encountered in the innards of computers. The information transmission capability of the elementary mechanism is very poor. Each mailbox may be used only once, and the existence of the message is also its value. We may detect the arrival of such a message, but may not transmit any additional intelligence.

I name the mechanism which allows us to transact a single one-way one-bit communication *elementary communication mechanism*, and re-state that its existence is contingent on the availability of a magical interference-proof mailbox, provided (in a properly initialized state) by some benevolent instrumentality. If we do not accept the premise of such an initial mailbox, we may never be able to construct the very first IPC mechanism.

MUTUAL EXCLUSION

There is no point in elaborating the limited usefulness of the elementary communication mechanism. Its significance lies in the fact that it might serve us as a building block for the construction of more useful, more sophisticated IPC mechanisms. For example, a useful mechanism -- such as the *WAIT/NOTIFY* functions suggested in [11] -- would be capable of a continuous *sequential* transmission of variable-length information-laden messages, and would also have a buffering effect minimizing the necessity for non-productive waits. We may visualize the communication-channel effect of such a mechanism in the form of a one-way information "pipeline"; the sender stuffs messages into one end, the receiver opens his faucet whenever necessity requires and draws information out of the other end. The realization of such a mechanism hinges on our ability to construct an interference-proof "pipeline"-type mailbox.

Yet if we reconsider the meaning of "interference-proof" we realize that all that is necessary is the assurance that among *N* communi-

cating processes, *N-1* would refrain from accessing the mailbox while the *Nth* process is manipulating it. Our primitive mailbox guarantees this by its very nature; the same effect can be achieved for any arbitrary component of the memory space if the processes agree *voluntarily* to adopt such a pattern of non-interfering behavior whenever the mailbox is being accessed.

Such agreement should be semantically expressible. Let us postulate a pair of functional mutual-exclusion brackets names *MUTEX/XETUM* (j). Whenever a process intends to access the mailbox, it announces the intention by performing a *MUTEX(mailbox)*. When it has finished manipulating the mailbox it signals the mailbox's availability by performing a *XETUM(mailbox)*. The logic of these functional brackets is such that at any given time at most a single process will be manipulating the mailbox.

Observation #4: The nature of our mailbox is now radically changed! While the elementary mailbox guaranteed coherency by its very nature no matter how the processes chose to access it, a mailbox whose coherency is achieved via the application of *MUTEX/XETUM* will remain interference-proof only as long as the communicating processes choose harmoniously to cooperate with one another. Let a single communicating process "do its own thing" and we are faced with an unbridgeable communication gap.

And how would we manufacture these functional mutual-exclusion brackets? Their nature implies a whole new dimension of underlying communication and cooperation among processes, and it might be argued that it is foolhardy to re-invoke the "chicken or egg" situation by proposing to solve a problem through a mechanism which manifests the same problem. We might have been forced arbitrarily to postulate the existence of *MUTEX/XETUM* as we have done earlier. Luckily, in his "Solution of a Problem in Concurrent Programming" [6], Dijkstra has demonstrated that the availability of an interference-proof mailbox is sufficient to assure the implementability of *MUTEX/XETUM* (k). And once we have constructed these mutual-exclusion brackets, the road is clear to the construction of mailboxes of arbitrary complexity and sophistication.

(j) The use of the inverted left bracket clause to designate the right bracket is inspired by the BLISS [13] systems programming language. The name *MUTEX*, originally used by Dijkstra [5] to designate a mutual exclusion semaphore, has for some time been used by rank-and-file programmers to designate the mutual-exclusion P [7] operation (possibly because of the confusion between *mutual-exclusion* and *private* semaphores); it is employed here *post facto*.

(k) Note that Dijkstra's mechanism requires, in addition to the coherent binary mailbox, a coherent integer mailbox *k*. Disregarding the possibility of modifying the algorithm to all-binary, we can safely postulate the integer mailbox for our purposes, because the hardware designer knows how to build it out of binary mailboxes (flip-flops).

THE IPC-SETUP

The existence of the elementary communication mechanism is conditional, depending upon a number of arbitrary postulates and conditions. These were introduced in a sequence dictated by the orderly development of the subject. These process communication prerequisites are the essence of this study, I shall therefore re-state them in an organized fashion. They are subdivided into three classes 1) conditions relating to the very nature and existence of processes, 2) conditions relating to the postulated, instrumentality-given mailbox, and 3) conditions relating to the processes' cooperative behavior.

First, we have to delimit our consideration to processes whose nature makes them capable of meaningful mutual communication. Processes which wish to communicate belong to a "set of compatible processes". The set is defined by the processes which communally display all of the properties listed below. A process that does not possess all of the properties peculiar to a given set does not belong in that set, but assuredly belongs in some other set.

Property #1: All *N* communicating processes must be sequential (l).

Property #2: All *N* memory spaces of the communicating processes must overlap (at least) a single common subset.

Property #3: All *N* lifespans of the communicating processes must overlap in Time.

Property #4: The intervals typical of all *N* communicating processes must be compatible.

Property #5: None of the *N* processes' termination times must arrive during the respective process' critical section.

Second, we have to postulate the availability of an interference-proof mailbox. This requires in turn that we postulate the existence of a benevolent *deus ex machina* or "instrumentality" which has a vested interest in letting the processes communicate, and which manifests this interest by conveniently providing the required mailboxes.

Postulate #1: There exists an instrumentality whose purpose it is to create mailboxes.

Postulate #2: A mailbox has the natural inherent property that its contents can never be in an unstable or incoherent state.

Postulate #3: The mailboxes are accessible to all *N* communicating processes because the instrumentality saw to it that they reside in the common memory space(s).

Postulate #4: The instrumentality has thoughtfully pre-set all the mailboxes to a non-message *Vinit* state at a point in Time which precedes

(l) This condition applies to the communication model developed in this paper. By devising a list of different process communication prerequisites, a model conductive to non-sequential process communication may undoubtedly be devised.

the creation time of any one of the *N* communicating processes.

Third and last, the processes' rules of behavior must be set up in a manner which will guarantee that they always adopt a pattern of harmoniously cooperative behavior insofar as communication is concerned. For this purpose we conveniently postulate another entity, that of the *programmer*, who is responsible for implementing these rules of behavior into the logic of those *N* communicating processes. The cooperative behavior is made possible by the adoption of certain conventions which all *N* processes agree to respect. Such common knowledge of conventions is in itself a manifestation of a previously transacted communication. As suggested in [11] I name this manifestation of pre-natal communication *IPC-Setup*. It originates in the single mind of the single programmer (thus, no "chicken or egg" dilemma) who incorporates it into the essence of the *N* processes prior to their creation time. The nature of the conventions depends on the nature of the communication; following is the list of conventions required for the existence of our elementary communication mechanism:

IPC-Setup #1: The *N* communicating processes agree on the common name of the single (commonly accessible) mailbox to be used.

IPC-Setup #2: The processes agree to use that mailbox for the purpose of communication.

IPC-Setup #3: The processes agree on their respective sender/receiver roles.

IPC-Setup #4: The *N* communicating processes agree to interpret the value Vinit, with which the instrumentality is known to have initialized the mailbox, as a non-message implying "no message has yet arrived".

IPC-Setup #5: The receiving processes agree to interpret any non-*Vinit* state of the mailbox as implying "a message has arrived".

IPC-Setup #6: They further agree to assign a meaning to any non-*Vinit* state of the mailbox and to interpret that value in some meaningful way.

BACK TO PRACTICALITY

A thesis was presented to the effect that organized, deliberate and meaningful communication does not spontaneously erupt into being; rather, it can always be traced to some pre-existing instance of preparation and communication. Many definitions, decisions and postulates made during the development of this paper were admittedly arbitrary, and openly acknowledged as such. My purpose was not to insist on a certain dogmatic point of view, I do not believe that this nebulous subject would ever accommodate dogmatism, but rather to convey some insight into the complex issues that have to be resolved before we can safely communicate a single bit of information, once only, between processes.

This study was motivated by the need to resolve the "chicken-or-egg" dilemma. It proposes a certain hierachy of causality: the interference-proof mailbox, the IPC-Setup, the elementary communication mechanism, and lastly the mutual exclusion function. Some other such hierarchy and its related list of communication prerequisites may undoubtedly be developed; I doubt that such a list of different prerequisites would be any less voluminous than the one proposed.

The causality (and terminology) developed in this paper lend themselves to the description and understanding of various IPC mechanisms. To illustrate, let me present the workings of the asynchronous serial simplex channel connecting a sending source to an electro-mechanical printing device (e.g., teletypewriter).

Both sending and receiving process are essentially devoid of buffering memory. The sender generates its message, the receiver intercepts it and acts on it. The commonly accessible mailbox consists of an electrically conducting wire connecting both machines. The presence/absence of current, or a high/low voltage arrangement represent the two value-states of the mailbox. The mailbox is reasonably coherent but is not interference-proof; it is said to be susceptible to "noise".

The list of process communication prerequisites applicable to this example is somewhat different from the one developed in this paper. In order to make the mailbox capable of transmitting *two* meaningful kinds of messages, namely bits *zero* and *one*, the mechanism does not support the notion of a non-message *Vinit*. Instead, by means of two (instrumentality-provided) synchronous clocking devices respectively incorporated into the two processes, each process is decomposed into a continuous sequence of "mini-processes" (the reader may wish to re-read observation #1). The lifespan of each mini-process is delimited to the duration of a single clock tick, and the mailbox is reset to the *FALSE* state at mini-process creation time. If a *TRUE* state is detected by the receiving process during its short lifespan, it is interpreted to mean "a *one*-bit has been received", otherwise upon its termination time a *zero*-bit message is assumed. A new mini-process is created and the same communication ritual is re-enacted.

By adding a clocking device and modifying the IPC-Setup, we have instilled more usefulness into the elementary communication mechanism. Also note that the judicious choice of "which process do I wish to observe" (i.e., "mini-process" vs. the larger "thread") is the key to this functional presentation.

The elementary communication mechanism is not very useful to the programmer. The effort of manufacturing functions *MUTEX/XETUM*, with which then to construct a more elaborate communication mechanism, is far from negligible. We therefore habitually require the availability of some prefabricated mutual exclusion primitives (such as interrupt inhibition) which we then consider, from the programming point of view, as elementary.

IPC mechanisms are typically designed to be easily applicable to the kind of processes which

exist within computer systems. They therefore are cognizant of two peculiarities of the computer process 1) the process is typically of a cyclic nature (i.e., may be decomposed into a repetitious sequence of essentially identical "mini-processes"), and 2) the virtual time flow in which the processes exist may literally be stopped and started.

The process's cyclic nature implies that unless the correspondent processes are pre-synchronized, harmoniously ticking away as does the exemplified teletype, a yet un-received message may erroneously be overwritten by the next, and the next...etc. We typically rule such pre-synchronization out because asynchronous processes can normally be put to better use. Instead, we implement a "pipeline" capability into even the binary mailbox, trading off inherent synchronization vs. inherent buffering effect. Such a buffer, or list of one-bit messages, is trivially implemented in the form of a binary counter. Assuming the availability of *MUTEX/XETUM*, the sending operation is now:

```
MUTEX(mailbox);
mailbox := mailbox + 1;
XETUM(mailbox);
```

and assuming that the zero state implies "mailbox is empty", the receiving operation is

```
busyloop:
    MUTEX(mailbox);
    IF mailbox = ∅ THEN
    BEGIN
        XETUM(mailbox);
        GOTO busyloop;
    END;
    mailbox := mailbox - 1;
    XETUM(mailbox);
```

Virtual processors are artificial constructs derived from some real life hardware CPU resource. In a system with *N* virtual processors, any non-productive activity of one is to the detriment of all others, wastefully misusing a finite CPU resource which could be put to some good productive use elsewhere. Our busyloop is archtypical of such wasteful behavior.

It is therefore economically desirable to include in the IPC mechanisms which are put at the programmer's disposal a provision by which a waiting process not only suspends its *logical* progression, but literally causes its virtual *time flow* to stop. Once stopped, the process is said to be "dormant" and can no longer insistently test the mailbox for the awaited message. It is the cooperative sending process which, after having deposited its message, helpfully "nudges" the dormant process back into wakefulness.

REFERENCES

1) Abramson N, "The ALOHA System", Computer Communication Networks, Abramson & Kuo Editors, Chapter 14, Prentice Hall 1972.

2) Brinch Hansen P, "The Nucleus of a Multiprogramming System", CACM April 1970, pp. 238-242.

3) Carr C, Crocker S, Cerf V, "Host/Host Communication Protocol in the ARPA Network", Proc. 1970 SJCC, pp. 589-597.

4) Dennis J B, Van Horn E C, "Programming Semantics for Multiprogrammed Computations", CACM March 1966, pp. 143-155.

5) Dijkstra E W, "Co-operating Sequential Processes", Programming Languages, Genuys Editor, Academic Press 1968, pp. 43-112.

6) Dijkstra E W, "Solution of a Problem in Concurrent Programming", CACM September 1965, pp. 569.

7) Dijkstra E W, "Synchronizing Primitives", Appendix to "The Structure of the 'THE' Multiprogramming System", CACM May 1968, pp. 345-346.

8) Horning J J, Randell B, "Process Structuring", ACM Computing Surveys, vol 5 #1, March 1973, pp. 5-30.

9) Johnston J B, "Structure of Multiple Activity Algorithms", Proc. 2nd ACM SIGOPS SOSP, October 1969, pp. 80-82.

10) Naur P, "The Place of Programming in a World of Problems, Tools and People", Proc. IFIP Congress 1965, pp. 195-199.

11) Spier M J, Organick E I, "The Multics Interprocess Communication Facility", Proc. 2nd ACM SIGOPS SOSP, October 1969, pp. 83-91.

12) Spier M J, Hastings T N, Cutler D N, "An Experimental Implementation of the Kernel/Domain Architecture", Proc. 4th ACM SIGOPS SOSP, October 1973.

13) Wulf W S, Russel D B, Haberman A N, "BLISS: A Language for Systems Programming", CACM December 1971, pp. 780-790.

THE EXPERIMENTAL IMPLEMENTATION OF A COMPREHENSIVE INTER-MODULE COMMUNICATION FACILITY

MICHAEL J. SPIER (a)
Department of Software Engineering
Digital Equipment Corporation, Maynard, MA 01754

Summary

In 1972, The Digital Equipment Corporation sponsored a limited-objective research project to investigate the properties of the new kernel/domain systems architecture, whose theoretical model was earlier developed by Spier [1]. A companion paper [2] reports on that project. The domain is a monitor (or supervisor, executive)-like local independent address space which may be mapped over a collection of (mostly) exclusive memory space partitions to provide a protected run-time environment. Similar to the classical monitor, control may be transferred into the domain through predesignated inter-domain entry points named gates [1]. In a domain system, supervisory code no longer resides in a single monolithic monitor, but is distributed among a number of supervisory domains; of these, the most central and most critical supervisory domain is named kernel [1] [2] [3]. The kernel is responsible for basic resource management only and is by definition devoid of any decision making code.

If we view the term *process* as meaning the *activity of a processor within a memory space* [4] then the execution of a processor within a domain (read, exclusive memory space) is an independent process. In a domain system where a single user computation may cause the activation of many domains, that computation's sequentiality may be viewed as the sequential activation of many processes. For the sake of conformity, we chose to apply the term *process* to the larger sequentiality, and coined the term *domain-incarnation* [2] to designate the execution of a single domain by a single processor. The transfer of control from one domain to another, although synchronous and sequential, displays some of the properties inherent to interprocess communication (IPC) mechanisms [4]. Our kernel-implemented comprehensive inter-module communication mechanism handled the following cases:

1. The explicit sequential activation of a procedure entry point, expressed in the form *CALL procedure(argument-list);*
2. The implicit sequential activation of a procedure entry point, currently known to be the handler for some predesignated condition, expressed in the form *SIGNAL condition(argument-list);*
3. The explicit non-sequential activation of a procedure entry point by some other process, expressed in the form *INTERRUPT process,procedure(argument-list);*
4. The implicit non-sequential activation of a procedure entry point by some other process, where the procedure is currently known to be the handler for some predesignated event, expressed in the form *NOTIFY event(argument-list);*
5. The abnormal cancellation of a sequence of calls through a *non-local GOTO* to a predesignated entry point declared to be a handler for the unwind condition, expressed in the form *UNWIND;*

Note that both conditions and events came in two flavors: 1) *LOCAL* to remain in effect only as long as the procedure activation that declared them, and to automatically be terminated upon *RETURN* from that procedure activation, and 2) *GLOBAL* to indefinitely remain in effect until explicitly terminated.

Thus, all the above inter-module communication functions were kernel-managed and invariably resulted in the argument-carrying formal activation of a procedure entry point, to uniformly be dismissed via a formal *RETURN;* Both of our inter-process communication functions were a software simulation of the classical hardware interrupt. Given our predominant concern to keep the kernel application independent, the software interrupt facility seemed to be the most general mechanism conceivable. A special kernel-call of the form *SLEEP(time-limit);* would put the calling process into a dormant state to be re-awakened when either 1) an *INTERRUPT* or *NOTIFY* signal is received, or 2) the time-limit has expired, whichever happened first.

The asynchronous nature of *INTERRUPT* and *NOTIFY* implied a certain minimal argument-buffering facility within the kernel. Also, the activation of a procedure entry point by either of the asynchronous invocations caused all further asynchronous signalling to that same process to be inhibited, until a return was made. We had additional, more specific kernel-calls to more finely control the inhibition/reactivation of inter-process signals, as well as mutual exclusion functions *MUTEX/XETUM* [4] which were also kernel-implemented.

Finally, note that our choice of the software-interrupt facility did not preclude the availability of more classical IPC interfaces, such as *MSG:=WAIT(mailbox);* and *NOTIFY(mailbox,MSG);* [5]. Such mechanisms could be implemented within dedicated supervisory domains by means of the tools just described. One of the reasons for choosing these more general tools was to provide the ability for virtual user computations to be multiprogrammed.

(a) This paper reports on a pure-research project, and may not be construed to imply any product commitment by the Digital Equipment Corporation.

References

[1] Spier M J, *A Model Implementation for Protective Domains,* International Journal of Computer & Information Sciences, vol 2 #3, 1973.

[2] Spier M J, Hastings T N, Cutler D N, *An Experimental Implementation of the Kernel/Domain Architecture,* Proc. 4th ACM SOSP, October 1973.

[3] Spooner C R, *A Software Architecture for the 70's: Part I - The General Approach,* Software Practice & Experience, vol 1 #1, 1971.

[4] Spier M J, *Process Communication Prerequisites, or the IPC-Setup Revisited,* Proc. 1973 Sagamore Conference on Parallel Processing, August 1973.

[5] Spier M J, Organick E I, *The Multics Inter-process Communication Facility,* Proc. 2nd ACM SOSP, October 1969.

A NOVEL METHOD OF CONSTRUCTING SORTING NETWORKS

Robert M. Keller
Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08540

Summary

The construction of sorting networks has been a topic of much recent discussion [1] - [5]. In view of the apparent difficulty of verifying whether a reasonably large proposed sorting network actually does sort, the most useful approach for constructing large networks seems to be to devise a recursive scheme which constructs a network which is guaranteed to sort, obviating the verification phase. Examples of this approach are presented in [1],[5]. In this note, another such approach is presented.

The most economical 16-line sorter known has been constructed by Green [3], [4]. His approach is to successively sort lines whose indices differ in one component of the binary expansion. This yields a partial ordering of the lines which is isomorphic to a Boolean "n-cube" configuration. This configuration is then further sorted to yield a linear order. The network for accomplishing this is constructed in a clever, but ad hoc manner, and no techniques for extending this approach to larger numbers of lines have appeared.

In this note such a technique is presented. However, it suffers from the fact that it produces networks which are no more economical than the odd-even merge networks of Batcher [1]. Nevertheless, some insight may result from a knowledge of this technique.

The approach is to reduce an n-cube configuration to an n-m cube in which the vertices represent linear orders of m components. A recursive rule is given which applies this technique to obtain a complete sorting network and the correctness of the rule is proved. It is then shown that the number of comparisons for an n-line network are the same as Batcher's construction, although the networks are definitely not isomorphic to Batcher's. For certain numbers of lines, this method yields networks which are related to Batcher's by a kind of "flipping" operation described in [2]. Precisely what relation holds between these two constructions has not yet been discovered.

A complete presentation of these results appears in [6]. The construction is derived for the more general k-ary n-cube, but upper bounds are only shown for $k = 2$ (the "Boolean" case). Whether other values of k yield better results has not been thoroughly investigated. Proofs of correctness are done in terms of partial orders, using a useful and general lemma about "cross products" of partial orders and the technique of Liu [7].

This work was sponsored by the National Science Foundation through grant GJ-3012, and by the Bell Laboratories, Murray Hill, N.J.

References

[1] K.E. Batcher, "Sorting Networks and Their Applications." AFIPS Conf. Proc., (Spring, 1968), pp. 307-314.

[2] R.W. Floyd and D.E. Knuth, The Bose-Nelson Sorting Problem, Comp. Sci. Dept., Stanford University, Stanford CS-70-177 (November, 1970).

[3] M.W. Green, "Some Improvements in Nonadaptive Sorting Algorithms." Proc. Sixth Annual Princeton Conf. on Information Sciences and Systems, (March, 1972), pp. 387-391.

[4] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, (1973).

[5] D.C. Van Voorhis, Large [g,d] Sorting Networks, Comp. Sci. Dept., Stanford University, Stanford CS-71-239, August, 1971).

[6] R.M. Keller, Another Recursive Technique for the Construction of Sorting Networks, Princeton Univ., Comp. Sci. Lab., Tech. Report TR-126, (June 1973).

[7] C.L. Liu, "Analysis of Sorting Algorithms," Proc. IEEE Twelfth Annl. Symp. on Switching and Automata Theory, October, 1971), pp. 207-215.

HIGH SPEED MULTIPLIER/DIVIDER ITERATIVE ARRAYS

V.C. Hamacher
J. Gavilan
Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada

Abstract -- Various 2-dimensional iterative arrays for the combined parallel implementation of signed binary multiplication and division are presented. Speed and cost comparisons are made with both commercial arithmetic units and recent design and prototype studies. It is shown that combined function arrays can be both speed and cost competitive with separate function arrays.

Introduction

Large, iteratively structured, combinational networks for all four basic arithmetic functions (Add, Subtract, Multiply, and Divide) have become a practical reality in high-speed, general-purpose scientific computers [1],[2] and special purpose applications [3],[4]. Recent design and prototype studies [5],[6],[7] on feasible variations have also been reported.

The parallel processing speed of the subsystem units for each arithmetic function has been enhanced from a system throughput viewpoint by employing both duplicated units and pipelining [1],[2]. On a uniprocessor system, the effectiveness of these latter system designs depends to a large extent on program and instruction mix as well as depth of instruction lookahead.

In most of the references cited above, there is a tendency towards optimizing a large combinational subsystem unit for each arithmetic function. Duplicating or pipelining these separate function units then achieves the desired system speed. A commercial exception is [1] in which a particular unit performs multiply or divide under appropriate conditioning and sequencing. Also, the design studies of [8] and [9] investigated a planar logic array that combines the same two functions.

The purpose of this paper is to present new combined Multiplier/Divider (MD) iterative arrays and analyze their effectiveness as compared to current alternatives. The MD arrays are 2-dimensional and accept two, signed, binary operands in 2's-complement notation along with a binary signal to denote M or D. A double-length product, or quotient and remainder, are generated after a specified delay. The basic approach is to start with a simple (but relatively slow) configuration, called MD1, that is similar in complexity to [8] and [9]. Design changes to increase speed are then incorporated in 3 successive steps that result in the MD4 array that is comparable in speed to the fast individual function arrays of [6] and [7], while at the same time has a cost much less than the sum of the costs of the individual function arrays.

The two basic parameters that are used for comparisons throughout the study are logic delay and cost. Delay is expressed in terms of a normalized value τ that represents delay through a functional level (AND-OR, NAND-NAND, NOR-NOR, etc.) under reasonable fan-in constraints on all gates. Processing rates based on pipelining are covered elsewhere [10]. Two different cost criteria are considered. Gate costs assuming individual gate counting is used, as well as integrated circuit count for reasonable assumptions on MSI level circuits. Both of these methods are justified in terms of currently available integrated circuits.

The Basic Comparison Parameters

Logic Delay

As stated above, all delay expressions will be stated in terms of a normalized value τ that represents delay through a functional level. The choice of a delay unit such as τ is not a straightforward one. Hopefully, the reason for choosing a delay unit in any logic design is to arrive at as simple and as accurate a measure as possible of the delay through an implementation of the design in some particular logic circuit family. This is achieved by substituting a typical value of time (say 12-16 nanoseconds in some TTL technologies) for τ in the delay expression. Now consider where this technique causes problems. In arithmetic arrays, the full adder (FA) function (three inputs, sum (S) and carry (C) outputs) and the exclusive-or (EX-OR) function usually account for a large part of the logic design components. If we assign τ on a functional level, as indicated above, all three outputs, S, C, and EX-OR, will occur with delay τ after inputs are available. (It should be noted that we ignore any input inversions needed in both delay and cost computations.) But, in many TTL integrated circuits, the delays in producing these three functions can be appreciably different. For instance, EX-OR might be 1·2 times the delay of a single NAND gate, and C is typically substantially faster than S. This presents a fundamental problem in attempting a general delay measure that is useful in comparing various logic designs to gauge their implementation effectiveness. Our compromise is to use delay expressions involving τ as defined above. We then claim that, although they might not be accurate enough to compute absolute delays achievable in implementing various designs (based

on some average τ for a certain logic circuit family), they suffice for our purposes of getting some quantitative figure of merit for comparisons. In fact, we depend on the S, C, and EX-OR type of discrepancies being averaged out along the longest delay path in the various designs.

Logic Cost

The technology used (wired-or capability, etc.) and level of integration (SSI, MSI, etc.) assumed complicate the definition of a suitable logic cost measure probably to a greater degree than they affect the adoption of a simple delay measure, as discussed above. In this paper, we will base logic cost on one of two distinct measures. The simplest and most often used measure will be total gate count. Since we are discussing relatively large combinational arrays of logic circuits, where fan-in ranges normally from 2 through 4, we will not explicitly include inputs in our gate cost measure. Implicitly, of course, the basic gate cost unit, g, can be taken to mean the cost of some "average" gate which "on the average" might have 3 inputs. Another cost measure that we will use in one instance is that of integrated circuit count under some reasonable current technology complexity level. This technique will be given in more detail later when it is applied.

Other Possible Parameters

Other design parameters that might illuminate the comparative merits among various logic designs are possible. Interconnection crossover complexity, array cell regularity, standard function utilization are among these. We will not work out the details on any other than delay and cost as defined above; however, we present logic diagrams, for all four arrays discussed, in enough detail that anyone can derive particular figures of merit that might be of interest.

Four Multiplier/Divider (MD) Arrays

The most familiar binary multiplication algorithm is to shift the multiplicand (B) left once for each multiplier (R) bit position, after adding B into an accumulating partial product (A) if the corresponding R bit is 1, until A finally becomes the product P = B·R when all multiplier bits, low order to high order, have been used. This scheme has been stated for positive operands; but, by modification due to Booth [11], it can be made to work for signed operands in 2's complement representation, yielding P directly in the correct 2's complement form. Subtraction of the multiplicand, as well as addition, is possible. Each operation decision is the result of inspecting the appropriate multiplier bit and its right-hand neighbour at each step.

One of the standard division algorithms is the non-restoring algorithm operating on a dividend A with a divisor B to generate a quotient Q. The altered dividend is referred to as the partial remainder at each step. An operation cycle is as follows: The sign of A is inspected. If it is positive, B is subtracted from A and if it is negative, B is added to A. The quotient bit generated is the complement of the sign bit of the new partial remainder. The divisor is shifted one binary position right after each cycle. Many authors have discussed this scheme; see, for example, Guild [12]. This algorithm can also be modified to operate on signed operands; however, the quotient generated is correct if it is positive; but it is in 1's complement if it is negative, so a one must be added later to convert it to 2's complement notation. Separate planar arrays of cells, each usually containing a full adder with controlled inputs, can be constructed fairly directly from these or similar algorithms. For instance see, Majithia and Kitai [13], Bandyopadhyay, et al [14], Deegan [15], and Hoffman, et al [16] for array implementations of multiplication based on variations of the above basic scheme. Division array implementations based on variations of the above discussion appear in Guild [12], Dean [17], Gardiner [18], and Gardiner and Hont [19].

MD1 Array

When we attempt to combine the separate arrays, the only sensible arrangement seems to be to associate the B vector (multiplicand or divisor) positions with each other and the A vector (partial product or dividend and partial remainder) positions with each other, moving downward through the rows of the array. That is why we have combined their names. The multiplier, R, and quotient, Q, are positioned at the left column edge of the array. There is one complication. In multiplication, B is shifted left with respect to A; but in division, B is shifted right with respect to A. The solution is to shift B right with respect to A in multiplication, and inspect and use the multiplier bits high order to low order instead of in the other direction as above. This is the scheme developed by Majithia and Kitai [13]. The arrays can then be combined as MD1 in Figure 1. In general, the B, Q and R vectors are n bits long, including the sign bit in the case of B and R. The A vectors are 2n-1 bits long, including the sign bit. It is convenient to consider the operands in fraction form, with A and B normalized in the case of division. We then have, (where all coefficients = 0 or 1):

$$B \text{ (Multiplicand or Divisor)} = B_0 \cdot B_1 \ldots B_{n-1} = -B_0 2^0 + B_1 2^{-1} + \ldots + B_{n-1} 2^{-(n-1)}$$

$$R \text{ (Multiplier)} = R_0 \cdot R_1 \ldots R_{n-1} = -R_0 2^0 + R_1 2^{-1} + \ldots + R_{n-1} 2^{-(n-1)}$$

$$Q \text{ (Quotient)} = Q_0 \cdot Q_1 \ldots Q_{n-1} = Q_0 2^0 + Q_1 2^{-1} + \ldots + Q_{n-1} 2^{-(n-1)}$$

$$A \text{ (Partial Product or Remainder)} = A_0 \cdot A_1 \ldots A_{2n-1} = -A_0 2^0 + A_1 2^{-1} + \ldots + A_{2n-1} 2^{-(2n-1)}$$

In the case of division, Q_0 is not the sign bit, but is a significant bit of the answer. This is because $1/2 \le Q \le 2$ for $1/2 \le A,D \le 1$. The sign bit for Q is thus not indicated in our arrays. The output of the M function in cell 1 of MD1 (Figure 1(b)) is given by $M = D_1\overline{D_2}\overline{B_i} + \overline{D_1}D_2B_i$. The "function" signal, F, is set to 0 for multiplication and 1 for division; cell 2 (Figure 1(c)) then routes the multiplier bit pairs R_k, R_{k+1} for Booth algorithm control in multiplication, or routes the sign bit control for division. Note that all R_i bits must be set to 0 when division is being performed. Thus, cell 2 acts as a control column of cells, and the M function in cell 1 uses the control signals to appropriately apply the correct version of the B vector to the A vector. The cost and delay expressions are:

$$\text{MD1 cost} = (18n^2 + 2n)g \tag{1a}$$

$$\text{MD1 mult. delay} = (2n + 1)\tau \tag{1b}$$

$$\text{MD1 div. delay} = (n^2 + 2n)\tau \tag{1c}$$

The combined multiplier divider arrays of Gex [8], and Gardiner and Hont [9] are similar in complexity of design and have about the same cost and delay properties.

MD2 Array

Our procedure is now going to be to introduce substantial design changes in three successive steps starting from MD1. They are substantial in that the basic algorithms for carrying out the arithmetic operations are altered significantly. In MD2, the partial product/remainder vector A is not developed explicitly at each row level but is represented by two binary vectors S and C, which, if added would produce the correct vector A at that row level. This is the familiar carry-save reduction technique that was originally introduced by Wallace [20] in a 3-dimensional multiplier logic design. The two vectors S and C are the result of a 3-to-2 carry-save reduction on the previous row's S and C vectors and the proper version of the B vector. In the case of multiplication, this necessitates a length 2n-1 fast adder operating on the S and C outputs of the last row to produce the product P. This is indicated in Figure 2(a). Since the division process requires the sign of A to determine the subsequent row operation, this must be determined by a carry lookahead network L at the leftend of each row. It operates on generate and propagate functions formed in the type 3 cells. These G_i and P_i functions are formed from the S and C vector outputs of each row. An examination of the non-restoring division algorithm reveals that the carry-out from the sign bit position directly yields the quotient bit, so this is the way it is done in Figure 2. This observation actually constitutes a suggested improvement to the design in [6]. $\overline{Q_i}$ is then fed to the control cell 5 of the next row. The L cell must be redesigned for each operand length, and if fan-in is constrained to equal to or less than eight, then a two-level (2τ) lookahead scheme must be employed for $n \ge 10$. This is reflected in the cost and delay figures shown below. The cost and delay expressions are:

$$\text{MD2 cost} = \begin{cases} (21n^2 - n)g & \text{for } n < 10 \\ (21n^2 + 2n\sqrt{n})g & \text{for } 10 \le n \le 64 \end{cases} \tag{2a}$$

$$\text{MD2 mult. delay} = (n + 2)\tau \tag{2b}$$

$$\text{MD2 div. delay} = \begin{cases} (6n)\tau & \text{for } n < 10 \\ (7n)\tau & \text{for } 10 \le n \le 64 \end{cases} \tag{2c}$$

The cost increase from MD1 to MD2 is small compared to the speed gain, especially in the case of division, which has been made essentially linear in n over practical operand ranges. This form of array division algorithm is due to Cappa and Hamacher [6] and the carry-save technique (along with multiplier bit grouping) has been used by Ramamoorthy and Economides [7] in a high-speed planar multiplier array. It is to be noted that the cost and delay of the Fast Adder has not been included in the above expressions. It can be designed (with carry lookahead techniques) so that it does not change any of the expressions by more than about 20%. To our knowledge, MD2 and the next two arrays have not appeared in the literature.

MD3 Array

The next change to make is to decode the multiplier bits in pairs and generate two quotient bits at a time. Although this increases the complexity of each row of cells in the array, the number of rows is reduced by a factor of two. A net cost saving then results. We get MD3, as shown in Figure 3, by making these two changes to the MD1 structure. When the MD2 techniques of carry-save reduction and carry-lookahead are also incorporated into MD3 we will finally have evolved to MD4 which is in the next subsection. The multiplier bit grouping technique is well known and has been used by Wallace [20] and Ramamoorthy and Economides [7] in their arrays so it will not be detailed here. The technique for generating two quotient bits at a time is somewhat more complex but has also been adequately described in detail by Flores [21]. It necessitates having 3/2 the divisor available as an input vector. We assume that this is formed before division is begun and is presented as one of the inputs. The r, s, t bundle of inputs into cell 6 (Figure 3(b)) is really a bit position of 1/2 B, B, and 3/2 B in the case of division; and in the case of multiplication, r and s represent one bit position of 1/2 B and B, respectively, with t not being used. The control signals T_1 and T_2, which are outputs from control cell 7 (Figure 3(d)) are used to select appropriately among r, s, t or their complements. This selection is done in the E logic (Figure 3(c)) of cell 6 (Figure 3(b)). The D signal decides complementation or not. An inspection of the wiring of cell 6 should convince the reader that the 2-place shift per row of B is performed correctly. The signals T_1, T_2 and D are determined by a multiplier bit pair and its adjacent bit neighbour on the right in the case of

multiplication; and by the leading three bits of A and bits B_0 and B_2 in the case of division. This is all accomplished in control cell 7 (Figure 3(d)) along with the generation of two quotient bits in the case of division. The cost and delay expressions are:

$$\text{MD3 cost} = (13n^2 + 37n + 25)g \quad (3a)$$

$$\text{MD3 mult. delay} = (2n + 3)\tau \quad (3b)$$

$$\text{MD3 div. delay} = (n^2/2 + 2n + 3)\tau \quad (3c)$$

There are actually small variations in these expressions depending on whether n is even or odd, but in each situation we have given the worst case. Compared to MD1, in MD3 the cost is appreciably lower, multiplication time is about the same, and division time has been halved. MD3 is slower than MD2, but costs less. The final evolution to MD4, which incorporates the MD2 techniques of carry-save reduction and carry-lookahead will prove to be the best design on all counts. It should again be noted before we leave this section that the time and cost involved in generating 3/2 B has been neglected. For practical values of n, this is a reasonable assumption.

MD4 Array

If the carry save and carry lookahead techniques described in the MD2 subsection are applied to the MD3 structure, we obtain the MD4 array shown in Figure 4. Since the control cell 7 is Figure 3(d), no further discussion of it is needed. Also, the E function in the main body cells 8 and 9 (Figures 4(b) and (c)) is the same as in Figure 3(c). The remainder of cells 8 and 9 is much the same as in cell 6 (Figure 3(b)) of MD3, the differences being that S and C vectors are produced to represent A, and P and G functions are included to provide inputs to the lookahead computation. The L cell of Figure 4(a) is similar to the L cell of Figure 2 and is used in MD4 to produce the carry-in to the A_2 position. This carry and the S and C vector bits for partial remainder positions A_0, A_1, and A_2 are inputs to the CL cell (Figure 4(d)). The CL cell computes A_0, A_1, and A_2 which are needed in the control cell 7 for the division process. The cost and delay expressions are:

$$\text{MD4 cost} = \begin{cases} (15n^2 + 47n + 33)g & \text{for } 7 \le n \le 13 \\ (15n^2 + 47n + 33 + (n+1)\sqrt{n-4}/2)g & \text{for } 14 \le n \le 68 \end{cases} \quad (4a)$$

$$\text{MD4 mult. delay} = (n/2 + 4)\tau \quad (4b)$$

$$\text{MD4 div. delay} = \begin{cases} (3n + 5)\tau & \text{for } 7 \le n \le 13 \\ (4n + 6)\tau & \text{for } 14 \le n \le 68 \end{cases} \quad (4c)$$

Again, as in MD2, the Full Adder has been omitted from both cost and delay expressions, as well as the formation of 3/2 B.

Comparisons

If we substitute the practical range of values n = 8, 16, 32, and 64 into equation sets (1), (2), (3), and (4), we obtain Table I, which allows convenient comparisons among the MD arrays. It is easy to conclude that MD4 is the best design from the cost/delay effectiveness standpoint. The rest of this section will be devoted to comparing MD4 to members of two classes of multipliers and dividers.

Other Logic and Prototype Designs

In this subsection, MD4 is compared to two high-speed planar separate function arrays that have been reported. The multiplier array (RE) of Ramamoorthy and Economides [7], that uses bit grouping of the multiplier and carry-save reduction as in MD4, has approximate cost and delay expressions as follows:

$$\text{RE array cost} = (10n^2 + 8n + 26)g \quad (5a)$$

$$\text{RE array mult. delay} = (n/2 + 2)\tau \quad (5b)$$

The division array (CH) of Cappa and Hamacher [6] that uses carry-save reduction and carry lookahead but generates only one quotient bit per row as in MD2, has approximate cost and delay expressions as follows:

$$\text{CH array cost} = \begin{cases} (17n^2 + 10n)g & \text{for } n < 10 \\ (17n^2 + 11n + 2n\sqrt{n})g & \text{for } 10 \le n \le 65 \end{cases} \quad (6a)$$

$$\text{CH array div. delay} = \begin{cases} (4n)\tau & \text{for } n < 10 \\ (5n)\tau & \text{for } 10 \le n \le 65 \end{cases} \quad (6b)$$

Table II allows a concise comparison of the RE, CH, and MD4 arrays.

Commercial Structures

The Advanced Micro Devices (AMD) Co. [22] produces a 2 bit x 4 bit 24-pin MSI multiplier chip (the AM2505) and a 4 bit 24-pin MSI adder chip (the AM9340) that can be used as the basic cells in a multiplication array. They use bit grouping of the multiplier, do not use carry-save reduction, but use a carry lookahead scheme for fast propagation of the carries along each row of AM2505's. The AM9340's are used in parallel to accumulate a set of partial products into the full product. At an operand length of n = 16, the delay is approximately 30τ as compared to about 17τ for MD4. The AMD array delay value is derived by evaluating the logic equivalent of their chip in a manner consistent with our evaluation of the MD, RE, and CH arrays. Since the AMD array generates the product P, we have included in the 17τ MD4 delay a plausible amount (5τ) for the 32-bit lookahead adder needed in conjunction with the basic MD4 structure.

There are 32 AM2505 chips and 16 AM9340 chips needed at n = 16. Now if we consider that 4 of the main body cells (8 and 9) in the MD4 are

implemented in a single 40-pin MSI chip, the MD4 would require about 40 of these chips plus the final full adder (8 AM9340 type of chips) and the left column of control logic (cells 7, CL and L). If we estimate this control logic at about the equivalent of 24 MSI chips, the total MD4 array has an MSI chip count of about 72, so that it would be about 50% more expensive. It is also instructive to estimate the equivalent gate count in the AMD array as compared to a gate count for the MD4 which can be derived from expression (4a) above plus a reasonable full adder gate count. If we do this for the n = 16 case, we get an approximate equivalent gate count of 4,400 for the AMD array and 5,300 for the MD4.

The 56-bit floating point fraction multiplier and divider circuitry in the IBM S360/91 [1] computer has equivalent logic delays of approximately 36τ and 110τ, respectively. The comparable figures for MD4 (including the Fast Adder) are 38τ and 185τ. It should be noted that this commercial unit performs division by the iterative multiplication technique which is completely different from the MD4 technique, but makes very effective use of the multiplication structure. Detailed cost comparisons of this unit with MD4 will not be attempted.

Acknowledgement

This research was partly supported by Grant A 5192 from the National Research Council of Canada.

References

[1] S.F. Anderson, et al, "The IBM System 360/91: Floating Point Execution Unit," IBM Journal of Research and Development, (January, (1967), pp. 34-53.

[2] P. Bonseigneur, "Description of the 7600 Computer System," Computer Group News, Vol. 2, No. 9, (May, 1969), pp. 11-15.

[3] S.D. Perazis, "A 40-ns 17-bit by 17-bit Array Multiplier," IEEE Trans. Comp. (Short Notes), Vol. C-20, (April, 1971), pp. 442-447.

[4] A. Habibi, and P.A. Wintz, "Fast Multipliers," IEEE Trans. Comp., Vol. C-19, (February, 1970), pp. 153-157.

[5] R. Stefanelli, "A Suggestion for a high-speed parallel binary divider," IEEE Trans.Comp., Vol. C-21, (January, 1972), pp. 42-55.

[6] M. Cappa, and V.C. Hamacher, "An Augmented Iterative Array for High-Speed Binary Division," IEEE Trans. on Comp., Vol. C-22, No. 2, (February, 1973), pp. 172-175.

[7] C.V. Ramamoorthy, and S.C. Economides, "Fast Multiplication Cellular Arrays for LSI Implementation," 1969 FJCC, AFIPS Press, (Montvale, N.J.), pp. 89-98.

[8] A. Gex, "Multiplier-Divider Cellular Array," Elec. Lett., Vol. 7, No. 15, 29 July 1971.

[9] A.B. Gardiner, and J. Hont, "Cellular-Array Arithmetic Unit with Multiplication and Division," Proc. IEE, Vol. 119, (June, 1972), pp. 659-660.

[10] T.G. Hallin, and M.J. Flynn, "Pipelining of Arithmetic Functions," IEEE Trans. on Comp. (Short Notes), Vol. C-21, No. 8, (August, 1972), pp. 880-885.

[11] A.D. Booth, "A Signed Binary Multiplication Technique," Quart. J. Mech. Appl. Math., Vol. 4, Pt. 2, (1951), pp. 236-240.

[12] H.H. Guild, "Some Cellular Logic Arrays for Non-restoring Binary Division," The Radio and Electronic Engineer, Vol. 39, (1970), pp. 345-348.

[13] J.C. Majithia, and R. Kitai, "An Iterative Array for Multiplication of Signed Binary Numbers," IEEE Trans. on Comp., (February, 1971), pp. 214-216.

[14] S. Bandyopadhyay, et al, "An Iterative Array for Multiplication of Signed Binary Numbers," IEEE Trans. on Comp., Vol. C-21, No. 8, pp. 921-922.

[15] I.D. Deegan, "Cellular Multiplier for Signed Binary Numbers," Elec. Lett., Vol. 7, No. 15, 29 July 1971.

[16] Hoffman, et al, "Multiplieur parallele a circuits logiques iteratifs," Elec. Lett., Vol. 4, (1968), p. 178.

[17] K.J. Dean, "Binary Division Using a Data Dependent Iterative Array," Elec. Lett., Vol. 4, No. 14 (July, 1968), pp. 283-284.

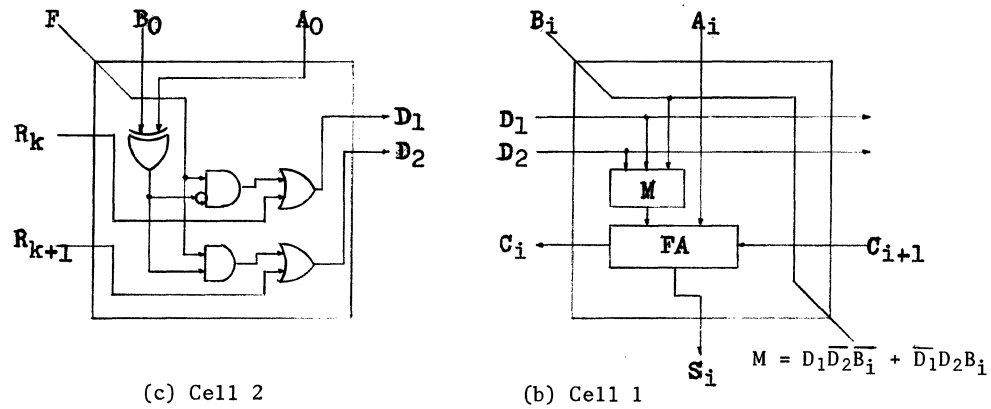
[18] A.B. Gardiner, "Fast Economical Binary Divider," Elec. Lett., Vol. 7, No. 23, (Nov., 1971).

[19] A.B. Gardiner, and J. Hont,"Comparison of Restoring and Non-restoring Cellular-Array Dividers," Elec. Lett., Vol. 7, No. 8, (April, 1971), pp. 172-173.

[20] C.S. Wallace, "A Suggestion for a Fast Multiplier," IEEE Trans. on Elec. Comp.," (February, 1964), pp. 14-17.

[21] I. Flores, "The Logic of Computer Arithmetic Prentice-Hall, Englewood Cliffs, N.J.(1963).

[22] Advanced Micro. Devices Co., Application Notes for the AM9340 (April, 1972) and the AM2505 (November, 1971).



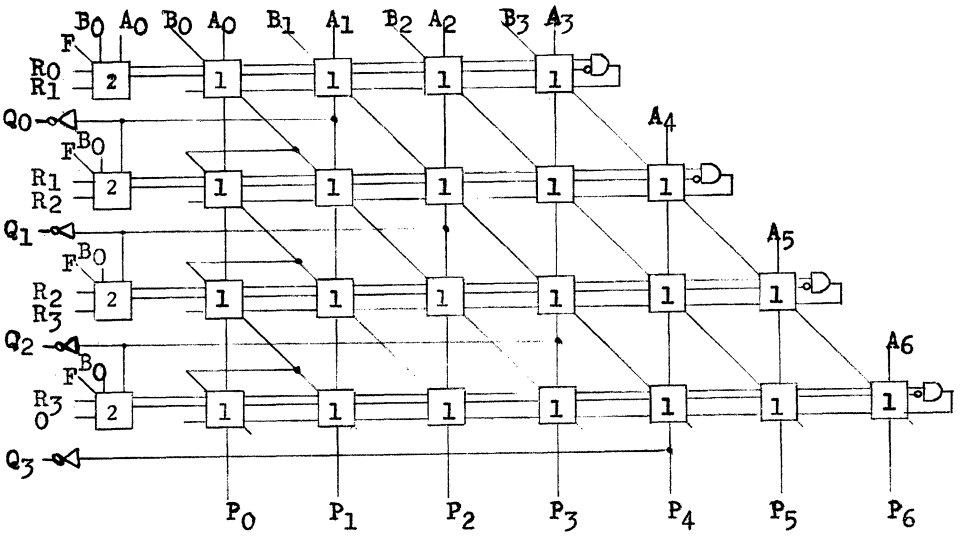


Figure 1. MD1 - Multiplier/Divider iterative array using Booth and Non-restoring algorithms, for n=4.

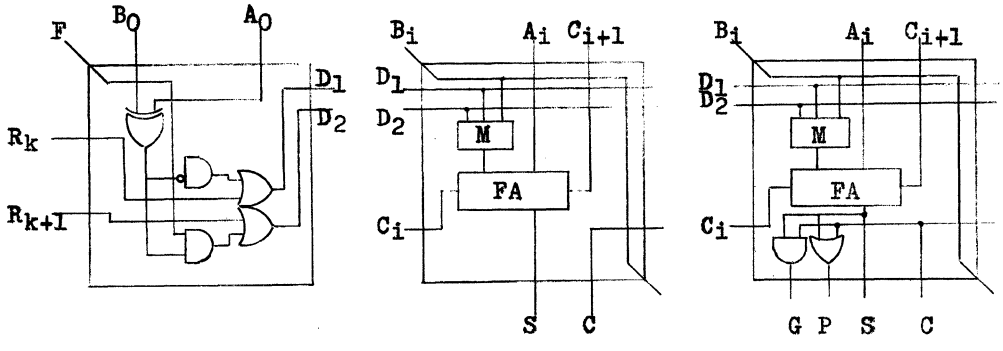
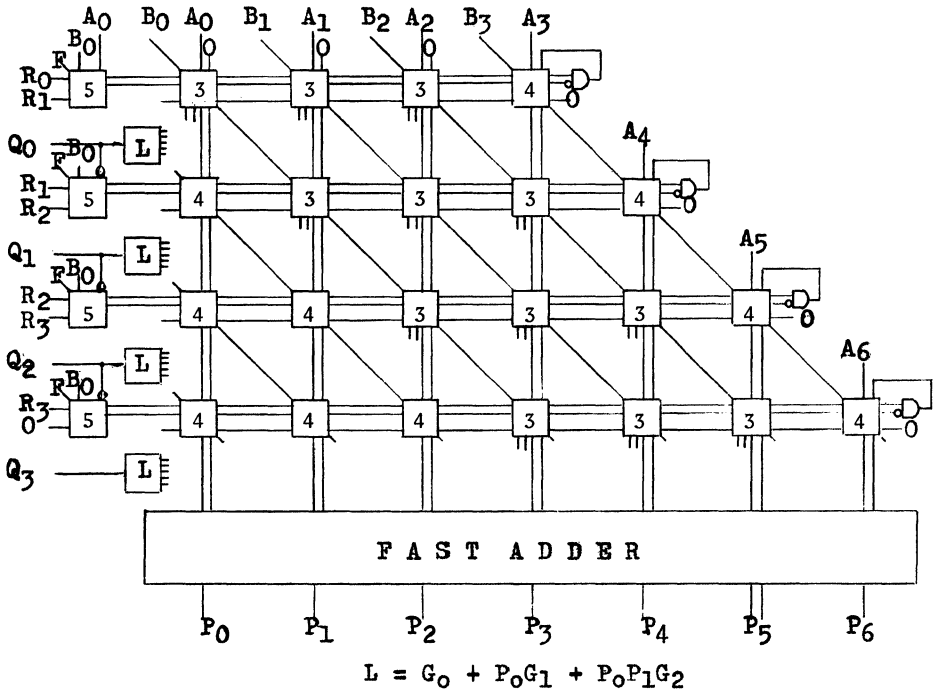
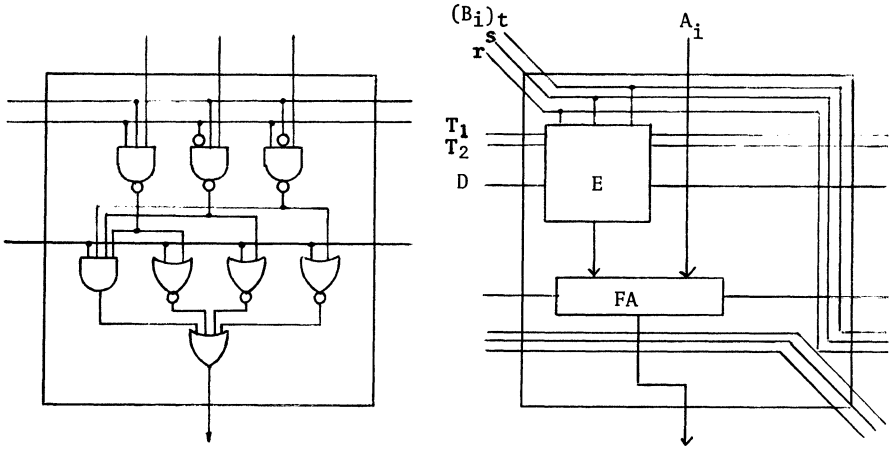


Figure 2. MD2 - Multiplier/Divider array using Booth and Non-restoring algorithms and carry-save along with carry lookahead, for n=4.



(a) The array

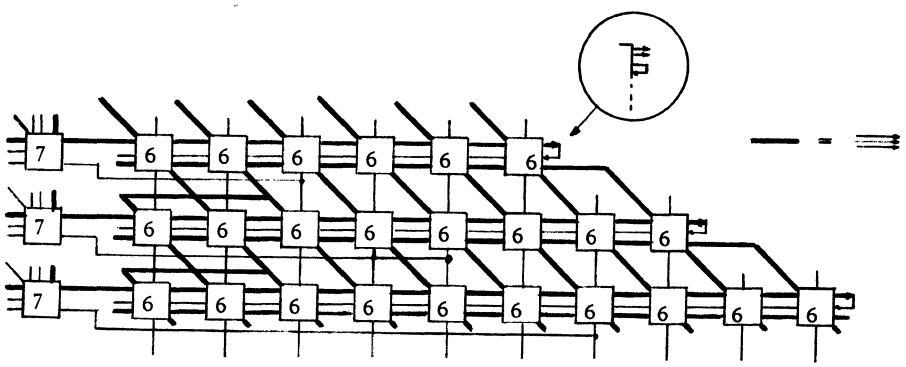
Figure 2 (cont'd.). MD2 - Multiplier/Divider array using Booth and Non-restoring algorithms and Carry-save along with Carry lookahead, for n=4.



(c) E function

(b) Cell 6

Figure 3. MD3 - Multipler/Divider array using multiplier bit pairing and 2-bit quotient generation, for n=5.



(a) The array

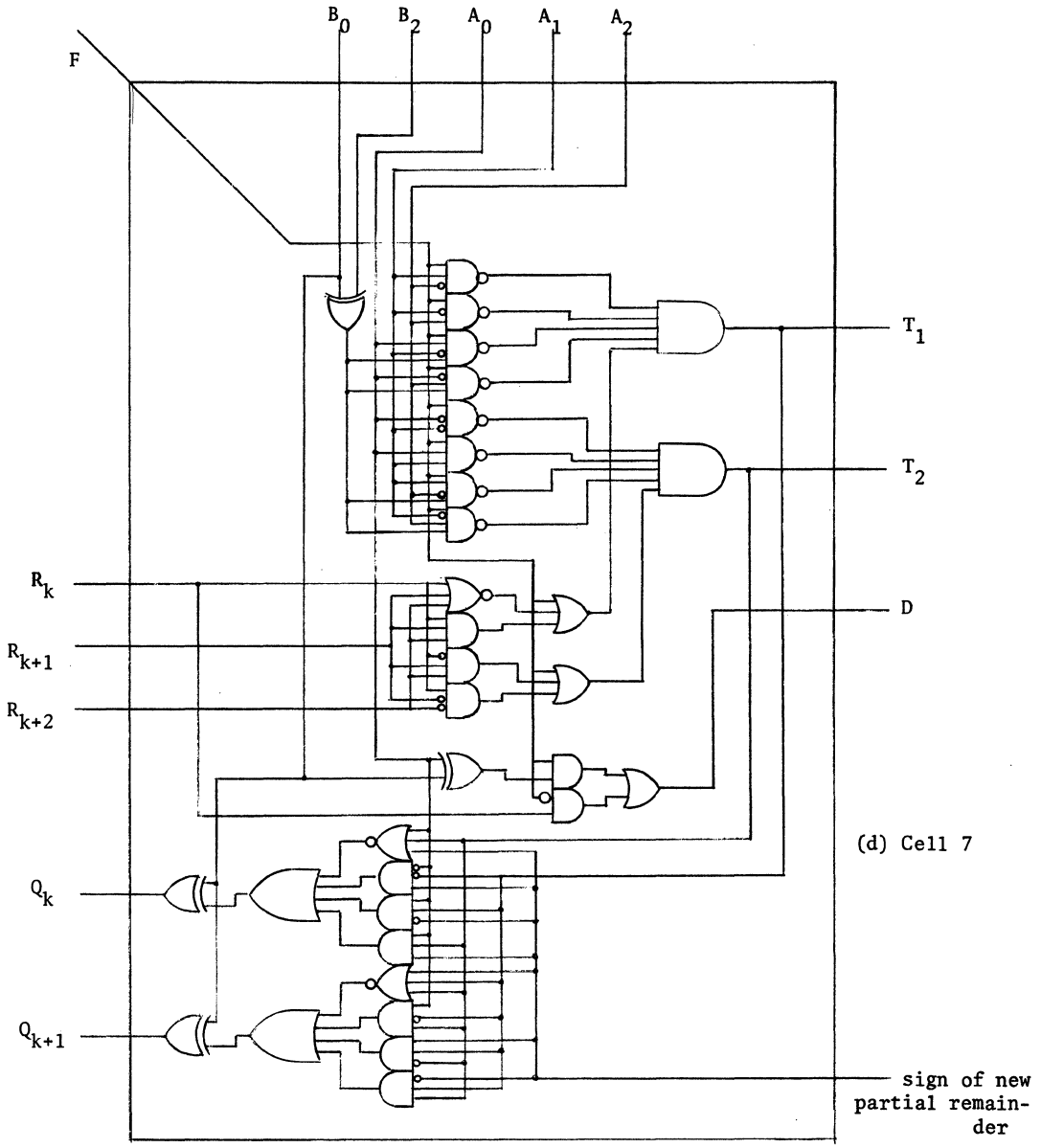
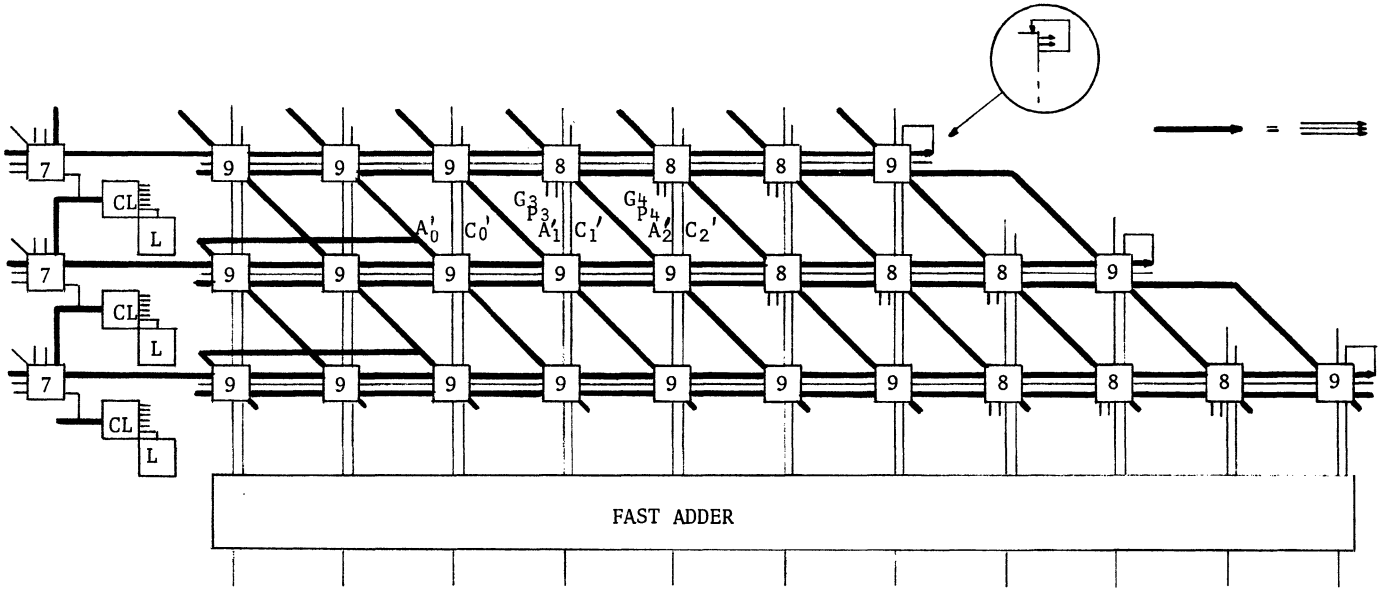
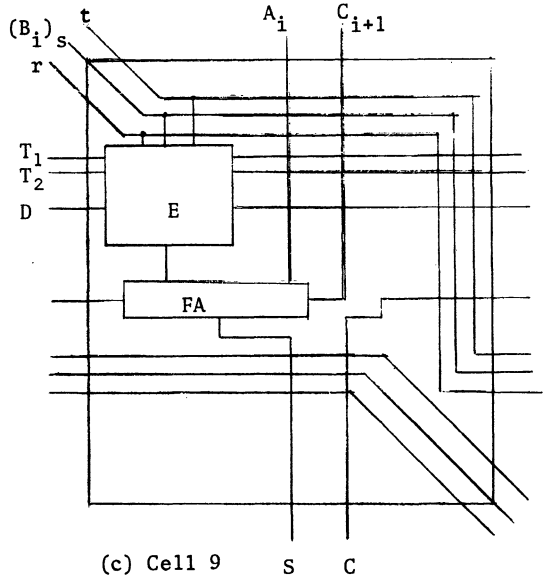


Figure 3 (cont'd.). MD3 - Multiplier/Divider array using multiplier bit pair grouping and 2-bit quotient generation, for n=5.

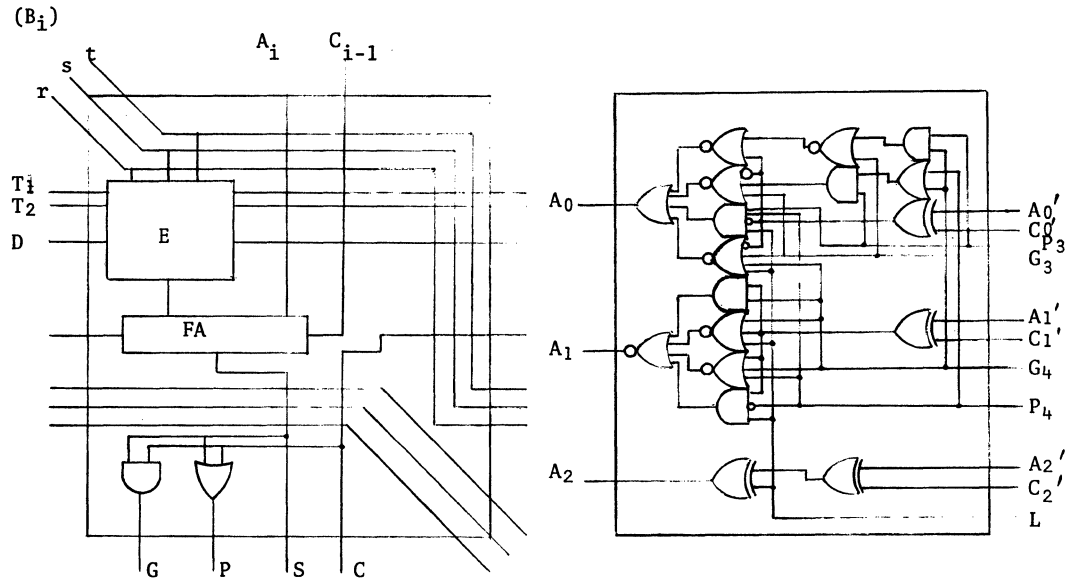


(a) The array



(c) Cell 9

Figure 4. MD4 - Multiplier/Divider array using multiplier bit pairing and 2-bit quotient generation with carry-save and carry lookahead technique, for n=6.



(b) Cell 8 (d) Cell CL

Figure 4 (cont'd.). MD4 - Multiplier/Divider array using multiplier bit pairing and 2-bit quotient generation with carry-save and carry lookahead technique, for n=6.

Table I: Cost and Delay Comparisons Among the MD Arrays

n	Cost g				Mult. Delay τ				Div. Delay τ			
	MD1	MD2	MD3	MD4	MD1	MD2	MD3	MD4	MD1	MD2	MD3	MD4
8	1,168	1,336	1,153	1,369	17	10	19	8	80	48	51	29
16	4,640	5,504	3,945	4,659	33	18	35	12	288	112	163	70
32	18,296	21,888	14,521	16,996	65	34	67	20	1,088	224	579	134
64	73,856	87,040	55,641	64,741	129	66	131	36	4,224	448	2,179	262

Table II: Cost and Delay Comparisons Among the RE, CH, and MD4 Arrays

n	Cost g			Mult.delay τ		Div.delay τ	
	RE	CH	MD4	RE	MD4	CH	MD4
8	730	1,168	1,369	6	8	32	29
16	2,714	4,656	4,659	10	12	80	70
32	10,522	18,144	16,996	18	20	160	134
64	41,498	71,360	64,741	34	36	320	262

A VERSATILE DATA MANIPULATOR

Tse-yun Feng
Department of Electrical and Computer Engineering
Syracuse University
Syracuse, N. Y. 13210

Summary

The main deviation of a parallel processor organization from a conventional (sequential) one can be seen to be in the data manipulating functions which are defined to be the functions required for preparing appropriate operands for fetching, execution, and storing [1]. Thus, data manipulating functions involve unary operations and they can be classified in the following categories: permuting, replicating, spacing, masking, and complementing.

The structure of a versatile data manipulator [2] is shown in Fig. 1[(a)].

The basic circuit has an N-by-N array construction (or N^2 cells). Each cell consists of four gates. The circuit can easily be partitioned. Thus, implementation of this circuit requires only one circuit type. At present state-of-the-art up to 8x8 cells and their decoders may be implemented on one chip.

This data manipulator is capable of achieving all the data manipulating functions mentioned above. Furthermore, it can achieve not only these functions for 2's-power data sets (or strings) and replications, but non-2's-power functions as well. Such a data manipulator is particularly attractive in applications requiring extensive spacing functions. Thus operations such as counting, multiple additions, bubbling process [3], can all be easily achieved. It is also evident that the system availability or self-repairability can be improved or provided by applications of the spacing functions.

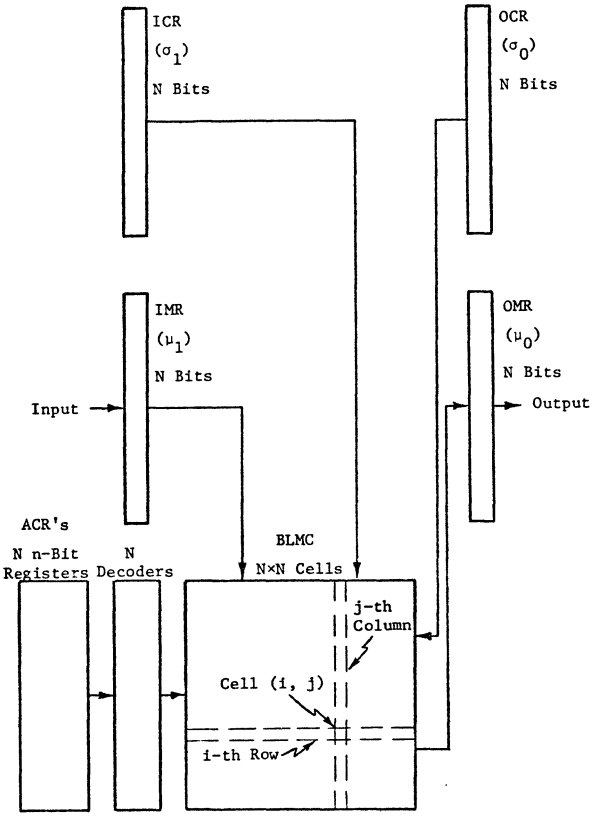
References

[1] T. Feng, Parallel Processor Characteristics and Implementation of Data Manipulating Functions, Tech. Report TR-73-1, Department of Electrical and Computer Engineering, Syracuse University (April 1973), Tech.Report RADC-TR-73-189 (July 1973), 74 pp.

[2] T. Feng, The Design of a Versatile Line Manipulator, Tech. Report TR-73-5, Department of Electrical and Computer Engineering, Syracuse University (June 1973), 85 pp.

[3] B. H. McCormick, "The Illinois Pattern Recognition Computer - ILLIAC III," IEEE Trans. on EC (December 1963), pp. 701-813.

(a) It is noted that the complementing and comparison circuits which may be located at either the input or the output side of the structure are omitted from Fig. 1 for clarity.



ACR Address Control Register
BLMC Basic Line-manipulator Circuit
ICR Input Control Register (σ_1)
IMR Input Mask Register (μ_1)
OCR Output Control Register (σ_0)
OMR Output Mask Register (μ_0)

Fig. 1 The Structure of a Versatile Data Manipulator

AN ARRAY OF COMPUTING MEMORY CELLS

E. Della Torre and Jorge Roitman
Department of Electrical Engineering,
McMaster University,
Hamilton, Ontario, Canada

Summary

A memory cell has been designed and constructed as an element of a highly parallel general purpose computer of a modified SOLOMON structure [1]-[2]. This cell has been interfaced to a PDP-11/20 computer and can perform array operations under central processor asynchronous control. The cell size has been minimized leaving, however, the capability of computing certain transcendental functions and performing iterative calculations in either the integer or the floating point modes.

The basic SOLOMON communication structure has been extended to include a ROW/COLUMN vector of cells. Each cell of the vector can communicate with all the cells of the corresponding row and column. With such a structure, array operations, such as the matrix transposition and a solution of Laplace's equation by the Jacobi type methods or the SOR methods, can be achieved efficiently. The cells normally operate in unison under the central processor control. Each cell has, however, the capabilities of performing individual operations under certain conditions. The array can be microprogrammed to perform iterative computations independent of the central processor until certain convergence condition has been achieved.

Each cell is a triple-address machine consisting of 15 words and arithmetic hardware. It operates between words or bytes of its own memory, or between one of its words or bytes and a word or a byte of another cell it can communicate with. The number of words was chosen so that various algorithms for computing transcendental functions can be implemented within a cell. This organization permits the simultaneous computation of certain functions for sets of argument values.

The addressing system has been designed to permit selecting a particular cell, a row, a column, the even rows, the odd rows, or all the cells of the array. In addition, three modes of cell addressing are available: direct, concatenated, and automatic allowing a very efficient way of the cell selection. The system operates by inhibiting all but the addressed cells.

Associative-memory capabilities [3] can be easily incorporated to the system. The existing inhibit hardware can be used for detecting the cells in which certain specified conditions are satisfied. The addresses of those cells can be read out sequentially by incorporating a cell priority detection address system.

The cell has been satisfactorily tested by using several algorithms. A multi-cell system has been simulated on a PDP-11/20 computer. A compiler has been written for the PDP-11/20 to translate the user mnemonic language into the appropriate contents of the 32-bit instruction register. Several standard subroutines have been provided for integer division, floating point arithmetic, and computation of transcendental functions.

References

[1] D.L. Slotnick, W.C. Bork, and R.C.McReynolds, "The SOLOMON Computer", Proc. AFIPS, Fall Joint Comp. Conference, 1962, pp. 97-107.

[2] E. Della Torre and F. Ho, "Implementation of a Cellular Computing Memory Array", Proc. of the Symposium of Computers and Automata, Brooklyn, N.Y., April 1971, pp. 625-634.

[3] C.Y. Lee and M.C. Paull, "A Content Addressable Distributed Logic Memory with Application to Information Retrieval", Proc. IEEE, June 1963, pp. 924-932.

AN EFFICIENT ASSOCIATIVE PROCESSOR USING BULK STORAGE

Hubert H. Love, Jr.
Equipment Engineering Divisions
Hughes Aircraft Company
Los Angeles, CA 90005

Abstract -- A hybrid associative processing system using an MOS shift-register bulk memory is described, together with its application to large-scale fact-retrieval applications. The system fulfills several criteria for balanced, efficient design of highly-parallel machines. A comparison with similar machines using rotating memories is made.

Introduction

The processor organization to be described here is an outgrowth of the Association-Storing Processor (ASP) project.(a) The object of the project effort was the development of processor organizations biased toward nonarithmetic applications. As a first step in this direction, a representative application, namely fact retrieval (i.e., question-answering), was chosen. The next step in the project was the development of a language, the ASP language, which expresses the data organizations and the processes of concern in the application. Following this, three processor organizations were designed, using the language as a guide. The organization described here is related to the third of these [3], and is an attempt to achieve efficient operation of an associative memory when the data base resides in a large, inexpensive bulk memory.

Speed, Cost and Balance

The justification for the associative/bulk memory combination lies in the desire to simultaneously achieve higher processing speed and throughput, lower cost and a balanced, efficient system. Processing speed has been a particular problem in such sophisticated fact-retrieval applications as military strategic command and control and the translation or interpretation of natural languages. This is because such applications involve very large data bases (at least the order of 10^9 bits), and because very often (such as when deductive inference is used in the retrieval process) many retrieval operations must be performed and many records processed in order to answer a single query. The ability of associative memories to search and process data in a highly-parallel fashion makes these devices natural candidates for consideration.

The large size of the data bases used in the applications of interest is the principal cost consideration in the processor design, and is the justification for the use of an inexpensive bulk memory as the primary data storage medium. It is particularly important in this respect that the ratio of associative memory to bulk memory size be small, and at it not increase as the size of the data base increases.

System balance and efficiency are closely related terms. A balanced system, as defined here, is a system in which no major part of the system normally waits for another part to complete its task. Balance is particularly important with respect to the associative memory in the system to be described and, to a lesser extent, with respect to the bulk memory. A system is said to be efficient if all principal subsystems are performing a non-trivial task all or nearly all of the time during normal operation. Both balance and efficiency directly affect cost and performance in any computer organization, and they are the keys to the design of the one to be described here.

The system concept is developed around a hybrid associative-memory/mass-memory hardware organization, a data structure and a processing strategy. These three ideas shall be described in that order.

System Organization

The general organization of the system is shown in Figure 1. The principal components are a set of associative memories and a bulk memory consisting of static MOS shift registers.

The associative memories are conventional in organization and bit-serial in operation. Each word contains a 64-bit static shift register for the storage of data. Each associative memory is capable of the following operations.

1. A simultaneous comparison of the contents of every word in the memory with the contents of an external register, called the compare register. A flip-flop, called the match flip-flop, is set at each word satisfying the comparison. The operation is field-selective, with the fields being defined by the contents of another external register.
2. An ordered serial retrieval or loading of those words having their match flip-flops set.
3. A field-selective mass-write operation, in which the contents of an external register are written into the selected fields of every word having its match flip-flop set.
4. The transfer of the states of the match flip-flops to the inputs of the data storage registers for the corresponding words, and vice-versa.

(a) See references [1] through [3].

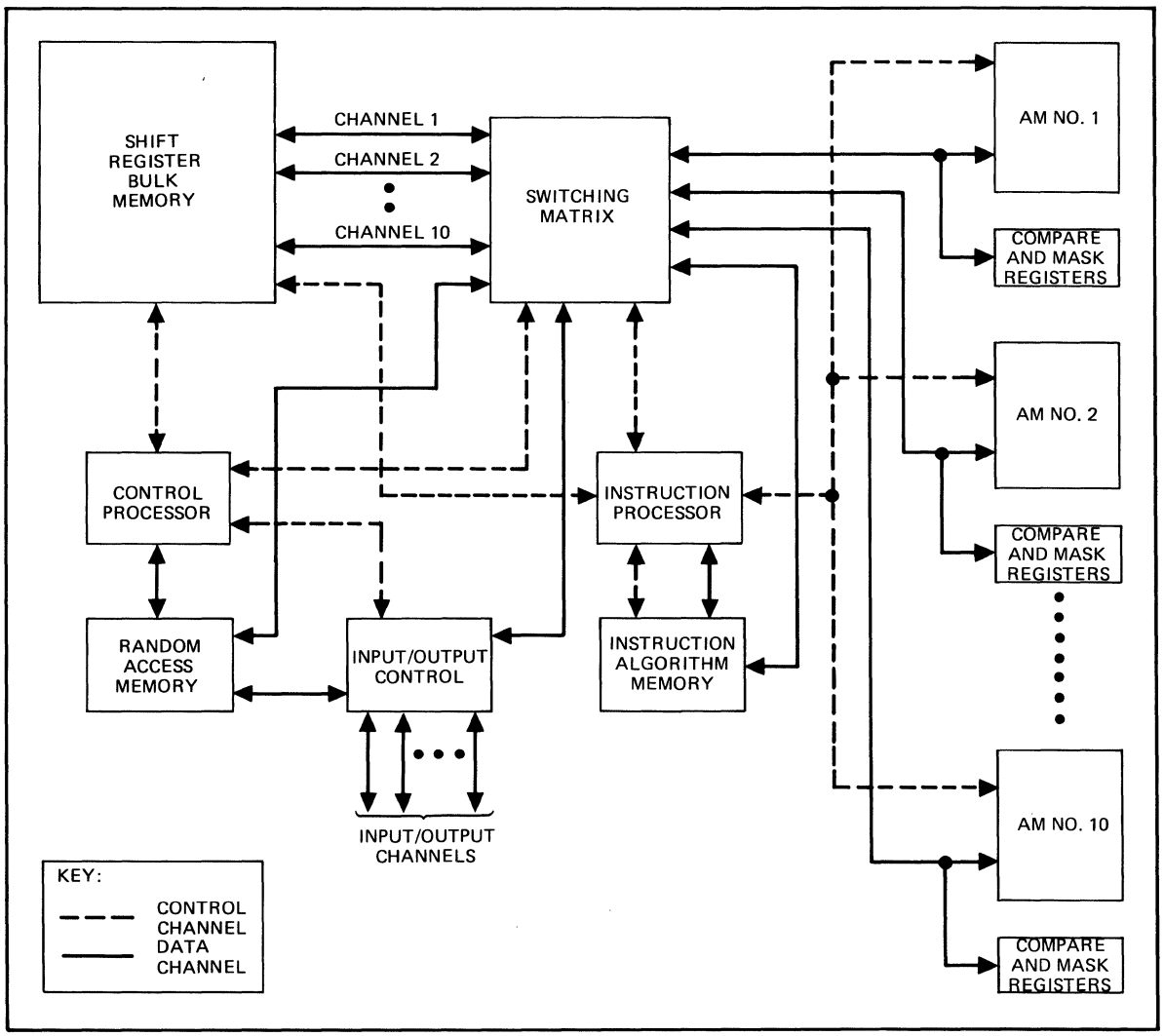


Figure 1. Associative/Shift-Register Processor Organization

5. Several logical operations involving the match flip-flops and two auxiliary sets of flip-flops, called the T1 and T2 flip-flops, whose functions will be described.

6. A number of auxiliary operations, such as the setting and resetting of all match flip-flops.

These operations are common to most "classical" associative memory designs. The number of words in each associative memory (ten memories are shown in the figure) is a function of the size of a subset of the average record in the data base, as will be discussed. The shift rate for the data registers of the associative memories during parallel operation is a nominal 5 MHz.

The bulk memory for the system consists of a set of individually-addressable MOS static shift registers.(a) These should be very large, at least 16,000 bits each, in order that the size of the address encoding and decoding matrices, and thus the cost of the memory, be as low as possible.

There are as many data transfer channels to the bulk memory (each channel bit-serial) as the number of associative memories. Those registers and only those registers assigned to a channel will shift their contents when data transfer commands are executed. This makes it possible to shift registers that are not involved in data transfers, by assigning them

(a) The newly-emerging charge transfer technology may make such devices equally or more suitable as a bulk memory for this system, it being a requirement that it be possible to suspend the shifting operation for brief periods (100 msec.) without loss of information. The magnetic bubble memory is another potential candidate.

to data transfer channels and not enabling the outputs at the other end of the channels.

The shift rate of the registers in the bulk memory is the same as that for the associative memories, that is, of the order of 5 MHz.

The interface between the bulk memory and the set of associative memories is a switching network. This network permits each associative memory to be assigned to a data transfer channel from the bulk memory, and also permits the associative memories to be connected together in parallel in various combinations. This latter is accomplished by connecting the external registers of the associative memories in parallel and connecting the propagating channels (used for control of serial input and output operations on words in the memories) in series. This capability makes it possible for several of the associative memories to operate as a single large associative memory when the amount of data requires it, or to operate individually in simultaneous independent operation.

The remainder of the system organization consists of

1. an instruction processor, which controls the execution of the special processing algorithms used in the retrieval and modification operations. These algorithms are stored in a read-only instruction algorithm memory.
2. a control processor of more conventional organization, together with a random-access memory. This processor performs part of the control of the bulk memory operation, and also controls input and output operations.

Data Organization

The data bases are constructed from ordered triples, called relations.[a] The three items in each relation are called, respectively, the subject, attribute and value of the relation. The relations are organized into records. Each record is constructed from all of the relations involving a particular item, called the head item for the record. Each data entry in a record consists of the other two items in a relation. The data entries are unordered. There is a record in the data base for every item in the data, the item being the head item for that record.

In the records, each item is represented by a 24-bit number, called the item number. The user represents the item by a unique corresponding symbol string, called the item name.

Since the size of the data records and the physical records (i.e., the shift registers in the bulk memory) are different, the data records must be segmented. Each segment is stored bit-serial on a physical record.

All entries in every segment are 64 bits in length. The first entry in each segment is a header word containing the item number for the head item for the record. This is used in locating the segment and in identifying the corresponding record. One of the segments, called the head segment, contains the bulk memory addresses of all of the other segments in the entries immediately following the header. The other segments each contain only the address of the head segment in the entry immediately following the header. The remaining entries in all of the segments are the data entries, each consisting of a pair of 24-bit item numbers stored contiguously and left-justified in the entry.

Operation of the System

The principal function of the system is the selective retrieval from and modification of the data base. The criteria for the retrieval and modification are each specified by a set of relations, called the retrieval structure and the replacement structure, respectively.[b] In these structures, the known items are represented by their item numbers. Unknown items, to be determined by the retrieval operation, are each represented by one of a set of special numbers reserved by the software for this purpose.

Both the retrieval and replacement structures may contain unknown items. In the replacement structures, each relation specifies a set of relations to be inserted into the data base. Relations appearing in the retrieval structure but not in the replacement structure each specify a set of relations to be deleted from the data base.

The central process from which the retrieval operation is constructed is that of context addressing an unknown item. This is the process of identifying all items in the data that satisfy the "context" of relations in which an unknown appears in the retrieval structure. An item is said to satisfy this context if, for every relation in the control structure containing the given unknown, there corresponds a relation in the data containing the given item in place of the unknown, and which is otherwise identical. If there is another unknown in the relation in the retrieval structure, a relation in the data is considered "identical" for any item corresponding to that unknown, if it is identical otherwise.

Only the retrieval operations will be described here. The data modification operations are described in [3] for a similar system.

All retrieval operations are performed by first context-addressing all of the individual

[a] This is the data structure of the ASP language [2].

[b] This is also the structure of the ASP language [2].

unknowns in the retrieval structure, and then resolving any relations involving more than one unknown. For brevity in describing the processes, all unknowns in the following discussion will be either items or values, but not attributes. The processes can easily be extended to cases in which an unknown is an attribute.

To perform a context-addressing operation, two different processes are used. They are, respectively, the Load Subrecord operation and the Compare Record operation. To describe these operations, a retrieval structure consisting of several relations involving a single unknown shall be assumed. The unknown item shall be assumed to be either the subject or the value in each one of these relations.

The Load Subrecord Operation

To begin the context-addressing operation, one of the relations in the retrieval structure is selected (at random, if desired), and the head segment of the record for the subject or value (one of these will be a known item) is accessed in bulk memory. The Load Subrecord operation is then executed to load a subset of the record into one or more of the associative memories. This subrecord consists of all record entries which contain the same attribute as the relation from the retrieval structure. The operation is the following.

1. As the shift register containing the home segment is shifted, one entry at a time, those entries for which the attribute is the same as the attribute of the corresponding relation from the retrieval structure are selected and loaded into one of the associative memories. The selection is made by comparing each entry with the contents of a register called the selection register. With each entry so loaded, the T1 flip-flop in the word is set. This tag bit, at the completion of the context address, will be set at all entries containing values of the unknown item. At the same time, the addresses of the other segments of the record are retrieved (from the home segment) and the shift registers containing them are shifted to make the segments available for processing.
2. As the processing of a segment is completed, and as another segment becomes available at the output of a shift register, the process is repeated for that segment. If the associative memory becomes filled, another associative memory is selected by the system and is loaded in turn.

When the processing of all segments of the record is completed, the associative memory or memories will contain the entries for all values for the unknown item that are specified by the retrieval structure relation. If the retrieval structure contained only that one relation, the entire context-addressing operation would now be completed.

The Load Subrecord operation is illustration in Figure 2. The example is shown for the retrieval structure relation (A, R1, X), in which the unknown item is represented by the X. That relation is also shown in the upper left-hand part of the figure in directed-graph form (which is the ASP language representation). The record being processed is the one having the item A as the head item.

Five contiguous entries from the record are shown in the figure. These are the entries (R1, B1), (R2, B1), (R1, B3), (R9, B9) and (R1, B6). In the illustration, the first and third of these entries have already been selected and loaded into the associative memory. The input (i.e., compare) register is shown containing the most recently loaded entry.

The selection criterion (which is that the attribute of the entry be the item R1) is shown as the contents of the corresponding field of the selection register, with the symbol "D/C", representing "don't care", shown in the other fields. In the associative memory, the column labeled T1 represents the T1 flip-flops, which are set for each loaded entry. The column labeled MFF (match flip-flop) shows the flip-flop set for the most recently loaded entry. This represents the use of the match flip-flop to identify a single word in the associative memory for which some operation (in this case, the load operation) is to be performed.

The I1 and I2 fields in the associative memory words are the 24-bit fields for the item numbers for the other two items (that is, other than A) in each data entry. At the conclusion of the operation, the I2 fields will contain the values of the unknown item which satisfy the relation (A, R1, X).

The operation establishes the criterion for the size of the associative memories, which should be that of the average subrecord in the data base, rather than the size of the entire record.

It can be seen that the Load Subrecord operation is essentially balanced, in that neither the associative memory nor the bulk memory must wait for the other to complete an operation. The only exception is the delay in accessing the home segment, and a possible delay in accessing another segment. The operation, however, is not efficient, since the associative memory is not performing a parallel operation, but is only being loaded. As will be seen, the remaining operations in the context-addressing process are both balanced and efficient.

The Compare Record Operation

During the Load Subrecord operation, the home segment of the record corresponding to one of the other relations in the retrieval structure (that is, having the known subject or value from the relation as its head item) is being accessed. When the Load Subrecord

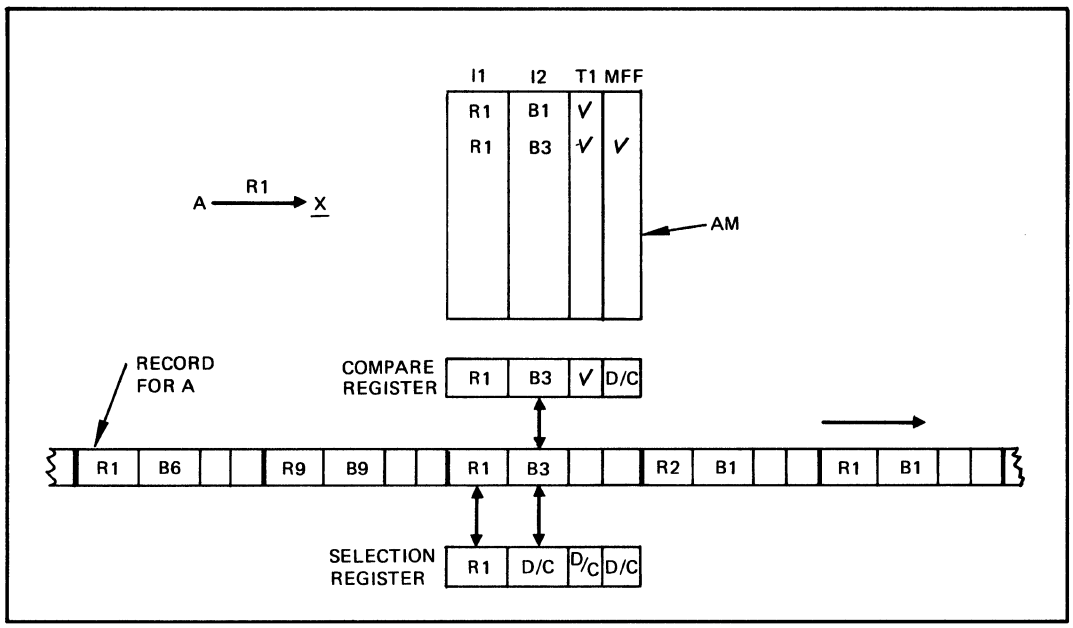


Figure 2. Load Subrecord Operation

operation is completed, this second record is processed, entry-by-entry, against the contents of the associative memory. This operation is called the Compare Record operation and, for each entry, is as follows.

1. The attribute of the entry is compared with the attribute from the corresponding retrieval structure relation. As in the Load Subrecord operation, the selection register is used in this process.
2. At the same time, the entry is compared simultaneously with all entries in the associative memory (i.e., the subrecord already loaded), comparing only the value fields and the T1 flip-flops. If both comparisons 1 and 2 are successful, the match flip-flop is set at each matching entry in the associative memory. (All match flip-flops are reset before the first entry is processed.)

After the last entry in the record has been so processed, the T1 flip-flops and the (corresponding) match flip-flops are logically ANDed together, and the results stored in the T1 flip-flops. Those entries that now have their T1 flip-flops set are the entries which contain, in their I2 fields, all values of the unknown item that satisfy both of the retrieval structure relations thus far processed. If there are no other relations in the retrieval structure, the context addressing operation is now completed.

The Compare Record operation is illustrated in Figure 3, which shows a retrieval structure of two relations. The first of these, (A, R1, X), is shown as already having been processed, using the Load Subrecord operation. The second relation in the retrieval structure is the relation (B, R2, X), and the record shown being processed, using the Compare Record operation, is the record for the item B. The selection register is shown containing the attribute from the relation in its I2 field.

Five contiguous entries from the record for B are shown, with the first three of these having already been processed. It is seen that the first and third of these entries have satisfied both compare operations, and the match flip-flops are set at the second and fourth words in the associative memory as a result.

If there are more than two relations in a retrieval structure involving a given unknown, the third, fourth, etc. of the relations are processed exactly like the second, using the Compare Record operation.

It is seen that the Compare Record operation is both balanced and efficient. It is balanced in the same way as the Load Subrecord operation, and it is also efficient since, unlike the former operation, the associative memory is performing parallel compare operations for every entry in the record. Moreover, the access delays experienced in connection with the processing of the first record will seldom if ever be encountered for the remaining records. This is because all of the records involved in the context of the unknown are known at the start of the context-addressing operation, and thus can be searched for simultaneously. By the time the first record is processed, at least one of the other records will be accessible for processing in turn.

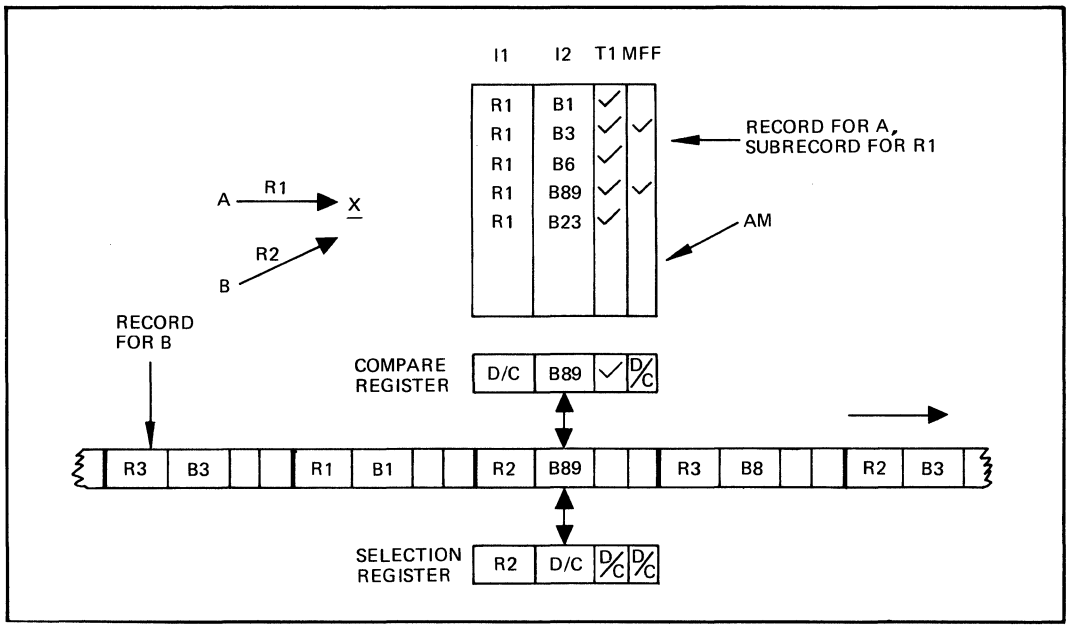


Figure 3. Compare Record Operation

If a replacement structure contains several unknowns, the system design permits several of them to be context-addressed simultaneously in the fashion just described. This is a result of having more than one associative memory and more than one data transfer channel between the bulk memory and the associative memories. The simultaneity is limited only by the number of associative memories and by the fact that more than one associative memory may be required for the selected entries from an unusually large record (selected by the Load Subrecord operation).

If there are no relations in the retrieval structure involving more than one unknown, the process of identifying all values of all unknowns in the retrieval structure can be accomplished by the processes already described. If there are such relations, a number of other operations have been defined. All of these operations require the use of more than one associative memory, and also special logic for operating on the success/fail results of the various comparison operations. All of these operations are performed, when applicable, after all individual unknowns have been separately (and simultaneously) context addressed. Three of the operations shall be described here. Each of them applies to a particular configuration of interrelated unknown items. For other configurations, the corresponding operations can be derived by reference to these.

The Find Pairs Operation

The first of the operations for interrelated unknown items, called the Find Pairs operation, identifies all corresponding pairs of values for two unknown items that are in a single relation in the retrieval structure. Each of these pairs corresponds to some entry in that record that has the attribute from the retrieval structure relation as its head item. These matching entries each represent relations in the data which have the same attribute as the retrieval structure relation, and whose subject and value are each candidates for the respective unknown items in the retrieval structure relation. The candidates are those items that have been identified by the earlier context addressing of the two unknowns.

Once the entries for the corresponding pairs of values have been identified, one or more of the following operations is performed.

1. The entries themselves are tagged directly in the record in the bulk memory by setting bits in the tag fields of the entries (bits 48-63).
2. The entries are written in an unused associative memory for later processing. Examples of such processing will be shown.
3. The entries are written into an unused (blank) physical record in the data base for later processing.
4. The entries are retrieved for output to the user (assuming that the entire retrieval operation has been completed with the completion of the current operation).

Figure 4 shows the hardware configuration of registers, associative memories and data records used in the Find Pairs operation, and illustrates the use of the operation in connection with an example retrieval structure. The figure shows three of the associative memories from the system, labeled AM#1, AM#2, and AM#3.

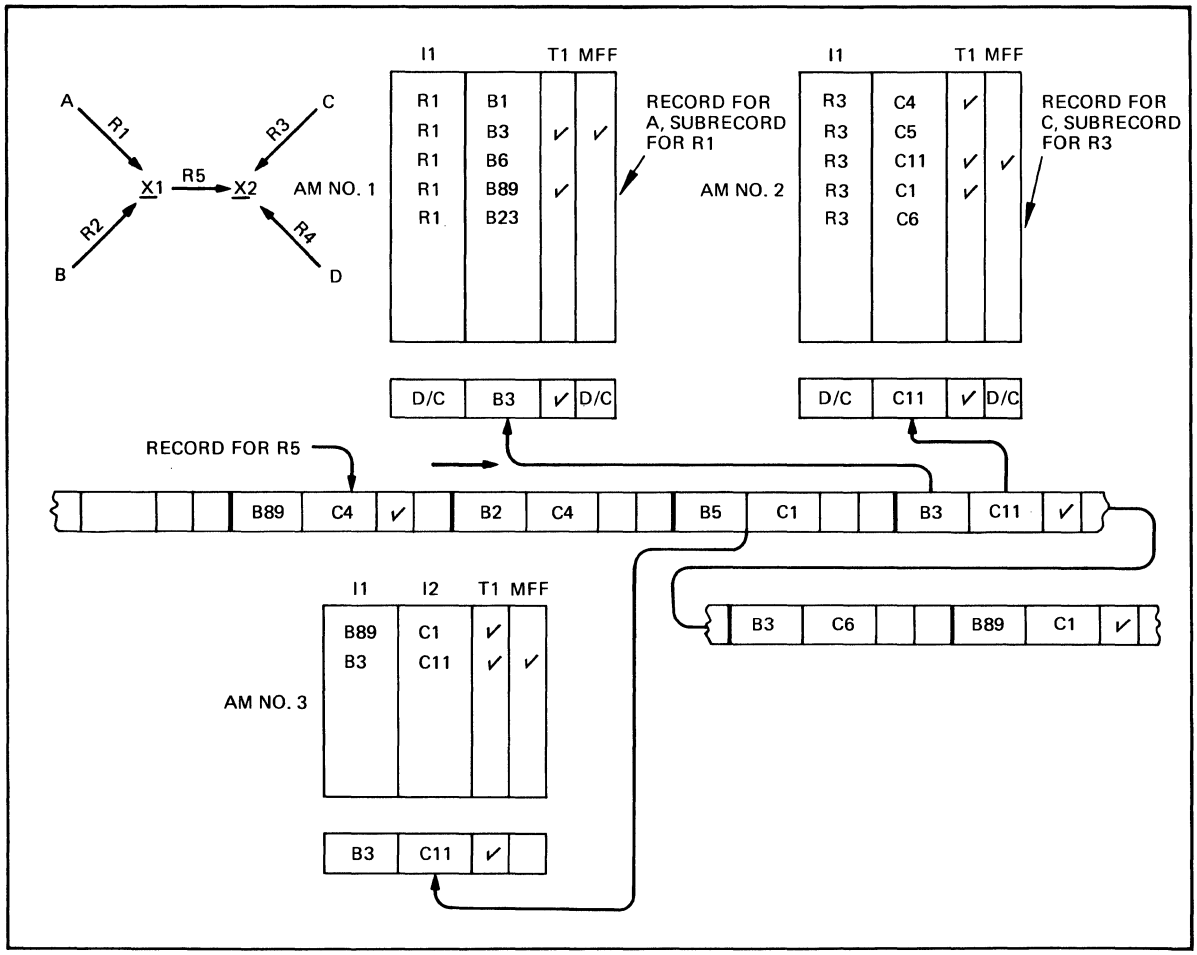


Figure 4. Find Pairs Operation

AM#1 and AM#2 each contain entries from the record that was processed first in context addressing one of the two unknown items. AM#1 contains the entries involving the subject of the relation involving the two unknowns. AM#2 contains the entries involving the value of the relation. The configuration in the figure also includes part of a record, drawn as though it were a tape, that is being processed against the contents of AM#1 and AM#2. The third associative memory, AM#3, is a spare memory that has been assigned for holding the matching entries from the record (i.e., the corresponding pairs of values for the two unknowns) if that option is specified in the operation.

To begin the Find Pairs operation, AM#3 is cleared, and the match flip-flops in all of the associative memories are reset. Following this, the first match flip-flop in AM#3 is set, "Don't care" (D/C) conditions are put into the compare registers of AM#1 and AM#2, as shown, and the T1 fields of all three compare registers are set.

Now each entry from the record for the attribute of the retrieval structure relation is processed as follows.

1. The contents of the I1 field in the entry are transferred to the I2 field of the compare register of AM#1. The contents of the I2 field of the entry are transferred to the I1 field of the compare register of AM#2. Following this, the record is shifted to the next entry.
2. A simultaneous compare operation is performed on both associative memories at the same time, and the two sucess/fail conditions are ANDed together. (The success condition is indicated by the setting of at least one match flip-flop in an associative memory.)
3. If both compare operations are successful, the current entry (which is now known to contain a corresponding pair of values for the two unknowns) can be copied into the current entry position of AM#3 (the entry being defined by the setting of the match flip-flop). Or, if desired, the entry can be copied into another

record, reserved for the purpose, in bulk memory. As a third alternative, the entry can be tagged directly in the record in its T1 field (or in one of bits 48-63).

After all of the entries in the record have been processed as just described, those entries that contain corresponding pairs of values for the two unknowns will have been identified and tagged and/or copied.

The example illustrated in Figure 4 illustrates the Find Pairs operation for the retrieval structure consisting of the five relations (A, R1, $\underline{X}1$), (B, R2, $\underline{X}1$), ($\underline{X}1$, R5, $\underline{X}2$), (C, R3, $\underline{X}2$) and (D, R4, $\underline{X}2$). The two unknown items are represented by the symbols $\underline{X}1$ and $\underline{X}2$. AM#1 contains the candidates for $\underline{X}1$, as determined by the context addressing of $\underline{X}1$. They are in the 12 fields of those entries that have tag T1 set. Similarly, AM#2 contains the candidates for $\underline{X}2$.

Six entries from the record for R5 are shown being processed against the contents of AM#1 and AM#2. R5 is the attribute of the relation that interrelates the two unknowns. Each entry in that record contains pairs of potential values of $\underline{X}1$ and $\underline{X}2$. The first three entries have already been processed, and it is seen that the first and third of these have matched. They are both tagged in the T1 fields of the entries themselves and have also been copied into AM#3. In particular, the third entry has just been tested, and the match flip-flops in AM#1 and AM#2 are set at the matching entries.

The Process Threes and Process Fours Operations

There are a number of possible retrieval structure configurations which involve three or more interrelated unknown items. For each such configuration there is a corresponding instruction with its hardware configuration and processing algorithm. The hardware configurations for two of these instructions, the Process Threes and Process Fours instructions, are shown in Figures 5 and 6, respectively, together with example retrieval structures. The two figures are given for illustration only. The operations themselves are described in detail in reference [3] for a similar system. Only a brief discussion is given here.

The Process Threes operation determines pairs of corresponding values of two unknown items that are indirectly related in a retrieval structure through a third unknown item. (In the example in Figure 5, the third unknown item is $\underline{X}2$.) The Process Fours operation determines such pairs of values for cases in which the two unknowns are related through two intervening unknown items ($\underline{X}2$ and $\underline{X}3$ in the example in Figure 6.)

Both operations are performed after all corresponding pairs of values for the directly related unknown items in the retrieval structure have been determined using the Find Pairs operation. These corresponding pairs have been variously stored on blank records or in associative memories as required for the current operation.

For the Process Threes operation, only one associative memory is required. (The second one shown in Figure 5 is for optional storage of the corresponding pairs determined by the operation.) The Process Fours operation requires two associative memories.

Cost and Performance Considerations

The associative/shift-register system is essentially a balanced system with respect to its two primary subsystems, the associative memories and the bulk memory. The shift rates for both memories are the same, and both memories are kept operating at or near that shift rate during normal operation.

Access delay is small in all processing of records from bulk memory. This is because all segments of a record except the first can be accessed nearly simultaneously and, once accessed, can be kept in readiness for immediate processing. There is a delay in accessing the home segment; this averages 1.6 msec. for the 16,000-bit registers in the bulk memory. Access delays for the remaining segments of the record, and for the segments of any other record being accessed at the same time for later processing, will be small or nonexistent.

As an example, consider a record consisting of 640 segments divided into ten segments of 64 data entries each. The total processing time for such a record (for the Load Subrecord and Compare Record operations) will be very close to the 1.6 msec. average access time for the home segment plus 0.8 msec. for processing each segment, a total of 9.6 msec.

For rotating memories, the limitation on processing speed is largely a function of the rate of rotation, since the instantaneous data transfer rates for modern fixed-head disks and drums are high. For such memories, the processing of most records will require an entire revolution (33 msec. for the typical disk rotating at 1800 rpm.) plus a fraction of a disk revolution for accessing the first segment to be processed. [a]

A large part of the advantage of using shift registers rather than rotating memories lies not in the increased speed but in the relative simplicity of system design and operation.

[a] It is assumed that if a disk is used as the bulk memory, every segment of a record would contain the addresses of the remaining segments. If cueing of access were then used, one of the segments could be accessed in an average of $1/n + 1$ of a disk revolution, where n is the number of segments.

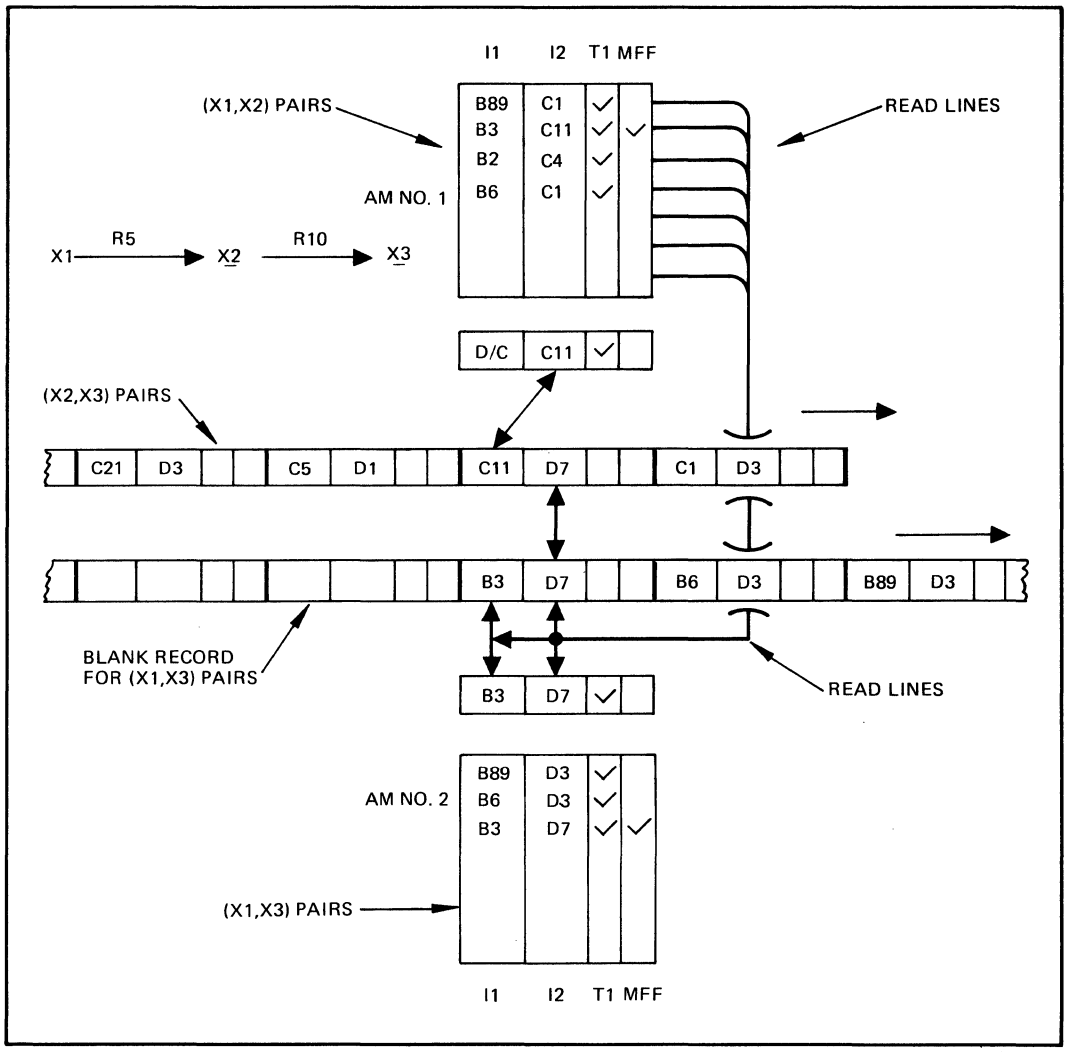


Figure 5. Process Threes Operation

Shift registers, for example, do not require cueing of accesses in order to minimize average access time per record. And they do not require the related buffering or the processing effort needed to handle the buffering and cueing operations. Moreover, such elaborate techniques as deferred modification[a] are much less needed when shift registers are used.

The present ratio of costs for MOS shift registers to fixed-head disks is of the order of 9 to 1. At this ratio, the sacrifice in processing speed, processing and buffering costs and design effort when disks are used may still be justified. However, with the reduced costs of LSI to be expected in the near future (charge-transfer devices costing about 1/4th the cost of disks are being announced), the simpler shift-register memory should be considered in any present effort to achieve a balanced associative system.

[a] This is a technique for increasing record-processing throughput in which modifications to the data base are made in a reserved region in fast memory as soon as they are determined, rather than in bulk memory. The main data base is then modified later from the contents of this buffer, overlapping later operations. In this way, subsequent operations on the data base can proceed immediately. This technique requires that all search operations must include a search of the buffer as well as the bulk memory, and is in general very complicated to implement.

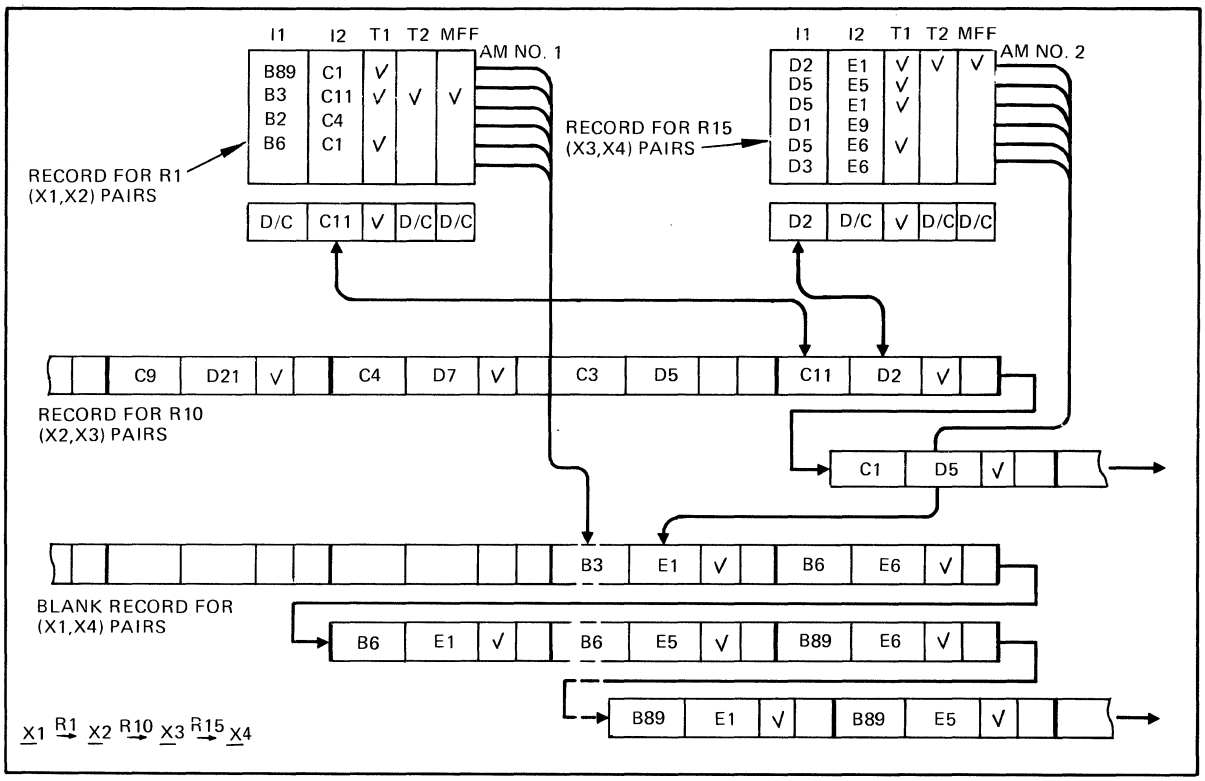


Figure 6. Process Fours Operation

References

[1] D.A. Savitt, H.H. Love, and R.E. Troop, Association-Storing Processor Study, Defense Documentation Center, Document No. AD 488538, (June 1966), 202 pp.

[2] Donald A. Savitt, Hubert H. Love, Richard E. Troop, Association Storing Processor, Defense Documentation Center AD 818529 and AD 818530, (June 1967), 2 vol., 182 pp. and 300 pp.

[3] Hubert H. Love, An Associative Processor Using Bulk Storage, Rome Air Development Center, Report Number RADC-TR-180, (June 1969), 146 pp. (also available from DDC).

THE USE OF TWO LEVELS OF PARALLELISM TO IMPLEMENT AN EFFICIENT PROGRAMMABLE SIGNAL PROCESSING COMPUTER

John P. Ihnat, Tomlinson G. Rauscher, Barry P. Shay,
Harold H. Smith, and William R. Smith
Information Processing Systems Branch
Communications Sciences Division
Naval Research Laboratory
Washington, D. C. 20375

Abstract -- The use of two levels of parallelism facilitates the design of an efficient programmable signal processing computer. At the system level, multiple functional units (multiprocessors) perform distinct functional tasks such as data gathering, data organization, and signal transformation. At the implementation level, horizontal microprogrammed control of parallel resources effects flexible and efficient processing.

Introduction

Modern signal processing systems perform many tasks by sampling analog signals and transforming the sampled digital data. In systems like radar and sonar there is typically so much information to analyze that it has been necessary to develop special-purpose hardwired devices to sample and transform the data in real time. The emergence of LSI circuit technology and high speed memories provides the capability of developing programmable signal processors which would reduce proliferation of special purpose devices, reduce the manufacturing cost (by economies of scale), and simplify maintenance. The use of two levels of parallelism facilitates the design of such a programmable signal processor [1].

Parallelism at the System Level

At the system level, an efficient signal processing computer assigns distinct processes to different functional units which operate in parallel. For system supervision and simple data organization and transformation, the system employs a sophisticated controller. For signal transformations a specialized arithmetic processor is required. Additional functional units collect and store data and control communication among other units. In the AN/UYK-17 (AADC/SPE) computer [2] (see Figure 1) separate functional units perform such distinct processes.

The Microprogrammed Control Unit (MCU) is the system controller. Its functions include data management, process scheduling, I/O control, interrupt handling, and some applications routine processing. The MCU massages (e.g. by ordering or scaling) signal information, placed in buffer memories by I/O devices, into a form amenable to transformation by signal processing algorithms, and leaves it in buffer memory for processing by a special purpose arithmetic unit. After the data is transformed, the MCU may perform some post processing functions and store information for later retrieval. The MCU also performs system functions such as handling operator requests, controlling displays, etc.

The Signal Processing Arithmetic Unit (SPAU) is the system arithmetic processor. Its function is to perform high speed execution of processing operations on arrayed data. These operations include spectrum generation, convolution, correlation, and digital filtering. SPAU processing is scheduled by the MCU. After SPAU processing is initiated, the SPAU operates independently of the MCU.

The AN/UYK-17 contains up to eight buffer storage modules (BSMs), which provide central, high speed (150 nanosecond cycle time) memory for the system. Each BSM contains 4096 32-bit words. The BSMs provide storage for MCU executive and application data tables, system data arrays, working storage for MCU and SPAU processing operations, and buffer areas for I/O data movement. Because the MCU supervises I/O operations and storage of data in the BSMs, the SPAU need not consider the problems of I/O processing; its data resides in the high speed BSMs.

To provide a fast and flexible means of moving data between BSMs and peripheral devices, the AN/UYK-17 contains one or more selector channel controllers (SCCs). The storage control unit (SCU) provides a switching interface between the independent BSMs and other system components. Because each MCU, SPAU, and SCC can access BSMs every clock cycle, the SCU switches each unit to the BSM it is addressing and resolves conflicts by a priority mechanism.

To provide general intermodule communication, the AN/UYK-17 contains a connector called the Z-bus which consists of sixteen

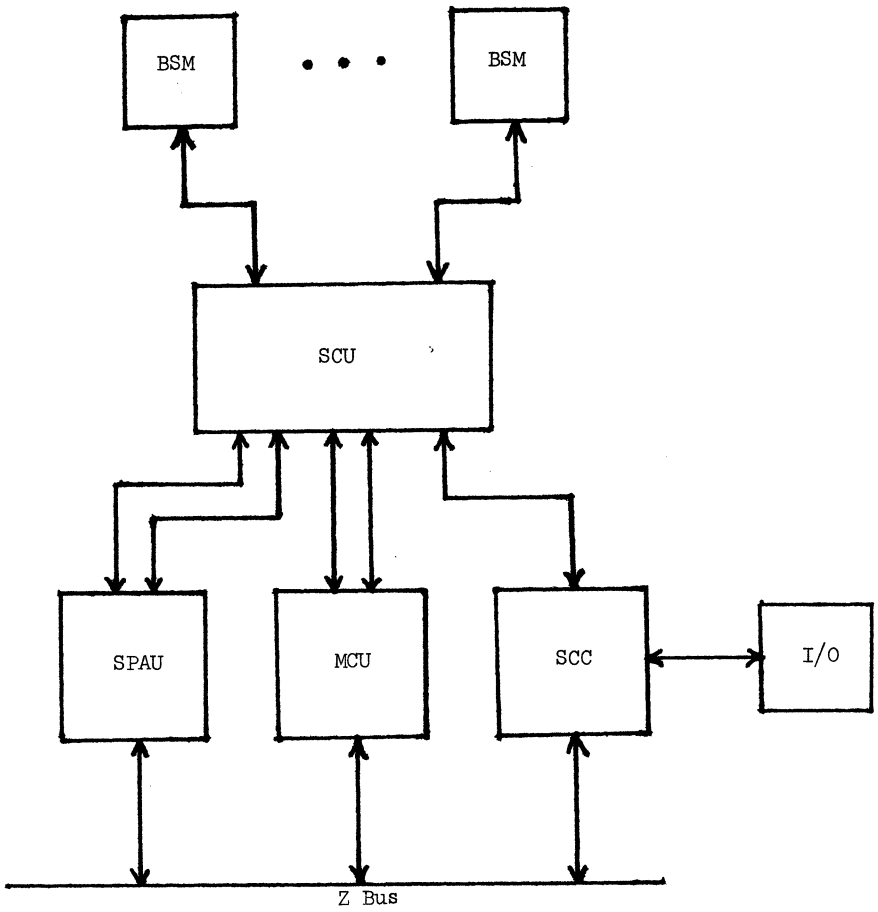


Figure 1. AN/UYK-17 System Block Diagram

bidirectional data lines and fourteen control lines. In addition, there is a sophisticated interrupt system through which system components and peripheral devices can notify central control, the MCU, of changes in their operation or status.

Parallelism at the Implementation Level

In the implementation of system components that perform multiple functions of a similar nature, the use of parallelism can significantly improve the performance of the components and hence of the system. In the AN/UYK-17 the MCU and the SPAU contain several resources that operate in parallel. User written horizontal microprograms control these resources.

Parellism in the SPAU

The capability to effect the second order recursive filter and the FFT butterfly is fundamental in signal processing [1,3]. Figure 2 shows the general configuration of a second order recursive filter [1]. Z^{-1} in Figure 2 represents a unit delay while the circles indicate addition or multiplication by a constant. The output y at any time can be described in the following two step computation, which uses the labels defined in Figure 2:

$$W_o = x - B_1W_1 - B_2W_2$$

$$y = W_o + A_1W_1 + A_2W_2$$

The data flow graph shown in Figure 3 follows from these equations. Squares in Figure 3 represent data items and circles represent multiplication or addition. "p," "q," "r" and "s" are intermediate data items, while T_1 and T_2 represent delay operators. Note the possibilities of performing the operations in parallel.

Figure 4 shows the arithmetic section of the SPAU. As in the data flow graph of

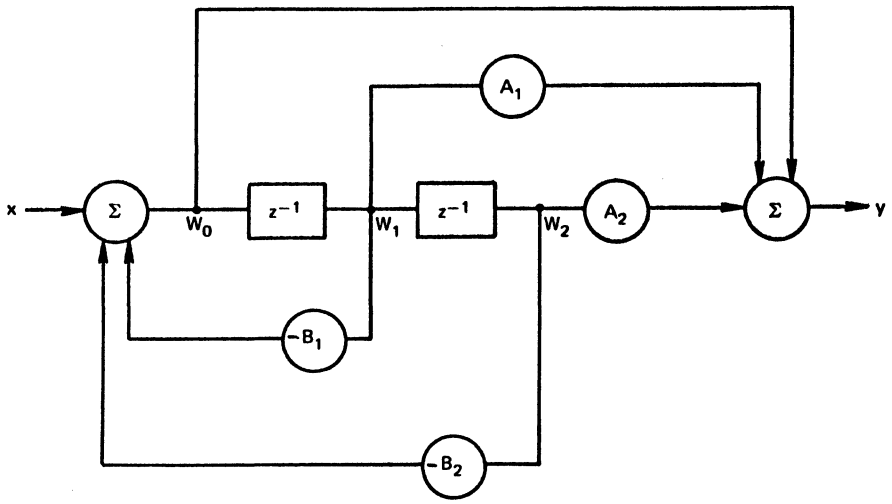


Figure 2. Second order recursive filter

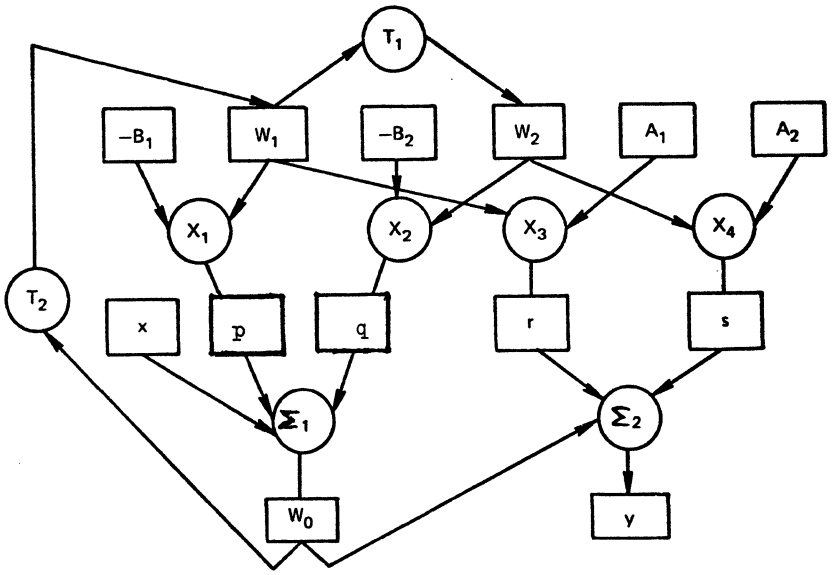


Figure 3. Data flow graph for second order recursive filter

Figure 3 there are four multipliers. These multipliers operate in parallel and produce a product every clock cycle (150 nanoseconds). Although there are four adders, the results of adders one and three may be inputs to adders two and four, respectively, in the same cycle; so two pairs of additions can be performed consecutively in a single cycle. This is equivalent to two three-input adders. The X and Y local stores are used to store intermediate results and to hold data that have been read from or will be written to BSMs. The adders, buffer reads and writes, and additional register transfers operate in parallel with each other and with the multipliers.

Fundamental to the computation of the fast Fourier transformation (FFT) is the FFT butterfly [4]. For data points represented as complex numbers, Figure 5 shows the data flow graph for computing the FFT butterfly. In this figure $X_m(i)$ and $X_m(j)$ are data inputs to the butterfly while $X_m(i+1)$ and $X_m(j+1)$ are output items. W^k is a weight term, where $W_R^k = \cos\left(\frac{2\pi k}{N}\right)$ and $W_I^k = -\sin\left(\frac{2\pi k}{N}\right)$ with N = the

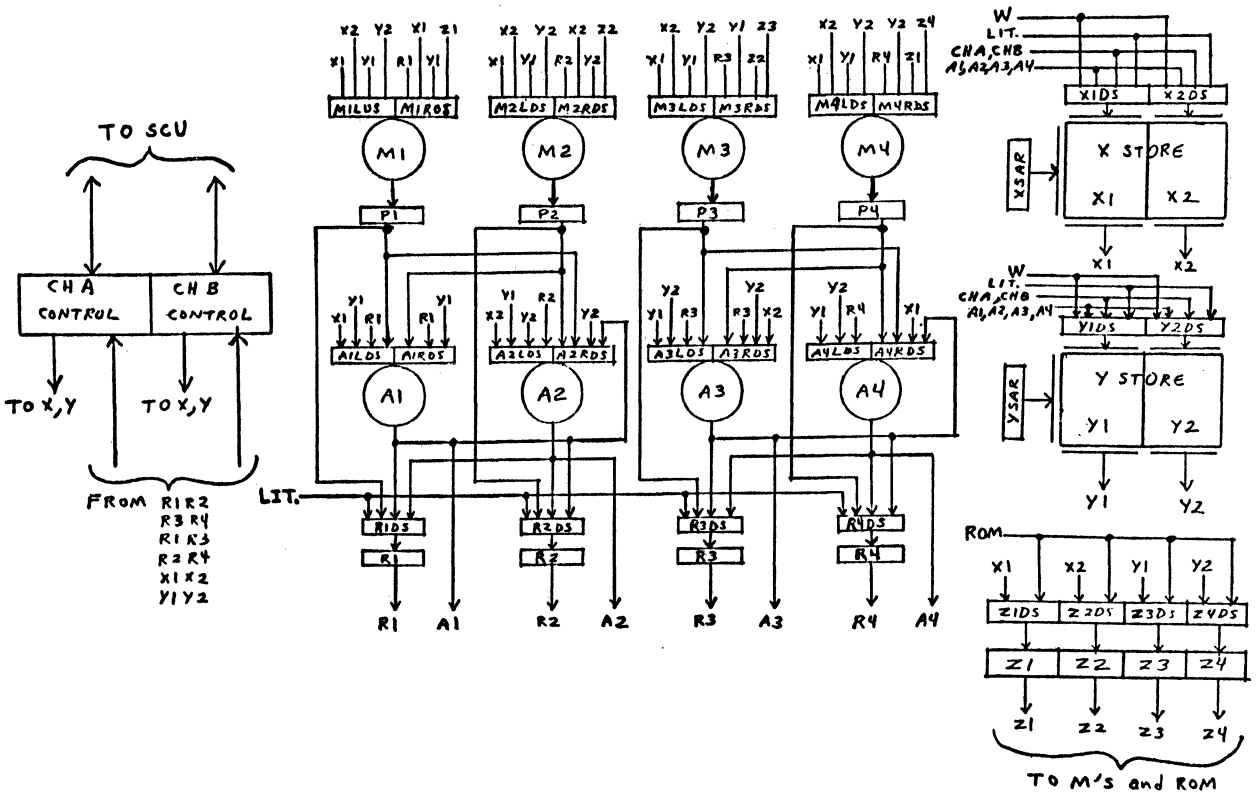


Figure 4. SPAU arithmetic section

total number of input data points to the FFT. By considering the data flow graph in Figure 5 two conclusions can be reached that may assist computation:

1) No memory cell in the data flow graph is reused by subsequent operators. Hence, the computation may be executed in a pipelined fashion.

2) The left-right symmetry in the data flow graph permits an increase in throughput when the input data points are all real rather than complex.

Referring again to Figure 4, the effect of the FFT butterfly computation on the design of the SPAU arithmetic section is apparent.

Note that the schemata for the computation of the FFT butterfly and second order recursive filter were similar enough so that the same hardware could easily be used to effect both algorithms. Register selectors facilitate dynamic reconfigurability. Control of the SPAU is effected by horizontal microprograms; 160 bit microinstructions which contain 63 fields control the resources of the arithmetic section and also the addressing section (which has three independent address formation units for computing buffer and ROM addresses) and sequencing mechanism.

Parallelism in the MCU

Like the SPAU, the MCU (see Figure 6) is controlled by horizontal microinstructions which execute in 150 nanoseconds, the system cycle time. Microinstructions in the MCU contain 64 bits which define seventeen fields. These fields control specific MCU facilities:

1) buffer input and output

2) source and destination register selection for the ALU/shifter

3) ALU/shifter operation

4) interrupt control

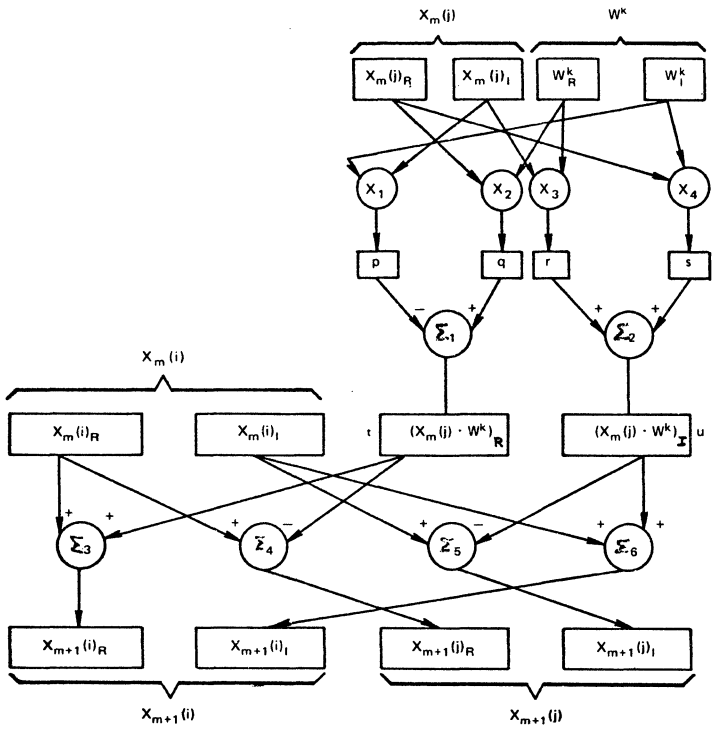


Figure 5. Data flow graph for FFT butterfly

5) auxiliary register transfers

6) sequence control.

As in the SPAU, the facilities operate in parallel. Thus throughput is normally greater than for standard systems. For example, the MCU can transpose a 40 by 40 matrix (involving 3200 memory references) in less than 1700 cycles.

AN/UYK-17 Configurations

Because the AN/UYK-17 system provides general intermodule communication facilities (the Z-bus and the interrupt capabilities), system components can be configured in a variety of ways. The basic simplex system consists of an MCU, a SPAU, an SCU, four BSMs, and an SCC. Additional components may be connected; an example (see Figure 7) follows the architecture of the CDC 6600 computer. One or more MCUs can serve as peripheral processing units that control I/O devices. A master MCU can serve as a (scoreboard) scheduler which manages the buffer memories and schedules the operation of the parallel functional units, i.e., the SPAUs. The SPAUs execute various arithmetic processes, communicate only with the high speed buffer memories, and are subservient to the master MCU.

References

[1] Barry R. Shay, Design Considerations of a Programmable Predetection Digital Signal Processor for Radar Applications, Information Systems Group, Naval Research Laboratory, NRL Report 7455, (December, 1972), 54 pp.

[2] W. R. Smith, Jr., J. P. Ihnat, H. H. Smith, N. M. Head, Jr., E. Freeman, Y. S. Wu, and B. Wald, AN/UYK-17 Signal Processing Element Architecture, Information Processing Systems Branch, Communications Sciences Division, Naval Research Laboratory, NRL Report 7668, (in press).

[3] Tomlinson G. Rauscher and Barry P. Shay, "The Influence of Computation Schemata Representations of Signal Processing Algorithms on the Architecture of the AN/UYK-17 Computer," Symposium on Complexity of Sequential and Numerical Algorithms, (May, 1973).

[4] Bernard Gold and Charles M. Rader, "Digital Processing of Signals", McGraw Hill, New York, 1969.

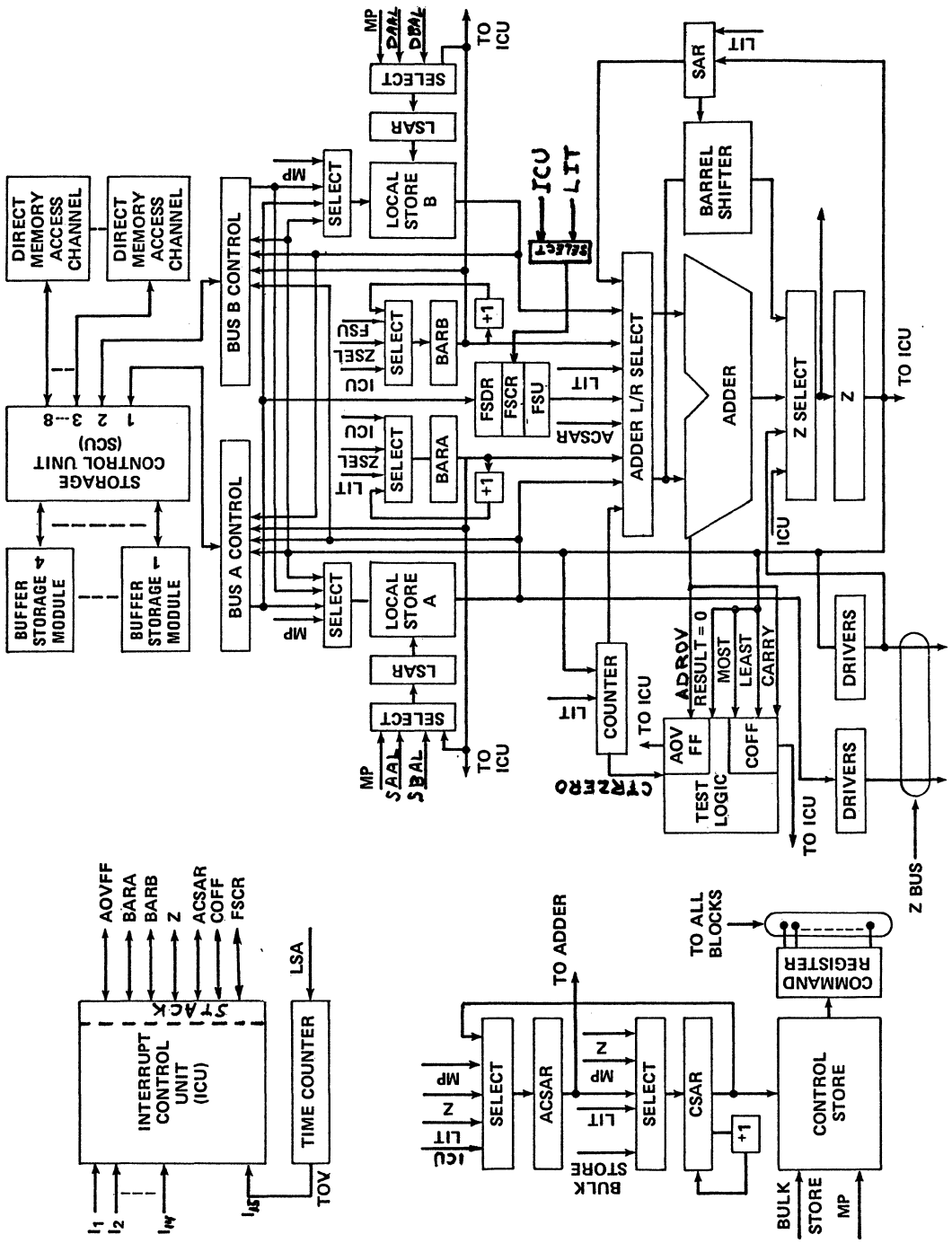


Figure 6. Microprogrammed Control Unit

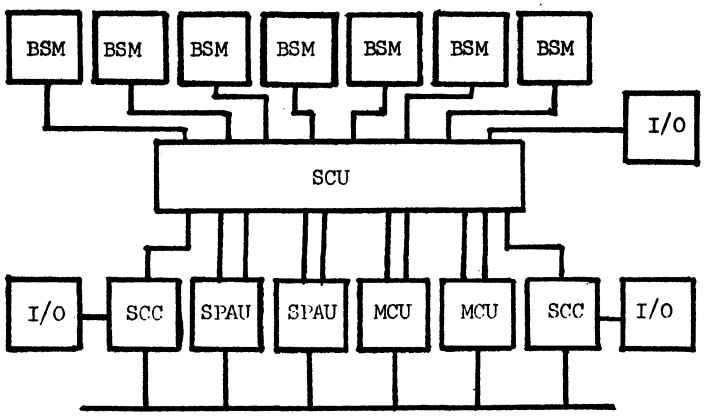


Figure 7. Alternate Configuration

ASYNCHRONOUS NETWORK OF SPECIFIC MICROPROCESSORS

François DROMARD and Gérard NOGUEZ
Institut de Programmation
Université PARIS VI
4 place Jussieu 75005 Paris FRANCE

Summary

This paper is a summary of another one [1],[2] where the multimicroprocessors architecture (MICROPUS) is described. The attempt here is to clarify the exchange mechanism between the microprocessors.

The studied network is designed on a local scale. It is composed of:

1) microprocessors
2) paths between them.

The microprocessors have specific functions (and sometimes a specific structure). They have their own storage which contains their specific data and working area.

At a logical level, a task consists of a sequence of specific sub-tasks.Each sub-task is processed by one and only one microprocessor.Only one sub-task can be processed at a given time.So that, at any time, a task needs just one resource: a microprocessor or a path (in order to be transmitted to the next microprocessor capable of processing next sub-task). At the same time,it is possible to have several tasks in the network.

The exchange between microprocessors are asynchronous. It implies the paths are buffered. The network local scale allows designing a common mechanism to manage all the paths. this exchange set is composed of a finite number of:

1) containers
2) stations.

The stations can be attached to the microprocessors or can be used to collect free containers (collectors). A path is the connection between two stations.Only one container can stay in a station. The others are waiting for on one or more paths. A microprocessor can have one or several stations.In order to transmit information to another one,a microprocessor must use a container staying in one of its own stations. If there is none, it can request a container to a collector.If there is no free container, the microprocessor cannot emit and has to wait for the free container collecting. During this waiting time,no container is kept by the microprocessor.So, there is no deadlock.

Such an exchange mechanism can be implemented in two different ways using:

1) semaphores and queues
2) double linked looped lists.

The first way is simpler than " producer-consummer" algorithm, because semaphores are associated to the stations and not to the paths. Only one semaphore is attached to each station.Transmitting a container involve decrementing (P operation) the emitter station semaphore and incrementing (V operation)the receiver station one. The sum of semaphores values is constant and equal to the initial containers number.A collector station also has its semaphore processed like the others.

In the second way, the exchange mechanism is made of a double linked list memory. A looped list is associated to each station. Such a list heading stitch points to:

1) the stitch attached to the staying station container (downstream link).
2)the stitch attached to the last waiting container of the station (upstream link).

Let S the stations number and C the containers one. There are S lists and the list memory contains -at most- S+C stitches.

There are two basic operations:

1)checking a list is empty or not (demand).
2)transmitting a container from a list to an other one (supply).

The demand consists of testing if the heading down stream link points to itself or not.Transmitting a container (supply) involves the following linking operations:

1)extracting first container from emitter list and looping this list on itself.
2)inserting this container in receiver list (pointed by the heading downstream link).

Getting a free container then needs the following operations:

1)the microprocessor demands to the collector.
2)loops until this list is not empty.
3)does a supply operation from the collector list to its own one.

References

[1] F.DROMARD G.NOGUEZ "Asynchronous network of specific microprocessors ". International Workshop on Computer Architecture, June 1973 Université de Grenoble, FRANCE.

[2] Item at the Sixth Annual Workshop on Microprogramming, September 1973, University of Maryland.

AN APPROACH TO A RESTRICTED SCHEDULING-PROBLEM FOR MULTIPROCESSOR SYSTEMS

Sigram Schindler and Harald Lüdtke
Fachbereich 20 (Kybernetik)
Technische Universität Berlin

Abstract

The problem of scheduling N tasks - the operational precedence structure, $\prec$, of which is represented as a finite, acyclic, directed, weighted graph G - on a multiprocessor system consisting of M identical processors is studied. The weight W_I of node I, $1 \leq I \leq N$, we regard as the processing time of the task represented by node I, and we want all N tasks to be processed completely within total processing time CT. We assume that no preemptions are allowed. Memory for instructions and data is assumed to be infinitely large. Processor switching time is neglected.

In this paper some results are derived for the case that $\prec$ can be put together from forests and antiforests in a simple way. For the case that $\prec$ is the disjoint union of a 1-tree and a 1-antitree, the set of all suitable schedules is given for arbitrary M and CT.

I. Introduction

The paper investigates the problem of scheduling M identical processors if the computational work to be done is known in advance and if memory (or channel) requirements can be neglected. The computational work for the processors is described by a finite set of 'programs', or 'tasks' which have to be executed (or processed) and each of which can be assigned for execution to an arbitrary one of the M processors. Such an assignment can last until the task is completely executed or its execution can be interrupted because the executing processor is needed for another task which has no processor. Schedules for the processors that allow such interrupts are called preemptive schedules; schedules that do not allow interrupts are called nonpreemptive schedules. The set of N tasks T_i, $1 \leq i \leq N$, their execution times and their operational precedence structure, $\prec$, are represented by a finite, acyclic, weighted, directed graph G (abbreviated as FAWD G). The N nodes of this FAWD G stand for the given tasks T_i, the weight W_i of task T_i, $1 \leq i \leq N$, is regarded as its processing time, sometimes called length of the task T_i. We assume that all tasks in G have positive lengths. We can assign a processor to a task (and vice versa) iff the task is free, i.e. it has no predecessor . As soon as a processor is assigned to a task it starts reducing the length of the task, i.e. processing the task; this reduction of the length of a task takes place with a constant, positive and finite speed. If the length of the task is reduced to 0, the task together with all its outgoing arrows is deleted from the graph. The units of length and time are determined such that a processor reduces a task by one unit of length in one time unit.

For given FAWD G and M processors we are interested in $CT_{min}(G;M)$, i.e. the minimal total processing time for G by M processors. Given furthermore an upper bound $CT \geq CT_{min}(G;M)$ for the total processing time of G by the M processors we are interested in not only a single schedule but in the class $\underline{A}(G;M;CT)$ of all schedules that meet these conditions. This latter interest arises from the attempt to take into account further parameters of a computer, like for example, memory size, channel, transfer rate and memory control, and not only the number of processors, M.

The foregoing model is obviously not suitable to describe problems of effective resource utilization in today's general purpose computers. But it seems reasonable for the investigation of the processor allocation problem in a computer system of SIMD-type or MIMD- type (see [4]), if a few complexes of programs have to be processed very often by this system. The importance of the processor allocation problem in such systems can be derived from [16,17,18], where it is shown that processor utilization tends to be lower than 30 % if scheduling considerations are omitted.

At the moment the problem formulated above cannot be solved effectively in full generality. Moreover it was shown recently in [19] and [20] that probably no algorithm exists at all for computing an element from $\underline{A}(G;M;CT_{min}(G;M))$ for which the number of steps is bounded by a polynomial in N. This result that the problem of determining time-optimal schedules (i.e. elements from $\underline{A}(G;M;CT_{min}(G;M))$) probably cannot be solved effectively in the general case even holds if certain restrictions ([20]) are imposed on the problem. On the other hand there exists a long list of results saying that - due to other restrictions - the problem to determine time-optimal schedules is polynomially solvable in the cases investigated ([1,2, 3,6,7,8,10,11,21,22,23]).

In order to make clear how the results of this paper are related to previous work we now discuss this point in some more detail. Restrictions can be put upon the general problem by

- specializing the number of processors (e.g. to M=2),
- specializing the weights (e.g. to $W_i=1$, $1 \leq i \leq N$),
- specializing the precedence relation, $\prec$, of G (e.g. to G being a tree),
- forbidding preemptions.

Polynomial bounded algorithms to determine $CT_{min}(G;M)$ and an element from $\underline{A}(G;M;CT_{min}(G;M)$ are derived for the case

- M=2, $W_i=1$, $1 \leq i \leq N$, preemptions forbidden, arbitrary $\prec$ in [6-8];
- M=2, arbitrary W_i, preemptions allowed, arbitrary $\prec$ in [24] and [21], furthermore $\underline{A}(G;2;CT)$ for arbitrary CT is given in [21];

- arbitrary M, $W_i=1$, $1 \le i \le N$, preemptions forbidden, G is a forest[a] in [1], and in [23] if G is an anti-forest[a];
- arbitrary M, arbitrary W_i, preemptions allowed, G is a forest in [2], and [11], furthermore $\underline{A}(G;M;CT)$ for arbitrary CT is given in [10];
- arbitrary M, arbitrary W_i, preemptions allowed, G is an anti-forest in [23], where $\underline{A}(G;M;CT)$ for arbitrary CT is given, too.

The importance of these restrictions can be seen from [20], where the cases

- M=2, arbitrary W_i, preemptions forbidden, ← empty;
- M=2, $W_i=1$ or 2, $1 \le i \le N$, preemptions forbidden, ← arbitrary;
- M arbitrary, $W_i= 1$, $1 \le i \le N$, preemptions forbidden, ← arbitrary;
- M arbitrary, W_i arbitrary, preemptions allowed, ← arbitrary

are investigated. There it is shown that - even with these restrictions - the problems listed are 'polynomial complete' (see [19]), i.e. essentially that a polynomial bounded algorithm to determine a time-optimal schedule for such a problem would provide us with quite many polynomial bounded algorithms to solve well known problems for which polynomial bounded algorithms are not known today.

From this short survey we see especially that

- the nonpreemptive time-optimal scheduling problem for a general FAWD G with $W_i=1$, $1 \le i \le N$, and arbitrary M is polynomial complete, but
- the same problem is polynomial bounded if either M=2 or G is a forest or anti-forest.

This paper shows that for arbitrary M the restriction to forests or anti-forests is not necessary to get polynomial bounded scheduling algorithms and derives such algorithms for other FAWD's: For an arbitrary elementary FAWD[b] G, with $W_i=1$, $1 \le i \le N$, arbitrary integers $M > 0$ and $CT > 0$ the set of all nonpreemptive schedules for M processors to process G completely within time CT, $\underline{A}^n(G;M;CT)$, is described by a scheduling scheme, the algorithms of which are polynomial bounded in N. So the attempt to keep track of the increase of complexity when generalizing the scheduling problem is made with arbitrary M, allowing the precedence relation to become somewhat more complex than that of a forest or anti-forest, and not the other way around, with arbitrary FAWD G, allowing M to become larger than 2.

For the preemptive case and arbitrary W_i, $1 \le i \le N$, the authors will submit further and more general results in [26].

(a) A FAWD G is called a forest (anti-forest) iff each node in G has at most one immediate successor (predecessor). If a forest (or anti-forest) G is connected, it is called a tree (or anti-tree).

(b) A tree (anti-tree) is called 1-tree (1-anti-tree) iff all nodes with indegree (outdegree) ≥ 2 are located on one path (respectively) and for each edge of G its target (source)-node lies on this path. A FAWD G is called an elementary FAWD iff G is the disjoint union of a 1-tree and 1-anti-tree (see figure 1).

II. Results

Let G be a basic FAWD and let $W_i=1$, $1 \le i \le N$. For the case that preemptions are not allowed an algorithm bounded by N^{M+3} is derived first that determines $CT_{min}(G;M)$. If G is an elementary FAWD moreover the set $\underline{A}^n(G;M;CT)$ of all nonpreemptive schedules will be described for an arbitrary given $CT \ge CT_{min}(G;M)$. The problem of determining $M_{min}(G;CT)$ for arbitrary $CT > 0$ such that $\underline{A}^n(G;M_{min}(G;CT);CT) \neq \phi$ will be investigated elsewhere. For ease of presentation we introduce the following notions.

For a basic FAWD G we define G^- to be a maximal anti-forest of G and G^+ to be the subgraph of G consisting of all nodes of G not contained in G^- and all edges between these nodes in G (see figure 1). Obviously G^+ is a forest. Note that in general G^- is not uniquely defined and that the algorithm for constructing it is polynomial bounded (see [25]). We do not represent a graph G in the usual way (see figure 1) but use the self-explanatory representation of G in figure 2. Such a representation of G is called stripe representation R(G). Note that for each G there are infinitely many stripe representations R(G). For an arbitrary stripe representation R(G) of G let $b(R;G;CT-t)$ for $0 \le t \le CT$ be the number of tasks cut by a height-line through CT-t (see figure 2). For an arbitrary G, M and CT, the stripe representations of principal interest are those for which $\int_0^{CT} b(R;G;CT-t)\,dt=N$ and $b(R;G;CT-t) \le M$, $0 \le t \le CT$; these are called (M,CT)-stripe representations of G. An (M,CT)-stripe representation R(G) of G is called monotonic increasing (decreasing) iff $b(R;G;CT-t) \le b(R;G;CT-t')$ ($b(R;G;CT-t) \ge b(R;G;CT-t')$, respectively) in this representation R(G) for $0 \le t \le t' \le CT$.

For arbitrary integers $CT > 0$ and $M > 0$ and an arbitrary subgraph G' of G let pd(G';CT-t) be a mapping: $\{[i-1,i);\ i=1,\ldots,CT\} \rightarrow \{0,1,2,\ldots,M\}$, called the processor distribution for G'. For $0 \le t \le CT$ the value of pd(G';CT-t) gives us the number of processors available for processing of G' at time t. We sometimes use the shorter notation pd when no confusion is possible. For arbitrary pd let $\underline{A}^n(G';pd;CT)$ denote the set of all schedules for complete processing of G' with pd processors in CT time units. An (M,CT)-stripe representation R(G') of G' such that $b(R;G';CT-t) \le pd(G';CT-t)$, $0 \le t \le CT$, is called a (pd,CT)-stripe representation of G'.

Lemma 1: Let G be a FAWD, let $CT > 0$ and let pd:= pd(G;CT-t). Then $\underline{A}^n(G;pd;CT) \neq \phi \Leftrightarrow \exists$ (pd,CT)-stripe representation R(G) of G.

Proof:

⇐ : The level-by-level schedule, using pd(G;CT-t) processors at time t and applied to G in representation R(G), is in $\underline{A}^n(G;pd;CT)$.

⇒ : Let $S \in \underline{A}(G;pd;CT)$. Processing G according S defines a (pd,CT)-stripe representation of G. q.e.d.

Theorem 1: Let G be an arbitrary basic FAWD with weights $W_i=1$, $1 \le i \le N$, and G^- and G^+ derived from G as defined above. Let CT and M be arbitrary integers such that $\underline{A}^n(G;M;CT) \neq \phi$.
Then there exists a $pd^- := pd(G^-;CT-t)$, monotonic increasing, and a $pd^+ := pd(G^+;CT-t)$, monotonic decreasing, such that
$pd^-(G^-;CT-t) + pd^+(G^+;CT-t) \le M$, $0 \le t \le CT$, and
$\underline{A}^n(G^-;pd^-;CT) \neq \phi$ and $\underline{A}^n(G^+;pd^+;CT) \neq \phi$.

Proof: From $\underline{A}^n(G;M;CT) \neq \phi$ and Lemma 1 we get an (M,CT)-stripe representation R(G) of G and therefore the stripe representations $R(G^-)$ and $R(G^+)$, too. If these $R(G^-)$ and $R(G^+)$ are not monotonic increasing and decreasing, respectively, then we change them - without violating precedence rules in G - such that the resulting stripe representations of G^- and G^+ have this property. The way this exchange is done can easily be seen from figure 3 and is described now. In this case there exists an integer t_o, $1 \le t \le CT$, such that at least one of the two equalities $b(R;G^+;CT-t_o) = b(R;G^+;CT-t_o+1) + K1$ and $K2 + b(R;G^-;CT-t_o) = b(R;G^-;CT-t_o+1)$ holds for some $K1, K2 > 0$. We show how to proceed in the case that both equalities hold; the case that only one of them holds is treated by applying only a part of the procedure described subsequently. We first investigate the case $K1 = K2 = 1$.

As $b(R;G^+;CT-t_o) = b(R;G^+;CT-t_o+1) + 1$ and G^+ is a forest there is at least one task T in G^+ starting on heightline $CT-t_o$ in the present representation $R(G^+)$, that has no predecessor ending on heightline $CT-t_o$. Therefore it could be shifted up by one, leading to a new representation $R'(G^+)$, if by this action no precedence constraint of the original graph G were violated. Let us assume that we violated such a precedence constraint of G. Then there is a task T' in G^-, which is a predecessor of T in G^+. Then because G^- is a maximal anti-forest the task T belongs to G^-, what is a contradiction. Therefore T can be shifted up by one to start on heightline $CT-t_o+1$. The analogous argument allows us to shift one of the tasks of G^- ending on heightline $CT-t_o$ down by one to start on $CT-t_o$. This is true because G is a basic FAWD.

If K1 and/or K2 are greater than one and/or there exist several t_o, for which the above equalities hold, finite repetition of this

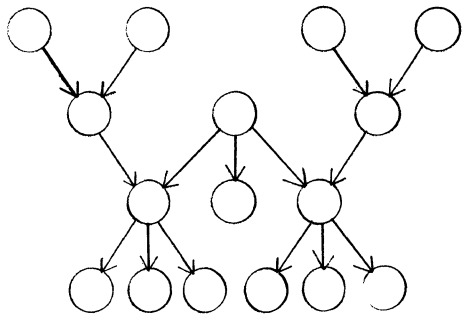


Figure 1a.

An example of a basic FAWD G represented in the usual way.

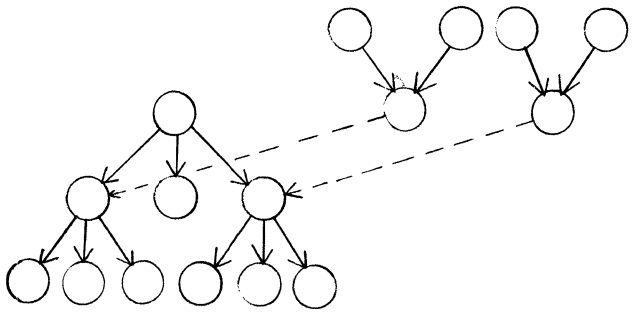


Figure 1b

A decomposition of G into a maximal anti-forest G^- and a forest G^+. G^+ is the subgraph of G consisting of all nodes of G not contained in G^- and all edges between these nodes in G.
The edges deleted from G are drawn by dashed-lines.

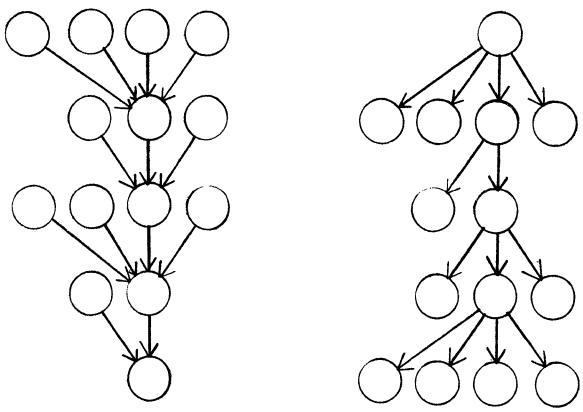


Figure 1c

An example of a 1-tree and a 1-anti-tree, respectively.

procedure eventually provides us with a monotonic decreasing (M,CT)-stripe representation $\overline{R}(G^+)$ of G^+ and a monotonic increasing one $\overline{R}(G^-)$ of G^-. Obviously then G is brought to another (M,CT)-stripe representation, immediately defined by $\overline{R}(G^+)$ and $\overline{R}(G^-)$.

We now define $pd^+ := b(\overline{R};G^+;CT-t)$ and

$$pd^- := b(\overline{R};G^-;CT-t),\ 0 \le T \le CT.$$

Then obviously pd^+ and pd^- are monotonic decreasing and increasing, respectively, and $pd^+ + pd^- \le M$.

Applying the level-by-level schedule ([11]) with pd^+ and pd^- processors to $\overline{R}(G^+)$ and $\overline{R}(G^-)$, respectively, shows that

$A^n(G^-;pd^-;CT) \neq \phi$ and $A^n(G^+;pd^+;CT) \neq \phi$.

q.e.d.

Corollary 1: Let the assumptions of Theorem 1 be true.
Then pd^+ and pd^- from Theorem 1 can be chosen such that at least one of the inequalities $pd(G^+;CT) \cdot pd(G^-;CT) \neq 0$ and $pd(G^+;0) \cdot pd(G^-;0) \neq 0$ holds.

Proof: Let $\overline{R}(G)$ be the (M,CT)-stripe representation of G constructed for the proof of Theorem 1. Let $b(\overline{R};G^+;CT) \cdot b(\overline{R};G^-;CT) = 0$ and $b(\overline{R};G^+;0) \cdot b(R;G^-;0) = 0$ (otherwise no further proof is needed).

Apply the exchanging procedure described above to a highest task T in $\overline{R}(G^-)$ and lowest task T' in $\overline{R}(G^+)$ such that T is moved up and T' is moved down and such that the resulting (M,CT)-stripe representations of G^- and G^+ are monotonic again. Repeat this step as long as necessary until the assertion becomes true (see [25]). q.e.d.

It seems to be not difficult to generalize Theorem 1 to an arbitrary FAWD G whose underlying undirected graph is acyclic. In this case the problem arises to determine an appropriate decomposition of G into a maximal anti-forest G^- and its associated forest G^+; this latter problem disappears if G is assumed to be a basic FAWD. But difficulties arise if one attempts to extend the above notions of G^- and G^+ such that Theorem 1 holds for the case that G's underlying undirected graph contains cycles (see example in figure 4).

Given an elementary FAWD G and an arbitrary pd we often will make use of the so called 'highest task first'-schedule, $S_{HTF}(G)$, for selecting free tasks for assignment to the pd processors while processing G. This schedule was investigated first for G being a tree in [1] and for G being an anti-forest in [23]; in both cases pd = M was assumed.

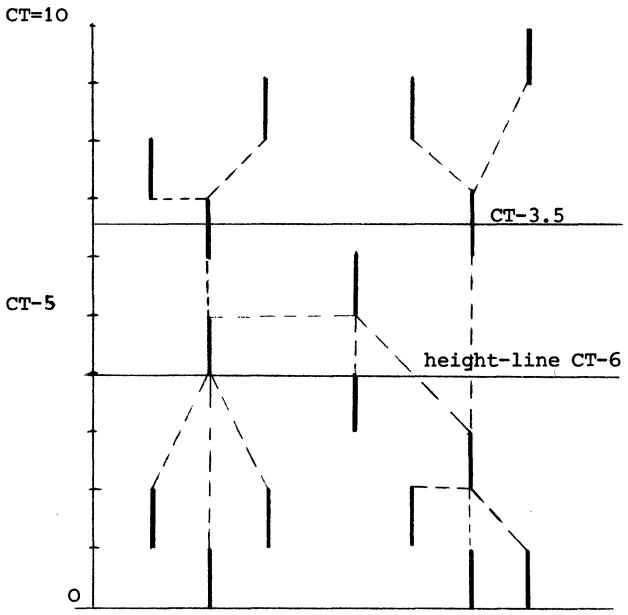


Figure 2a

The stripe representation R(G) of G. The lines represent the nodes, the weights of which determine the lengths of the lines (in this case $W_i = 1$, $1 \le i \le N$). The precedence rules in G are shown by the dashed lines; in this example b(R,G,CT-3.5) = 2 and b(R,G,CT-6) = 1.

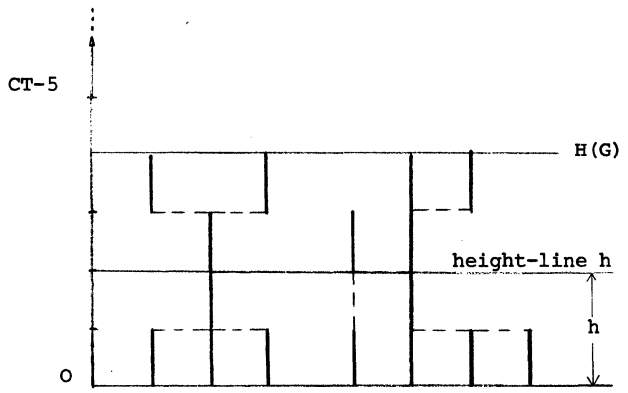


Figure 2b

The special stripe representation $R^L(G)$ of G. The lines are placed as low as possible; in this stripe representation $b(R^L,G,CT-6) = 4$ and $b(R^L,G,CT-8) = 2$.

Lemma 2: Let G be an arbitrary 1-tree or 1-anti-tree. Let CT > 0 and pd be an arbitrary processor distribution for G. Then the implication holds:

$$S_{HTF}(G) \notin \underline{A}^n(G;pd;CT) \Rightarrow \underline{A}^n(G;pd;CT) = \phi$$

The proof of this Lemma is an elementary modification of arguments used in [10,11,22,23], and is therefore omitted here.

Theorem 2: Let G be an arbitrary elementary FAWD and M > 0. Then $CT_{min}(G;M)$ can be computed by testing (at most) N^M different processor-distributions, i.e.: $CT_{min}(G;M)$ can be computed by an algorithm the number of steps of which is bounded by const · M · N^{M+3}.

Proof: Due to Theorem 1 we may restrict ourselves to monotonic processor-distributions, the total number of which is N^M.
For a given processor distribution pd^+ for G^+ we define $pd^- := M - pd^+$. By applying the S_{HTF}-schedule to G^+ and G^- (with pd^+ and pd^- processors, respectively) and using Lemma 2 it can be decided in at most const · M · N^2 steps, whether $\underline{A}^n(G^+;pd^+;CT) \neq \phi$ and $\underline{A}^n(G^-;pd^-;CT) \neq \phi$. In order to find the smallest such CT at most N repetitions of the whole procedure are required. q.e.d.

Remarks:

1) Note that without Theorem 1 it would have been necessary to test $(2^N)^M$ different processor-distributions instead of N^M.

2) The restriction of G to be an elementary FAWD is sufficient but not necessary for validity of Theorem 2. The restriction allows us to use the simple S_{HTF} for deciding the question, whether for an arbitrary given 1-tree or 1-anti-tree G', CT > 0 and pd the set $\underline{A}^n(G';pd;CT)$ is nonempty. If we omit this restriction totally, no effective algorithm is known at present to decide the same question for the resulting more general case. The authors will give an investigation of this problem elsewhere and hope to be able to derive polynomial bounded algorithms to solve the more general problem.

3) Obviously we used extremely crude bounds. These bounds can substantially be improved by taking into account the structure of the graph investigated (see [25]).

Let G be an elementary FAWD, CT an integer, pd(G;CT-t) a processor distribution for G and CT. Then the triple (G;pd;CT), as well as all its components, are called admissible iff $\underline{A}^n(G;pd;CT) \neq \phi$. For given G and CT let $\underline{PD}(G;CT)$ denote the set of all admissible pd's. Let G^+ and G^- be G's underlying 1-tree and 1-anti-tree, respectively; let $pd^+ \in \underline{PD}(G^+;CT-t)$ and $pd^- \in \underline{PD}(G^-;CT-t)$ and let $\overline{pd}(G;CT-t)$ denote the processor distribution for G with $\overline{pd}(G;CT-t) := pd^+(G^+;CT-t) + pd^-(G^-;CT-t)$, $0 \leq t \leq CT$, which only allows pd^+ processors for G^+ and pd^- processors for G^- at any time t, $0 \leq t \leq CT$. Let (G;pd;CT) be an arbitrary admissible triple; then an assignment X of at most

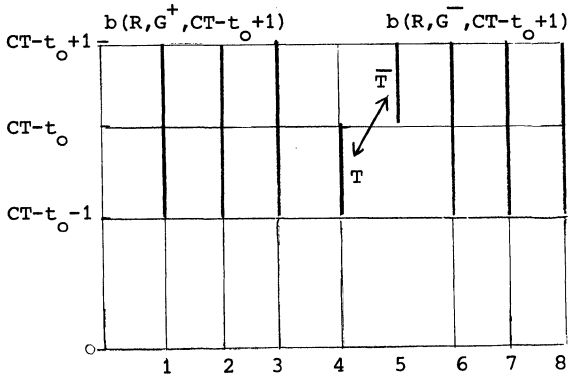


Figure 3

Situation before exchanging

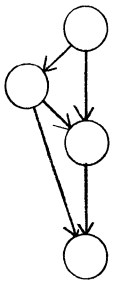


Figure 4

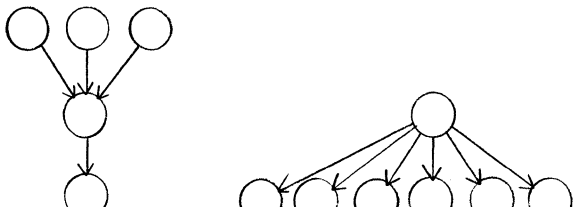


Figure 5

An example for the case M = 3.

pd(G;CT) processors to free tasks of G is called admissible iff the triple $(G\backslash\underline{T}(X);pd;CT-1)$ is admissible, where $\underline{T}(X)$ denotes the set of tasks from G assigned by X; let $\underline{X}(G;pd;CT)$ denote the set of all admissible assignments X for (G;pd;CT).

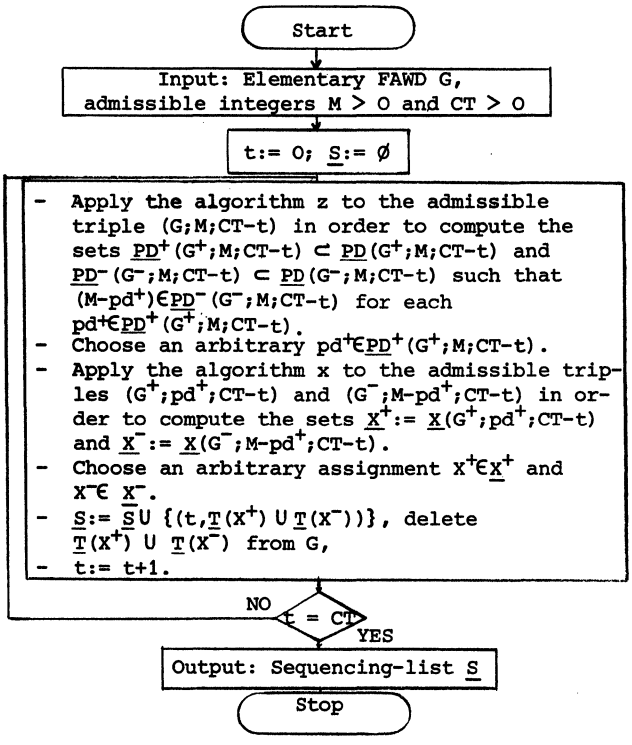
Theorem 3: Let G be an elementary FAWD and let G^+ and G^- be its underlying 1-tree and 1-antitree; let CT > 0. Let $pd^+(G^+;CT-t)$ be an arbitrary processor distribution for G^+. Let G^+ and G^- be processed with pd^+ and $M-pd^+$ processors, respectively, both according to S_{HTF}.
Then $\overline{pd}$ is not admissible if G is not processed completely after time CT.

The proof of Theorem 3 is an elementary application of Lemma 2. Remember that the complexity of the S_{HTF}- scheduling algorithm is bounded by const $\cdot$ M $\cdot$ N^2. Note also that S_{HTF} for M processors applied to an elementary FAWD G (omitting the processor distribution prescription) need not imply complete processing of G in CT time units (see figure 5).

We will now explain the form of the solution to the problem of describing $\underline{A}^n(G;M;CT)$ that one would like to get and that one we are able to derive at present.

For an arbitrary given admissible triple (G;M;CT), where G is an elementary FAWD, we give a scheduling scheme ρ from that all schedules from $\underline{A}^n(G;M;CT)$ could be derived (by appropriate interpretation of this scheme) provided that we can find suitable algorithms x and z.

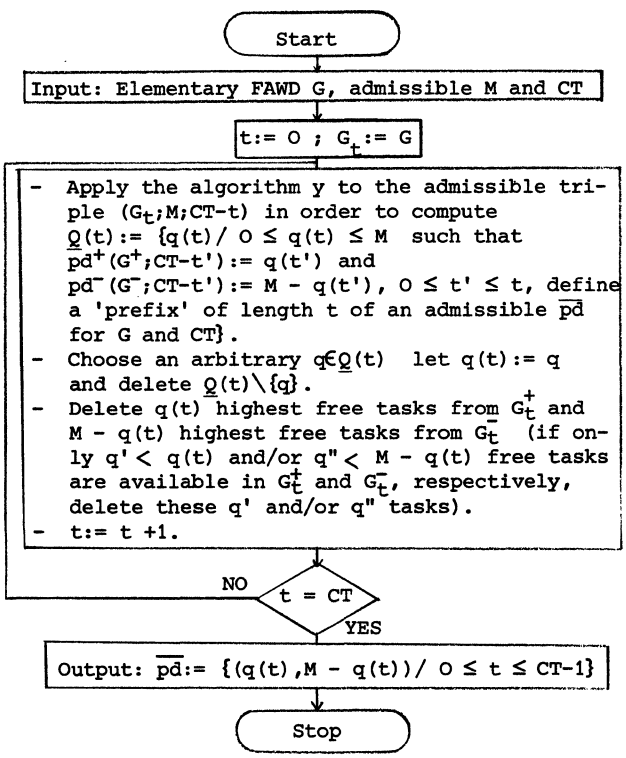
SCHEMA ρ



We first note that scheme ρ becomes a scheduling algorithm as soon as it is interpreted, i. e. a rule is added, how to choose $pd^+\in\underline{PD}^+$, $X^+\in\underline{X}^+$ and $X^-\in\underline{X}^-$. The algorithms z and x are not affected by this specification. Second we see that scheme ρ would provide us with the most general solution to the scheduling problem in this case At the beginning of each time interval 0,1,2,...,CT-1, the algorithm z first shows us what possibilities exist for the choice of admissible pd's. After having chosen a suitable pd^+, algorithm x tells us what possibilities exist for the choice of X that are compatible with the already fixed $\overline{pd}$.

As the investigations concerned with an algorithm z are quite elaborate ([25]), another scheme σ for describing the set $\underline{A}^n(G;M;CT)$ for an arbitrary admissible (G;M;CT) is presented. Compared to the above scheme ρ this new scheme σ will not contain the algorithm z but an algorithm scheme π which describes the set $\underline{PD}^+(G^+;M;CT-t)$ and therefore $\underline{PD}^-(G^-;M;CT-t)$, too. We give this scheme π first. As we are interested mainly in polynomial boundedness we can afford to construct a simple π.

SCHEMA π



Note that for algorithm y we can use a simple modification of an algorithm y' to compute $CT_{min}(G;M)$ according Theorem 2; then y is bounded by const $\cdot$ $M^2 \cdot N^{M+3}$.

Theorem 4: Let (G;M;CT) be an admissible triple, where G is an elementary FAWD. Then the algorithm scheme π describes the set of all admissible pd's for the triple (G;M;CT).

Again we do not give a formal proof but note that

- each interpretation of π leads to an admissible $\overline{pd}$ for $\underline{(G;M;CT)}$,
- each admissible pd for (G;M;CT) can be obtained from π by an appropriate interpretation (defined by this $\overline{pd}$).

For the rest of the paper we are mainly concerned with the definition and investigation of the algorithm x from ρ. We will define x only for the case that G is a 1-tree G^+; the case that G is a 1-anti-tree can be treated similarly. Let us remember that x is to be a polynomial bounded algorithm the application of which to an admissible triple $(G^+;pd;CT)$ provides us with the set $\underline{X}(G^+;pd;CT)$ of all admissible assignments. This set $\underline{X}$ will essentially be defined by a 'lowest assignment function $\overline{X}_o$' which has the property that an arbitrary X belongs to $\underline{X}$ iff X's 'associated' assignment function $\overline{X}$ is 'higher' than $\overline{X}_o$. Explicitly this means: $\overline{X}_o$ is a total function from $\{1,2,\ldots H(G)\} \to \{0,1,\ldots,k\}$ such that $\sum_{i=1}^{H(G)} \overline{X}_o(i) = k$, where k is defined by x; for an arbitrary assignment X its <u>associated assignment function</u> $\overline{X}$: $\{1,2,\ldots,H(G)\} \to \{0,1,\ldots,k\}$ is total and defined by $\overline{X}(i)$ = number of processors assigned to tasks of G^+ starting on height-line i, $1 \le i \le H(G)$. $\overline{X}$ is <u>higher</u> than $\overline{X}_o$ iff $\sum_{i=0}^{j} \overline{X}(H(G)-i) \ge \sum_{i=0}^{j} \overline{X}_o(H(G)-i)$ for all $j = 0,1,\ldots,H(G) - 1$.

<u>Definition of algorithm x</u>

Let an arbitrary admissible assignment $(G^+;pd;CT)$ be given, where G^+ is a 1-tree in $R^L(G^+)$ representation.

1) Determine the maximal number k' of processors that might be left idling by an admissible assignment $X \in \underline{X}(G^+;pd;CT)$. (This k' can be computed by applying the S_{HTF} for 1-anti-tree to the 'inversed' of G^+, see [23]). Let $k := pd(G^+;CT) - k'$. <u>If</u> k = 0 <u>then</u> Stop; (because an arbitrary assignment of k" processors, $0 \le k" \le pd(G^+;CT)$, to arbitrary tasks of G^+ is in $\underline{X}$). In this case therefore algorithm x ends here.

We now determine $\overline{X}_o$ for k processors. Let initially $\overline{X}_o(i) := 0$, $1 \le i \le H(G)$. Let i = 1, $G_1^+ := G^+$ and h(i) := 1.

2) <u>If</u> i > k <u>then</u> Stop; i.e. in the case k > 0 algorithm x ends here.
<u>else</u>
<u>begin</u>
L: <u>if</u> check $(G_i^+;CT-1;h(i))$ <u>then</u>
<u>begin</u> $\overline{X}_o(h(i)) := \overline{X}_o(h(i)) + 1$;
i := i+1;
delete one task from G_{i-1}^+ on height-line h(i-1) and call the result G_i^+;
goto 2;
<u>end</u>; <u>else</u>
<u>begin</u> h(i) := h(i) + 1;
goto L;
<u>end</u>;
<u>end</u>;

The boolean procedure check $(G_i^+;CT-1;h(i))$ returns the value true iff

a) in $R^L(G_i^+)$ there is a free task T on height-line h(i) and

b) deletion of task T and the highest k-i free tasks from G_i^+ results in a graph $G^{+'}$ such that $(G^{+'};pd;CT-1)$ is an admissible triple.

End of description of algorithm x.

<u>Theorem 5:</u> Let G^+ be a 1-tree, let the triple $(G^+;pd;CT)$ be admissible and let the algorithm x, and the result k and $\overline{X}_o$ of its application to $(G^+;pd;CT)$ be as defined above. Finally let X be an arbitrary assignment of k processors to free tasks from G^+ and $\overline{X}$ its associated assignment function. Then the following implication holds:
$X \in \underline{X}(G^+;pd;CT) \Leftrightarrow \overline{X}$ is higher than $\overline{X}_o$.

<u>Remark:</u> If $|\underline{T}(X)| > k$ then only k of the tasks assigned by X are subject to the above constraint.

The proof of Theorem 5 is elaborate and voluminous; therefore only its main ideas are characterized by listing the Lemmas involved. A complete proof is given in [25].

<u>Lemma 3:</u> Let $(G^+;pd;CT)$ be admissible, let $X \in \underline{X}(G^+;pd;CT)$ and let X' be an arbitrary assignment with $|\underline{T}(X')| \ge |\underline{T}(X)|$.
The following implication holds:
$\overline{X}'$ is higher than $\overline{X} \Rightarrow X' \in \underline{X}(G^+;pd;CT)$.

As X_o defined by algorithm x is admissible by construction, the Lemma 3 assures that all assignments 'lying above' X_o are admissible, too; i.e. Lemma 3 proves $\Leftarrow$ from Theorem 5.

<u>Lemma 4:</u> Let the assumptions of Theorem 5 be true and let $X \in \underline{X}(G^+;pd;CT)$. Then

a) if $\overline{X}_o(i) = 0$, $0 \le i \le i' \Rightarrow \overline{X}(i) = 0$, $0 \le i \le i'$.

b) if $\overline{X}(i) = \overline{X}_o(i)$ $0 \le i \le i' \Rightarrow \overline{X}(i'+1) \ge \overline{X}_o(i'+1)$.

By Lemma 4 the monotonic increase of h(i) in algorithm x is justified.

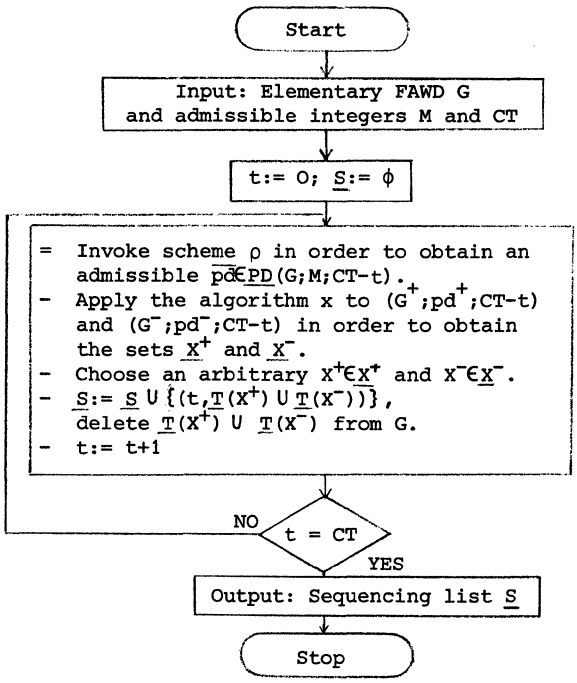
<u>Lemma 5:</u> Let $(G^+;pd;CT)$ be an admissible triple. Let X be an arbitrary assignment such that there exist i', i", $0 \le i' < i'+2 \le i" \le H(G)$, for which $\overline{X}(i) \le \overline{X}_o(i)$, $0 \le i \le i'$ and $i" \le i \le H(G)$ and $\overline{X}(i) > \overline{X}_o(i)$, $i' < i < i"$. Then $X \notin \underline{X}(G^+;pd;CT)$.

By the last Lemma the uniqueness of X_o is established. The proof of implication $\Rightarrow$ from Theorem 5 follows essentially from Lemma 4 and Lemma 5. Note finally that the complexity of the algorithm x is bounded by const $\cdot$ (M+N) $\cdot$ $M^2 \cdot N^{M+3}$.

We summarize the results of this paper in

<u>Theorem 6:</u> Let (G;M;CT) be an admissible triple, where G is an elementary FAWD. Then scheme σ, defined by the diagram

SCHEMA σ



characterizes the set $\underline{A}^n(G^+;M;CT)$. All algorithms involved are polynomial bounded.

References

[1] HU, T. C.: Parallel Sequencing and Assembly Line Problems, Operations Research 9, No.6 (1961), 841-848.

[2] MUNTZ, R. r. and COFFMAN, E. G.: Preemptive Scheduling for Real-Time Tasks on Multiprocessor Systems, JACM, Vol.17, No.2 (April 70) 324-338.

[3] SCHINDLER, S.: On Optimal Schedules for Multiprocessor Systems, Princeton Conference on Information Sciences and Systems (March 72).

[4] FLYNN, M. J.: Some Computer Organisations and Their Effectiveness, IEEE Transactions on Computers, Vol.C 21, No.9 (Sept.72)

[5] MOWLE, F. J. and JULIUSSEN, J. E.: Multiple Microprocessors with a Common Microprogram Memory, Princeton Conference on Information Sciences and Systems , (March 72).

[6] M. FUJII, T. KASAMI, K. NINOMIYA: Optimal Sequence of Two Eyuivalent Processors, SIAM Journal of Applied Mathematics, 17, No.3 (1969), 784-789.

[7] ———————————: Erratum, SIAM Journal of Applied Mathematics 20, No.1 (1971), 141.

[8] COFFMANN, E. G. and GRAHAM, R. L.: Optimal Scheduling for Two-Processor Systems, Acta Informatica 2 (1972).

[9] BARSKIY, A. B.: Minimizing the Number of Computing Devices Needed to Realize a Computational Process within a Specified Time, Engineering Cybernetics, No.6 (1968).

[10] SCHINDLER, S.: Classes of Optimal Schedules for Multiprocessor Systems, 2. Jahrestagung der Gesellschaft für Informatik, Karlsruhe, (Oct. 1972).

[11] SCHINDLER, S.: Quantitative Aspects of Optimal Schedules for Multiprocessor Systems, Workshop on Parallel Computation, Seattle, (June 1972).

[12] LIU, C.L.: Optimal Scheduling on Multiprocessor Computing Systems, Conference on Switching and Automata Theory, Maryland, (Oct. 1972).

[13] FERNANDEZ, E. B. and BUSSEL, B.: Bounds on the Number of Processors in Parallel Computation, Workshop on Parallel Computation, Seattle, (June 1972).

[14] GRAHAM, R. L.: Bounds on Multiprocessing Timing Anomalies, SIAM Journal of Applied Mathematics, Vol. 17, No.2 (March 1969).

[15] C. V. RAMAMOORTHY, K. M. CHANDY, M. J. GONZALES: Optimal Scheduling Strategies in a Multiprocessor System, IEEE Transactions on Computers, Vol. C-21, No.2 (Feb. 1972).

[16] STONE, H. S.: Problems of Parallel Computation, Symp.on Complexity of Sequential and Parallel Numerical Algorithms, Carnegie-Mellon University, Pittsburgh, (May 1973).

[17] FLYNN, M. J. et al.: A Multiple Instruction Stream Processor with Shared Resources, in: Parallel Processor Systems, Technologies and Applications, Spartan Books, (1970).

[18] KUCK, J. K., MURAOKA, Y., CHEN, S. S.: On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup, IEEE Transactions on Computers, Vol. C-21, No.12, (Dec. 1972).

[19] KARP, R. M.: Reducibility Among Combinatorial Problems,Technical Report 3, Department of Computer Science, University of California at Berkeley,(April 1972).

[20] ULLMAN, J. D.: Polynomial Complete Scheduling Problems, Technical Report 9, Dept. of Computer Science, University of California at Berkeley, (March 1973).

[21] SCHINDLER, S. and SIMONSMEIER, W.: The Class of All Optimal Schedules for Two-Processor Systems, Princeton Conference on Information Sciences and Systems, (March 1973).

[22] SCHINDLER, S.: The Complexity of Scheduling Algorithms for Multiprocessor Systems, Symposium on Complexity of Sequential and Parallel Numerical Algorithms, Carnegie-Mellon University, Pittsburgh, (May 1973).

[23] ———————— : Scheduling a Multiprocessor Systems on Anti-Forests, SIAM-SIGNUM Fall Meeting, Austin (Oct. 1972).

[24] MUNTZ, R. R. and COFFMAN, E. G.: Optimal Preemptive Scheduling on Two-Processor Systems, IEEE Transactions on Computers, C-18, No.11, (Nov. 1969).

[25] LÜDTKE, H. and SCHINDLER, S.: Scheduling Multiprocessor Systems on Finite Acyclic Weighted Directed Graphs, Technical Report, to be published.

[26] SCHINDLER, S. and LÜDTKE, H.: Polynomial Bounded Preemptive Schedules for Multiprocessor Systems, to be published.

A SCHEDULING MODEL FOR COMPUTER SYSTEMS WITH TWO CLASSES OF PROCESSORS

R. E. Buten
V. Y. Shen
Computer Sciences Department
Purdue University
Lafayette, Indiana 47907

Abstract -- This paper describes a simple algorithm to schedule a restricted set of jobs on a multiprocessor system with two classes of processors. Through deterministic analysis an upper bound is established for the behavior of the algorithm. This bound is seen to compare favorably with the upper bound intrinsic to the model. Simulation results show the algorithm to be useful in scheduling less restricted job sets.

1. INTRODUCTION

In recent years computing systems have been routinely called upon to support a variety of on-line services in addition to carrying an ever increasing computing load. One major mainframe manufacturer's response to these divergent needs is a multiprocessor system composed of two types of processors.

One class or type of processor is primarily designed to perform floating-point arithmetic operations very efficiently. Accordingly, absent from its instruction set are many functions required by general purpose usage. Notable among these are character oriented and I/O instructions. Let the type and number of this kind of processor be designated as A and m respectively.

The other class of processor is equipped with a very low level instruction set. Character operations if not elegant are at least straightforward. Its significant aspect, however, is the ability of this class of processor to perform I/O. In like manner, let the type and number of this kind of processor be designated as B and n, respectively. Any job run on such a system will necessarily require both kinds of resource.

The resource requests of a job may be represented by a weighted, directed, and acyclic graph as shown in Figure 1. The graph is called the resource request graph of a job. This graph completely specifies the resource requests on the two types of processors and their precedence relations. As indicated in the graph, the two types of requests are made to processors A and B, respectively.

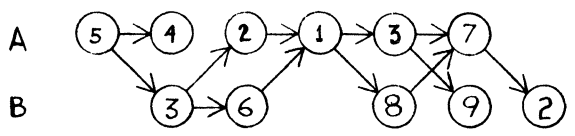


Figure 1
Resource request graph for a typical job.

Given a collection of such jobs it is the function of the scheduling algorithm to assign tasks (nodes in the resourse request graph) to available processors. A task is the basic unit of allocation, i.e. once begun on a processor, it executes without interruption to completion. This is to say, consideration will be restricted to non-preemptive scheduling algorithms.

The performance of a scheduling algorithm may be measured in several ways. Some of these are: mean throughput, average response time, and deadline compliance. The measure used in this paper, however, shall be the amount of time needed to complete the entire set of jobs. An optimal schedule, therefore, is one in which the entire set of jobs is completed in the minimal time.

It is generally acknowledged, based on analysis of similar models, that the generation of optimal schedules for such a general problem requires an exponential number of steps. If a problem of this nature were to have very large nodes, then the benefits derivable from the optimal schedule might very well justify a branch-and-bound approach, or perhaps even an exhaustive search. Since this is not the case under consideration, the hope for problems of this type lies in the development of simple heuristic algorithms which will produce optimal or near optimal results for the models in question. Where the models themselves defy analysis, it may be useful to develop heuristics which apply to simplified subsets. This approach seems to be the underlying motivation for work done on several similar models.

T. C. Hu obtained results scheduling a tree of equal length nodes on a system of n identical processors [5]. Fujii, et al [2,3] and Coffman and Graham [1] have treated arbitrary acyclic graphs composed of equal length nodes and achieved optimal results for two processors of equal ability.

Optimal results were also achieved with a simple algorithm for flow-shop jobs run on two machines [7]. Extensions were made to this result which produced optimal schedules for two machines and all jobs with resource graphs of two nodes [6].

This research was supported in part by the Atomic Energy Commission.

The possibilities are shown below:

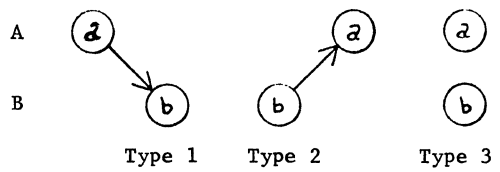


Figure 2
Resource graphs treated by Jackson.

Graham [4] points out the existence of, and bounds the anomalous behavior of acyclic graphs executed on n identical processors when demand scheduling is used. Simulation results presented by Manacher [8] show the occurrence of anomalous behavior to be a common occurrence. This leads one to suspect that the bounds achieved for such cases may be a valid indicator of expected behavior.

Shen and Chen [9] have achieved bounds for a multiprocessor system with two classes of processors when the job set is a restricted class of flow-shop jobs. This suggests a valid starting point for this analysis: the unrestricted flow shop.

2. Analysis of the Flow-Shop Model

Definition of the Model

Let S denote a system composed of m processors of type A, and n processors of type B. Let $F = \{f_1, f_2, \ldots, f_{r-1}, f_r\}$ be a set of flow-shop jobs. Each job in F is represented by a two-tuple, (a_i, b_i), where a_i = A-processor request and b_i = B-processor request, and a_i must precede b_i. That is to say, F is the set of jobs of type 1.

Algorithm

Johnson's optimal solution [7] is for the special case of the model for which m=n=1. His strategy consisted of ordering the jobs according to the following simple criterion:

$$f_i \text{ procedes } f_j \text{ if: } \min(a_i, b_j) < \min(a_j, b_i)$$

A job set in which all its members have been sorted by the above criterion is said to follow a Johnson Order (JO). The optimality of the resulting schedules was shown by proving that the ordering minimizes the wait time on the B-processor. Furthermore, the existence of ties, when the left and right side of the relation are equal, indicates the non-uniqueness of the optimal schedule.

The ordering as it stands is unsuitable for use on multiprocessor systems since it fails to take account of the number of processors of each type. One would like to measure the impact of a node on the total resources of the system. This suggests a modification to the Johnson ordering as follows:

$$f_i \text{ precedes } f_j \text{ if: } \min\left(\frac{a_i}{m}, \frac{b_j}{n}\right) < \min\left(\frac{a_j}{m}, \frac{b_i}{n}\right)$$

in case of equality, largest f first. This Modified Johnson Ordering (MJO) has a comfortable intuitive feeling since one is obtaining the optimal schedule on a Johnson machine of equivalent power. Denote this system as S'. S' has a single A'-processor of speed=m·speed(A), and a single B'-processor with speed=n·speed(B). It is also noteworthy that MJO is a generalization of JO, in that they are identical for m=n=1.

One would hope that the demand schedule resulting from the MJO is optimal on S as well. The following example shows such a case.

Example 1: Let m=n=2 and F = {(9,0),(9,0), (1,5),(1,5),(1,10)}

A demand schedule for the job list ordered as given results in a worst case schedule, T_L.

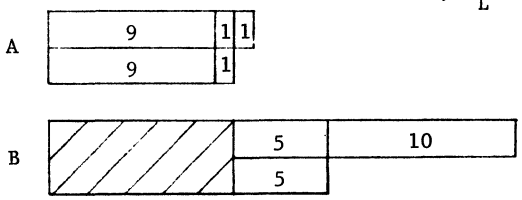


Figure 3

Worst case schedule for example 1.

MJO causes the jobs to be executed in the reverse order of their appearance in F. This produces the optimal schedule for this set, shown below

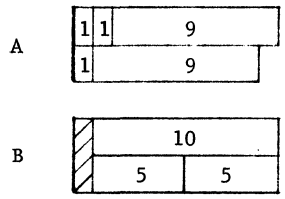


Figure 4
Optimal schedule for example 1.

Optimal schedules, however, are not always produced by MJO. The next example shows MJO generating a schedule which is much larger than optimal.

Example 2: Let m=n=2 and F={(1,5),(1,5),(2,10)} . The jobs as listed are sorted by MJO. The demand schedule, which is actually a worst case, is shown below.

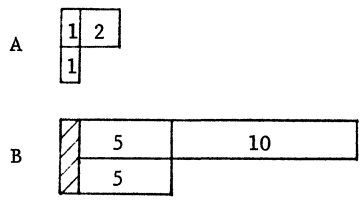


Figure 5
Worst case schedule for example 2.

Reversing the job list results in a better schedule:

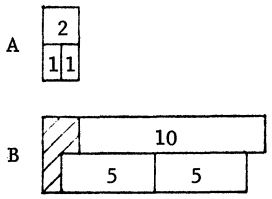


Figure 6
Optimal schedule for example 2.

These examples prompt two questions regarding MJO. How much of a saving can MJO produce? And, how badly can it perform?

Bounds

In deriving the MJO algorithm for S, we made intuitive use of S', a Johnson Machine of equivalent power. Perhaps it would be instructive to further compare these two systems in order to determine the performance of MJO. The requests for resource usage on S was represented by a two-tuple, (a_i,b_i). This same job, when executed on S' requires resource usage of (a'_i,b'_i), where

$$a'_i = \frac{a_i}{m} \quad , \quad b'_i = \frac{b_i}{n} \quad . \qquad (1)$$

Since S' has the equivalent power of S and is of simpler structure, one would intuitively expect that it could perform the same work load as S.
Formalizing this expectation we have:
Lemma 2.1: Given a schedule for a job set F, on a system S whose completion time is T_e, then there exists a schedule on S' with a completion time of T'_e such that

$$T'_e \leq T_e \quad .$$

Proof: The proof is by construction of the required schedule on S'.

As a first step, one considers a simulation by S' of the schedule yielding T_e on S. To accomplish this we divide the schedule into unit time slices. For each time slice on S from 1 to T_e, let A' execute a unit portion of that task executing on each of the m A-processors, while B' executes a unit portion of that task executing on each of the n B-processors. Since speed (A') =m•speed(A) and speed(B')=n•speed(B), it is clear that S' can keep up with the progress of S. Let a time slice of a_i or b_i executed on S' be denoted by a^*_i and b^*_i respectively, where

$$a^*_i = \frac{a_i}{mT_e} \quad , \quad b^*_i = \frac{b_i}{nT_e} \quad . \qquad (2)$$

A time slice in which no task is executing is said to be executing the idle task, denoted by x.

Define

Ta_i = time of the last occurrence of a^*_i

Tb_i = time of the first occurrence of b^*_i

The second step is to rearrange this simulated preemptive schedule into a non-preemptive permutation schedule. The following procedure is used:

Step 1. Sort the list of Ta's in ascending order. Do the same for the list of Tb's.

Step 2. Relabel the time slices as follows:

$a^*_i \rightarrow a^\dagger_j$ where j is the rank of Ta_i in the sorted list of Ta's

$b^*_i \rightarrow b^\dagger_j$ where j is the rank of Tb_i in the sorted list of Tb's

Step 3. Apply the following interchange rules until no further interchanges are possible, i.e. until all slices of each task are juxtaposed.

Rule 1. If $a^\dagger_i$ immediately precedes $a^\dagger_j$ and i >j, or if x immediately precedes $a^\dagger_j$, then interchange the two.

Rule 2. If $b^\dagger_i$ immediately precedes $b^\dagger_j$ and i > j or if $b^\dagger_i$ immediately precedes x, then interchange the two.

Rule 1 has the property that no Ta_i is ever increased, i.e. the completion time of no A task is delayed by the use of Rule 1. Rule 2 has the complementary property in which no Tb_i is ever decreased, meaning that all precedence constraints between A- and B- tasks are preserved.

After all time slices for each task are juxtaposed, the third step is to replace the time slice notation with the job notation. The order of appearance of these jobs on A' is a permutation of F which will produce this schedule or one better. The schedule thus constructed is a permutation schedule since the order of completion of A-tasks is the same as the order of the initiation of B-tasks, which for S' is the same as the order of initiation of A-tasks.

Since there could exist wait time between the completion of an A task and the initiation of its associated B task, the schedule produced thus far is not necessarily a demand schedule. Therefore, the final step is to convert that schedule to a demand schedule by advancing the start of all b_i's until either the time their associated A process completed or the completion of the previous B-task, whichever is greater.

The following example will serve to clarify the proof as well as to show the need for such a

mechanism to guarantee the inequality.

Example 3: Let m=n=3, and $F=\{f_1,f_2,f_3,f_4,f_5\}$ where $f_1=f_2=(4,0), f_3=(1,3), f_4=(1,2), f_5=(1,1)$

The schedule producing T_e is shown below

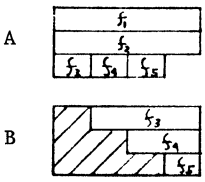


Figure 7
Optimal schedule for F on S.

The first step produced a simulated schedule on S' as follows:

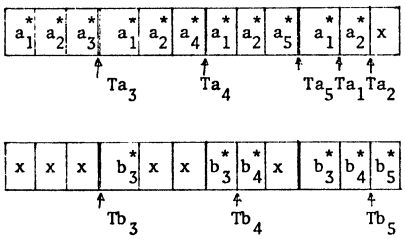


Figure 8
Simulation of T_e by S'

The second step first sorts the lists of times:

Index	Ta's	Tb's
1	Ta_3	Tb_3
2	Ta_4	Tb_4
3	Ta_5	Tb_5
4	Ta_1	
5	Ta_2	

then re-labels the the time slices using rank in the sorted lists.

A' $a_4^\dagger\ a_5^\dagger\ a_1^\dagger\ a_4^\dagger\ a_5^\dagger\ a_2^\dagger\ a_4^\dagger\ a_5^\dagger\ a_3^\dagger\ a_4^\dagger\ a_5^\dagger$ x

B' x x x $b_1^\dagger$ x x $b_1^\dagger\ b_2^\dagger$ x $b_1^\dagger\ b_2^\dagger\ b_3^\dagger$

Step three performs all possible interchanges resulting in

A' $a_3^*\ a_4^*\ a_5^*\ a_1^*\ a_1^*\ a_1^*\ a_1^*\ a_2^*\ a_2^*\ a_2^*\ a_2^*$ x

B' x x x x x x $b_3^*\ b_3^*\ b_3^*\ b_4^*\ b_4^*\ b_5^*$

The ordering of F which produces T_e' is now clear, namely f_3,f_4,f_5,f_1,f_2. The transformation by the last step to a demand schedule results in a time $T_e' = 3.67$.

As pointed out in Example 2, MJO may not always produce an optimal schedule on S. Since it does produce the optimal schedule on S', it is of interest to compare the performance of MJO on S to that on S'. A multiprocessor system such as S since its power is based on parallel execution of many jobs, cannot function effectively when severely "underloaded". To obtain a meaningful comparison we would therefore like to discard cases where a single job dominates the schedule. This may be accomplished by requiring the following loading constraint (LC):

$$a_i + b_i \leq T_h' \qquad 1 \leq i \leq r \qquad (3)$$

Lemma 2.2: Given a job set F, subject to LC and ordered by MJO. If T_h and T_h' are the completion times of a demand schedule on S and S' respectively, then

$$\frac{T_h}{T_h'} \leq 2 - \frac{1}{\max(m,n)} .$$

Proof: The proof is by contradiction. We shall assume that there exists a set of jobs which violates the bound. We may further assume that the number of jobs, r, is minimal. That is, b_r is the last B process to terminate on S when a_r is the last job started, i.e. the last job in MJO.

We can make the above assumption because if b_k is the last terminating process and $k < r$, we can consider the truncated set $f_1,f_2,\ldots,f_k$. The completion time on S for the truncated set, P_h, is exactly T_h. On the other hand, the completion time on S' of the truncated set, P_h', is less than or equal to T_h' since there are less jobs in the truncated set. Therefore,

$$\frac{P_h}{P_h'} \geq \frac{T_h}{T_h'} > 2 - \frac{1}{\max(m,n)} ,$$

and the truncated set forms a smaller counter-example to the lemma. We shall therefore consider r to be minimal.

The proof divides into two cases:

Case 1: The last task to complete is on a B processor
Case 2: The last task to complete is on an A processor.

Case 1:

Let Ta_i denote the time a_i completes execution and Tb_i the time b_i begins execution for all i. The proof of this case shall be treated in three subcases.

Case 1a: $Ta_r = Tb_r$

Demand scheduling requires there be no imbedded idle time on the A or A' processors, i.e. the schedule is compact to the left. The latest time at which a_r can begin is immediately after the completion of all other tasks. Thus for s,

$$T_h \leq \frac{1}{m} \sum_{i=1}^{r-1} a_i + a_r + b_r . \qquad (4)$$

Correspondingly the time on S' is given by

$$T'_h \geq \frac{1}{m} \sum_{i=1}^{r} a_i + \frac{b_r}{n} \quad . \tag{5}$$

Dividing (4) by (5) yields

$$\frac{T_h}{T'_h} \leq 1 + \frac{a_r(\frac{m-1}{m}) + b_r(\frac{n-1}{n})}{T'_h} \quad . \tag{6}$$

Since the set forms a counter-example to the lemma

$$2 - \frac{1}{\max(m,n)} < \frac{T_h}{T'_h} \leq 1 + \frac{a_r(\frac{m-1}{m}) + b_r(\frac{n-1}{n})}{T'_h} \quad , \tag{7}$$

which reduces to

$$T'_h \frac{\max(m,n)-1}{\max(m,n)} < (a_r+b_r) \frac{\max(m,n)-1}{\max(m,n)} \quad . \tag{8}$$

And this is a contradiction by the loading constraint (3).

Since consideration is restricted to demand type schedules, the following cases which treat $Tb_r > Ta_r$ necessarily have all B processors busy on the interval $[Ta_r, Tb_r)$, as shown in Figure 9. We use T_k to indicate the end of the last idle period on the B processors prior to Tb_r.

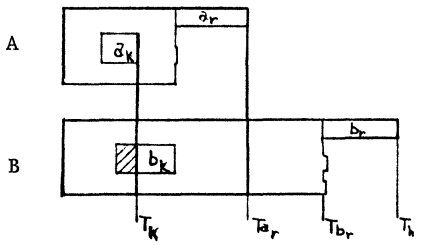


Figure 9

Case 1b: $Tb_r > Ta_r$ & $T_k = 0$

If there is no idle time on the B processors except terminal idle then the latest start time for the last B process is immediately after all other B processes have completed, or

$$T_h \leq \frac{1}{n} \sum_{i=1}^{r-1} b_i + b_r \quad . \tag{9}$$

If there is no initial wait time on the B processors then the first m A processes must be zero. And the time needed on S' is

$$T_h \geq \frac{1}{n} \sum_{i=1}^{r} b_i \quad . \tag{10}$$

The desired ratio is

$$\frac{T_h}{T'_h} \leq 1 + \frac{\frac{n-1}{n} b_r}{T'_h} \quad , \tag{11}$$

which leads to the same contradiction as (6).

Case 1c: $Tb_r > Ta_r$ & $T_k > 0$

From Figure 9, one sees that the schedule for F which produces T_h is first A-bound, and then B-bound. All previous cases dealt exclusively with one class of processors or the other. Therefore, to facilitate treatment of this case one would like to treat separately each group of jobs. Define partitioned job sets as follows:

Job set 1, F^1, contains every job which completes before T_k in addition to the truncated portions of those jobs in execution at that time. Or,

$$F^1 = \{f_i^1 = (a_i^1, b_i^1) \ , \ 1 \leq i \leq r \text{ such that}$$

$$a_i^1 = \max(0, \min(a_i, T_k - Ta_i - a_i))$$

$$b_i^1 = \max(0, \min(b_i, T_k - Tb_i))\} \quad .$$

Job set 2, F^2, contains the remainder of all jobs truncated to form F^1 as well as all jobs which begin execution after T_k. Or

$$F^2 = \{f_i^2 = (a_i^2, b_i^2) \ , \ 1 \leq i \leq r \text{ such that}$$

$$a_i^2 = \max(0, \min(a_i, Ta_i - T_k))$$

$$b_i^2 = \max(0, \min(b_i, Tb_i + b_i - T_k))\} \quad .$$

Each job set has r jobs as before although admittedly many are null. This manner of definition, however, leaves the indices constant. For the treatment of these partitioned job sets to have relevance in bounding the total set, the following relation must be established.

$$T_h \leq T_h^1 + T_h^2 \quad , \tag{12}$$

where T_h^1 and T_h^2 are the completion times of each of the partitioned job sets, sorted by MJO, and executed on S.

The composition of F may be divided into the following subsets according to the makeup of F^2. Define

$$X = \text{set of job indices} \quad a_x^2 = 0 \ , \ x \in X$$

$$Y = \text{set of job indices} \quad a_y^2 = a_y \ , \ y \in Y$$

$$U = \text{set of job indices} \quad 0 < a_u^2 < a_u \ , \ u \in U \quad .$$

Referring to the ordering of the original set F, it is clear that

$$f_u \text{ precedes } f_y \text{ for all } u \in U \text{ and for all } y \in Y \quad . \tag{13}$$

Since the set was subjected to MJO

$$\text{Min}(\frac{a_u}{m}, \frac{b_y}{n}) < \min(\frac{a_y}{m}, \frac{b_u}{n}) \text{ for all } u \in U \text{ and for all } y \in Y \quad . \tag{14}$$

Therefore

$$f_x^2 \text{ precedes } f_u^2 \text{ precedes } f_y^2 \quad \text{for X, U, \& Y} \quad . \tag{15}$$

Since the validity of (14) cannot be altered merely by the reduction of a_u/m when all else remains equal, all f_x, having zero A-requests, precede all else. Therefore, the execution of jobs in F^2 is identical to their execution in F to within a permutation of processor numbers. Therefore,

$$T_h = T_k + T_h^2 \quad . \tag{16}$$

And

$$T_k = \frac{1}{m} \sum_{i=1}^{r} a_i^1 \leq T_h^1 \quad . \tag{17}$$

Substituting (17) into (16) produces

$$T_h \leq T_h^1 + T_h^2 \quad , \tag{18}$$

and the validity of (12) is established.

Consider next the relationship between T_h' and $T_h^{1'} + T_h^{2'}$, the sum of execution times on S' of F^1 and F^2 ordered by MJO. From the definition of this subcase and the description of the partitioning, the execution time of $T_h^{1'}$ is governed by the A-tasks. Thus when $T_h^{1'} + T_h^{2'}$ is considered, no additional time is required for execution of the A-tasks. The only source of increased execution time for the two subsets arises when the execution of all or part of a B task is "held up" due to the construction of the partitioned sets. This can happen in two ways.

First, the "release" of b_i for consideration by B' in the partitioned set is governed by the schedule of the A-tasks on S. If a_i is an initial task on S, the release time of b_i is exactly a_i. If a_i is also chosen to be an initial task on S', then the new release time is a_i/m. Thus it is possible to delay the release of b_i by a maximum of $a_i(\frac{m-1}{m})$ due to the partitioning of the job set. If a_i is not an initial task on S, the earliest possible time to schedule a_i on S' can not be sooner than the schedule time of a_i on S. Therefore for tasks whose ordering are not changed by the partitioning of the job set, the release time of the B-tasks on B' can be delayed up to $a_\ell(\frac{m-1}{m})$, where a_ℓ is the largest A-task.

The second source of delay is an A-task whose ordering is altered by the partitioning itself. There are at most m A-tasks which were altered by the partitioning. Designate this set as U and let a_u be any member of the set. When the job set is executed as a whole, the time at which b_u is released to B' is given by:

$$R_u^\omega = \sum_{i=1}^{u} \frac{a_i}{m} \quad . \tag{19}$$

When F^2 is executed, the members of U may be reordered. The release time for b_u in the sequential execution of the partitioned sets is given by

$$R_u^p \leq \sum_{i=1}^{u-1} \frac{a_i}{m} + \sum_{i \in U} \frac{a_i}{m} \tag{20}$$

$$\leq R_u^\omega + \sum_{\substack{i \in U \\ i \neq u}} \frac{a_i}{m} \quad . \tag{21}$$

Let a_ℓ denote the largest member of U. Then (21) becomes

$$R_u^p \leq R_u^\omega + (\tfrac{m-1}{m})\, a_\ell \quad . \tag{22}$$

Since on S', the B-tasks are executed sequentially, the delay experienced by the last B-task is bounded above by the largest delay previous to it. Thus we have established the desired relation between the execution time of the whole set to that of the two partitioned sets,

$$T_h^{1'} + T_h^{2'} \leq T_h' + (\tfrac{m-1}{m})\, a_\ell \quad . \tag{23}$$

From the construction of the partition we have,

$$T_h^{1'} = T_k \quad . \tag{24}$$

Substituting this into (16)

$$T_h \leq T_h^{1'} + T_h^2 \tag{25}$$

$$\leq T_h^{1'} + \frac{1}{n} \sum_{i=1}^{r} b_i^2 + (\tfrac{n-1}{n})\, b_r \quad . \tag{26}$$

And from S' we have

$$T_h^{2'} \geq \frac{1}{n} \sum_{i=1}^{r} b_i^2 \quad . \tag{27}$$

Substituting this into (26) leaves

$$T_h \leq T_h^{1'} + T_h^{2'} + (\tfrac{n-1}{n})\, b_r \quad . \tag{28}$$

which using (23) reduces to

$$T_h \leq T_h' + (\tfrac{m-1}{m})\, a_\ell + (\tfrac{n-1}{n})\, b_r \quad . \tag{29}$$

The assumption of a minimal set requires $\ell \leq r$ in the MJO, for any A task which is all or in part

contained in F^1. This implies

$$\min\left(\frac{a_\ell}{m}, \frac{b_r}{n}\right) < \min\left(\frac{a_r}{m}, \frac{b_\ell}{n}\right) . \qquad (30)$$

This requires either $b_r \leq b_\ell$ or $a_\ell \leq a_r$

making the appropriate substitution leaves

$$T_h \leq Th'_h + \left(\frac{m-1}{m}\right) a_\ell + \left(\frac{n-1}{n}\right) b_\ell \qquad (31a)$$

or $$T_h \leq Th'_h + \left(\frac{m-1}{m}\right) a_r + \left(\frac{n-1}{n}\right) b_r . \qquad (31b)$$

Dividing (31a) and (31b) by T'_h leaves equations of the same form as (6), which leads to the same contradiction of the loading constraint.

Case 2: An A-process is last to terminate. With the exception that $b_r=0$, this case produces the same equation as (4) and of course leads to the same contradiction.

All cases and subcases treated are seen to lead to a contradiction and thus the lemma is proved.

The two previous lemmas may be used to prove the following theorem which provides a performance bound for MJO.

Theorem 2.1: For a job set F, $\frac{T_h}{T_e} \leq 2 - \frac{1}{\max(m,n)}$

where T_e = completion time of the shortest schedule possible for F on S,

T_h = completion time of the MJO schedule.

Proof: Lemma 2.2 provides

$$\frac{T_h}{T'_h} \leq 2 - \frac{1}{\max(m,n)} .$$

By Johnson's result T'_h is optimal, thus

$$T'_h \leq T_e' .$$

By Lemma 2.1, $T_e' \leq T_e$.

Combining yields

$$\frac{T_h}{T_e} \leq \frac{T_h}{T'_h} \leq 2 - \frac{1}{\max(m,n)} .$$

Q.E.D.

Example 4: Let F = $\left\{\begin{array}{l} n(n-1) \text{ jobs of the form } (\epsilon,1) \\ 1 \text{ job of the form } (2\epsilon,n) \end{array}\right\}$ in a system that m=n.

The MJO schedule is shown in Figure 10:

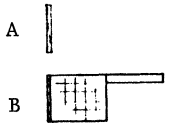


Figure 10

The optimal schedule is shown in Figure 11:

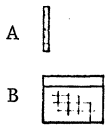


Figure 11

From Figures 10 and 11, we have

$$\frac{T_h}{T_e} = \frac{2n-1+\epsilon}{n+2\epsilon} = 2 - \frac{1+3\epsilon}{n+2\epsilon} \to 2 - \frac{1}{n} .$$

Example 4 shows that the bound of Theorem 2.1 is approachable. If we remove the "largest first rule" to break ties in MJO, which is not used in the proof of Theorem 2.1, the bound may be reached by scheduling n(n-1) jobs of the form (0,1) and one job of the form (0,n).

The following theorem gives the worst case bound for the flow-shop model with two classes of processors.

Theorem 2.2: For a given job set F, where T_e is the earliest completion time possible for a demand schedule on S, the latest completion time, T_ℓ, is given by

$$\frac{T_\ell}{T_e} < 3 - \frac{1}{\max(m,n)} .$$

Proof:

Consider a schedule which produces T_1:

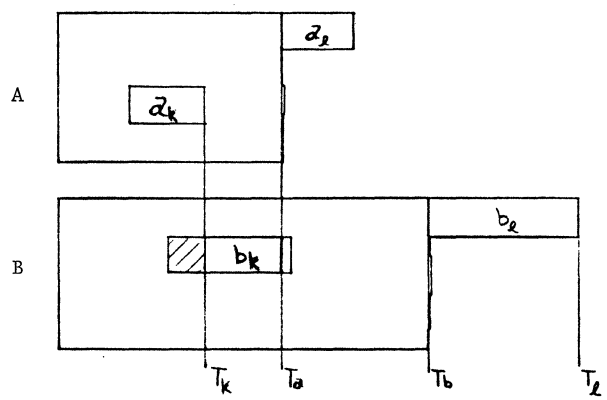


Figure 12

The following notations are used in Figure 12:

b_r = last task to complete

a_r = A-task associated with b_r

T_a = time a_r completes execution

T_b = time b_r begins execution

T_k = time of the last idle time on any B-processor before T_b.

The following are lower bounds on the earliest completion time possible:

$$T_e \geq \frac{1}{m}\sum_{i=1}^{r} a_i, \text{ time needed to execute all A tasks} \quad (32a)$$

$$T_e \geq \frac{1}{n}\sum_{i=1}^{r} b_i, \text{ time needed to execute all B tasks} \quad (32b)$$

$$T_e \geq a_i+b_i \quad 1\leq i\leq r, \text{ time of the largest job} \quad (32c)$$

The proof is divided into three cases.

Case 1: $Ta_r=Tb_r$, i.e. the last B-task to complete begins execution immediately after the completion of its associated A task.

By the constraint of demand scheduling

$$T_\ell \leq \frac{1}{m}\sum_{i=1}^{r} a_i + (\frac{m-1}{m})\, a_\ell + b_\ell \quad , \quad (33)$$

which reduces immediately by (32a) and (32c) to

$$T_\ell \leq 2T_e \quad \text{or} \quad \frac{T_\ell}{T_e} \leq 2 \quad . \quad (34)$$

Case 2: $Ta_r \neq Tb_r$ & $T_k = 0$, i.e. there is no imbedded wait time on a B processor. The time is then given by

$$T_\ell \leq \frac{1}{n}\sum_{i=1}^{r} b_i + (\frac{n-1}{n})\, b_\ell \quad , \quad (35)$$

which also reduces with the application of (32a) and (32c) to

$$T_\ell \leq T_e + (\frac{n-1}{n})\, T_e \quad \text{or} \quad \frac{T_\ell}{T_e} \leq 2 - \frac{1}{n} \quad . \quad (36)$$

Case 3: $T_a \neq T_b$ & $0 < T_k < T_a$

By definition

$$T_\ell = T_b + b_r \quad . \quad (37)$$

And the time that the last B-task starts can be bounded by

$$T_b \leq T_k + \frac{1}{n}\sum_{i=1}^{n-1} b_i \quad . \quad (38)$$

And the time of the last idle time on a B-processor is bounded by

$$T_k < T_a \leq \frac{1}{m}\sum_{i=1}^{r-1} a_i + a_r \quad . \quad (39)$$

Substituting (38) and (39) into (37) gives

$$T_\ell < \frac{1}{m}\sum_{i=1}^{r-1} a_i + \frac{1}{n}\sum_{i=1}^{r-1} b_i + a_r + b_r \quad . \quad (40)$$

Re-arranging,

$$T_\ell < \frac{1}{m}\sum_{i=1}^{r} a_i + \frac{1}{n}\sum_{i=1}^{r} b_i + (a_r+b_r)\,\frac{q-1}{q} \quad , \quad (41)$$

where $q = \max(m,n)$.

Applying (32a), (32b), and (32c) leaves

$$T_\ell < T_e + T_e + \frac{q-1}{q}\, T_e \quad \text{or} \quad \frac{T_\ell}{T_e} < 3-\frac{1}{q} \quad , \quad (42)$$

which is the largest of the case bounds.

Q.E.D.

The following example shows that the bound is approachable for a large set of jobs.

Example 5:

$$\text{Let } F = \left\{\begin{array}{ll} m \text{ jobs of the form} & (n,0) \\ n(n-1) \text{ jobs of the form} & (\in,1) \\ 1 \text{ job of the form} & (\in,n) \end{array}\right\}$$

The demand schedule with the longest completion time, T looks like

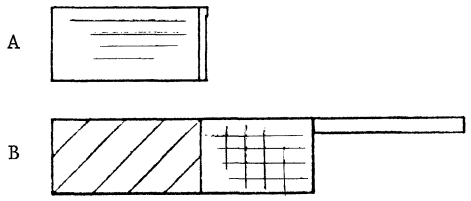


Figure 13

It follows that $T_e = 3n-1+\in(\frac{n}{m})$

The demand schedule with the shortest completion time, T is

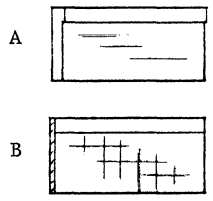


Figure 14

And $T_e = n + \frac{n(n-1)}{m} \in +\in$

$$\frac{T_\ell}{T_e} = \frac{3n-1+\in(n/m)}{n+\frac{n}{m}\in+\in}$$

$$\lim_{\in\to 0} \frac{T_\ell}{T_e} = 3 - \frac{1}{n}$$

Table 1 places this flow-shop result in perspective with known results of simple heuristic algorithms for similar models.

SYSTEM	JOB SET	ALGORITHM	T_L/T_E	T_A/T_E	T_A/T_L
2 processors of different types	Flow-shop	Johnson Ordering	2	1	1/2
n identical processors	independent tasks	largest first	2-1/n	(4/3)-(1/3n)	(4n-1)/(6n-3)
M A's and n B's	Flow-shop	Modified Johnson Ordering	$3-\frac{1}{\max(m,n)}$	$2-\frac{1}{\max(m,n)}$	(2n-1)/(3n-1)

Table 1: Comparison of several known results on similar models

3. Extensions of the Model

In the previous section we dealt exclusively with jobs of type 1. Define $G=\{g_1,g_2,\ldots,g_{s-1},g_s\}$ to be a set of flow-shop jobs of type 2, i.e. the first of two nodes must be executed on a B processor. The MJO criterion can be restated for jobs of type 2:

g_j preceeds g_i if: $\min\left(\frac{a_i}{m},\frac{b_j}{n}\right) < \min\left(\frac{a_j}{m},\frac{b_i}{n}\right)$

in case of equality, largest g first.

From symmetry considerations the bounds for type 1 jobs also apply to type 2 jobs. Jobs of type 3 may be considered as two jobs, a type 1 and a type 2.

It would be of interest to see if MJO thus extended is of value in scheduling jobs with more general resource graphs. To be applicable, the more complex structures must be mapped into the two nodes of the model. This is done by taking the first available node of the resource graph as the first of two in the model. The second node of the model is constructed from the scaled sum of all nodes remaining. It is assumed to belong to the processor opposite to that selected as the first node, in accordance with the constraints of the model. Nodes belonging to the opposite kind of processor are scaled by m/n or n/m, whichever is necessary to convert all nodes to the same dimension.

The pseudo jobs thus created conform to all the constraints of the model. These pseudo jobs are then sorted according to MJO. Assignments to processors are then made on a demand basis. When the first node of any pseudo job, the real one, completes a new pseudo job is computed for all nodes whose execution was precluded by the node now complete. Pseudo jobs are created in this manner until only two real nodes remain for a job, at which time there is no further need of pseudo jobs.

A simulation test of this extension was conducted. The structure of the resource graphs was limited to two parallel paths, one A-request and one B-request. A given node could depend on the previous A-node, B-node, or both. 10,000 jobs were constructed in this fashion with the number of nodes per job being a uniformly distributed random number between n+m and 2mn. The precedence relations of each node were also determined by a random variable. The size of the nodes was also uniformly distributed between zero and the respective number of processors.

The job set was executed with the MJO extension described above, and then again using the task list as generated. The MJO extension was found to provide job times that were on the average 4-8% smaller than produced by the random ordering. It is interesting to note that Manacher [8] quotes 5-15% as the typical savings of a heuristic over random in scheduling tasks on a system of n identical processors.

REFERENCES

1. E. G. Coffman, Jr. and R. L. Graham, "Optimal Scheduling for Two-Processor Systems" Acta Informatica 1 (1972), pp 200-213.
2. M. Fujii, T. Kasami, K. Ninomiya, "Optimal Sequencing of Two Equivalent Processors" SIAM J. Appl. Math. 17, No. 3 (1969), pp 784-789.
3. ----- "Erratum", SIAM J. Appl. Math. 20, No. 1 (1971), pp 141.
4. R. L. Graham, "Bounds on Multiprocessing Timing Anomalies" SIAM J. Appl. Math. 17, No. 2 (March 1969), pp 416-429.
5. T. C. Hu, "Parallel Sequencing and Assembly Line Problems" Oper Res 9, No. 6 (1961), pp 841-848.
6. J. R. Jackson, "An Extension of Johnson's Results on Job-Lot Scheduling" Naval Res. Log. Quart. 3, No. 3 (1956).

   Also, Theory of Scheduling, R. W. Conway, et al., Reading Mass: Addison-Wesley, 1967, Chapter 5.
7. S. M. Johnson, "Optimal Two- and Three-Stage Production Schedules with Setup Times Included" Navel Res. Log. Quart. 1, No. 1 (1954).

   Also, Theory of Scheduling, R. W. Conway, et al., Reading Mass: Addison-Wesley, 1967, Chapter 5.
8. G. K. Manacher, "Production and Stabilization of Real-Time Task Schedules" JACM 14, No. 3 (1967) pp 439-465.
9. V. Y. Shen, Y. E. Chen, "A Scheduling Strategy for the Flow-Shop Problem in a System with Two Classes of Processors" Proc. 6th An. Princeton Conf. on Infor. Science and Systems (1972), pp 645-649.

SCHEDULING IN A MULTIPROCESSOR ENVIRONMENT[a]

J.M. Gwynn and R.J. Raynor
School of Information and Computer Science
Georgia Institute of Technology

Summary

In a multiprocessor system, the handling of interrupts generated by jobs in the processors is assigned to a supervisory program and associated data base. Techniques for deciding which processor executes the supervisor includes master-slave, floating executive control, and others[1]. Regardless of the technique employed, queueing of requests to the supervisor may occur. In a master-slave system, the master processor can handle only one request at a time. In a floating executive system, only one processor can access the supervisor's data base at a time[2].

Madnick has developed a finite-source queueing model which explicitly relates the number of processors in the system to the average number of processors idle due to clustering of requests to the supervisor. As an indication of the severity of the problem, his model predicts that a system with 21 processors will have an average of 2.8 processors idle due to supervisor clustering[2]. Due to the nature of his model, however, this may be a pessimistic estimate.

A resolution to this problem can exist only if the supervisor is not saturated, ie, if the total expected execution time of the supervisor during a given period is not greater than the length of that period. Stated another way, the supervisor will not be saturated if the system is designed such that the supervisor is not a limiting resource. Assuming an unsaturated system, the natural solution to the problem would seem to be to schedule jobs to the processors in such a way that they would cause an interrupt at a time when the supervisor was idle[3]. The assumption implicit in this solution is that, for each job in the system, the time until the next interrupt must be predictable from the job's history of execution. While prediction of this information has not been implemented in many situations, Pass has used a single exponential smoothing formula and corrector which dynamically modifies the smoothing constant at each interrupt with promising success[4]. It will therefore be assumed that this information, as well as the length of time the supervisor requires to handle an interrupt, can be predicted with some degree of accuracy.

The algorithm to implement this solution would be a simple two table search. The first table would have an entry for each ready job in the mix specifying the time until the next interrupt and the supervisor time required to handle that type of interrupt. The other table would be a schedule of supervisor idle periods. For each job in the mix, a decision would be made as to whether the supervisor had an idle period corresponding to the period from current time plus process time to current time plus process time plus supervisor time. A match would cause that job to be scheduled. The order in which the first table is searched may be determined by priority or some other external criteria.

While the algorithm just described is simple, the amount of computation involved would perhaps be prohibitive. For this reason a sub-optimal algorithm was developed which requires much less computation at the price of a small decrease in effectiveness. This algorithm is based on the original but with a discretization of time into blocks of time. Based on the number of comparisons in the search, the sub-optimal algorithm is approximately $2(P+1)/F$ times faster than the optimal one; where P is the number of processors and F is the ratio of average supervisor time to block size. A more important point is that the sub-optimal algorithm would allow a hardware implementation, using only a few special registers, which would reduce the search to a few logical operations.

For F=1, a case in which the hardware implementation would be especially feasible, a GPSS simulation model has predicted that for 21 processors there would be a reduction in average number of idle processors to 0.7, with a corresponding increase in thruput of 12%. While this is 75% of optimal improvement, it is expected that this could be improved, possibly to 90%, thru fine tuning of the algorithm parameters.

Since the mix size is assumed to be large enough to find a job that will interrupt during a supervisor idle period, it is likely that some jobs may be delayed an excessive amount of time. The standard procedure for dealing with this problem is dynamic priority assignment. Current investigations are underway to determine the effect of this and other such modifications on the performance improvement gained thru the use of the algorithm developed here.

[a]This research was supported in part by NSF Grant GN-655.

References

[1] Goutanis, R.J. and Viss, N.L. "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System", Proceedings of the IEEE, v54, #12, (1966), 1812-1819.

[2] Madnick, S.E. "Multi-Processor Software Lockout", Proc '68 ACM Nat. Conf., 19-24.

[3] Meridallio, R.A. and Holland, R.C. "Simulation Design of a Multiprocessing System", AFIPS FJCC, (1968), 1399-1410.

[4] Pass, E.M. An Adaptive Microscheduler for a Multiprogrammed Computer System, Ph.D. Dissertation, (1973), Georgia Tech, Atlanta, Georgia.

RADCAP: AN OPERATIONAL PARALLEL PROCESSING FACILITY

James D. Feldman
Goodyear Aerospace Corporation
Akron, Ohio 44315

Oskar A. Reimann
Rome Air Development Center
Rome, N.Y.

Summary: An overview is presented of RADCAP, the operational associative array processor (AP) facility installed at Rome Air Development Center (RADC). Basically, this facility consists of a Goodyear Aerospace STARAN[a] associative array (parallel) processor and various peripheral devices, all interfaced with a Honeywell Information Systems (HIS) 645 sequential computer, which runs under the Multics time-shared operating system. The RADCAP hardware and software are described briefly here because they are detailed in companion papers presented at this conference (1) (2). The latter part of this paper dwells on the objectives of the RADCAP facility and plans for its use.

[a]TM, Goodyear Aerospace Corporation, Akron, Ohio.

RADCAP Facility

Figure 1 shows a block diagram of the hardware within the RADCAP facility. The 645, which has been in existence at RADC for several years, is a very large computer system with a multitude of peripherals typical of large time-shared systems. In March 1973, hardware was delivered to RADC in the form of a STARAN parallel processor with four arrays, a custom input/output unit (CIOU), a hardware performance monitor, and a variety of peripherals. Subsequently, the CIOU was used to interface STARAN with a 645 I/O channel. At the same time, STARAN software was interfaced with the 645 Multics time-shared operating system.

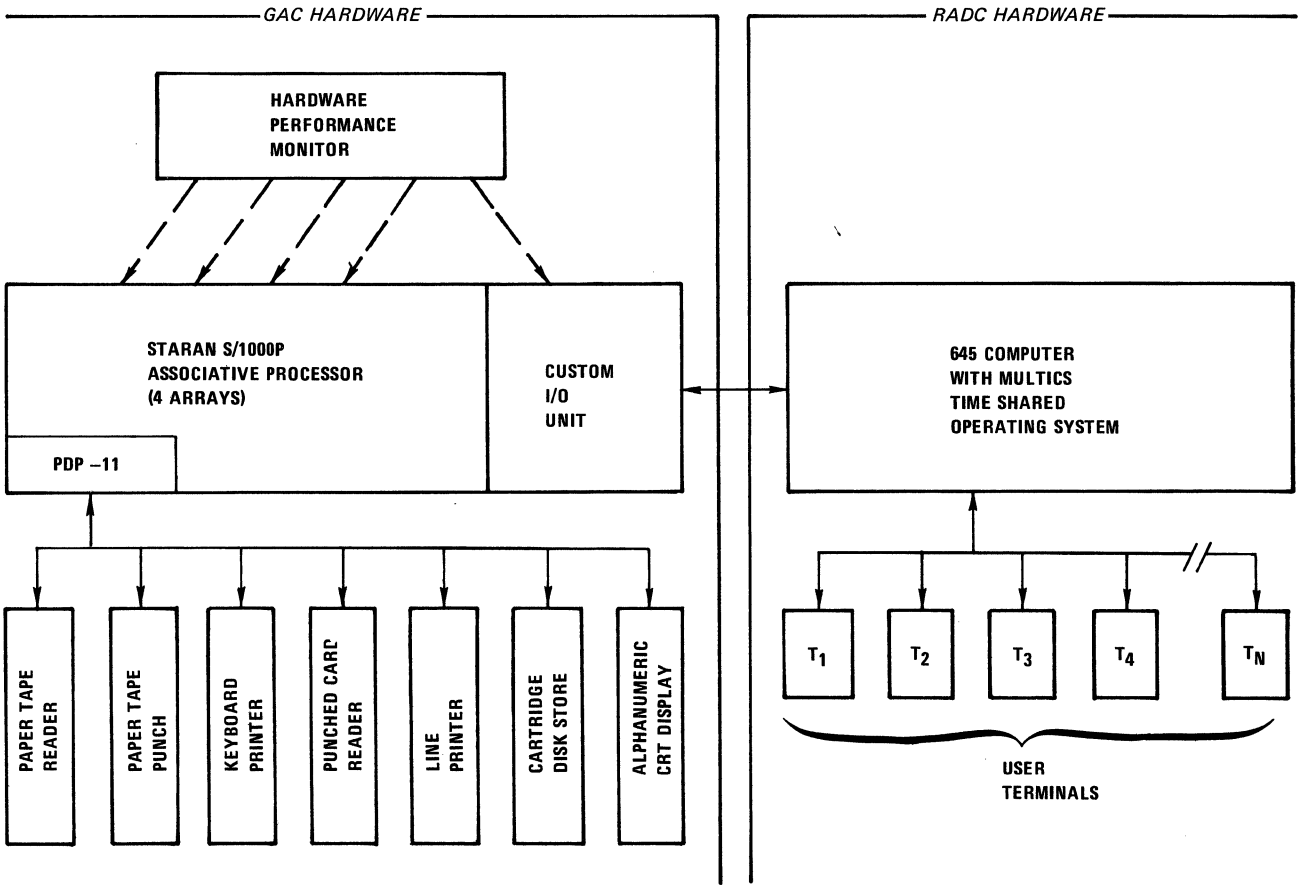


Figure 1. RADCAP Facility

At present, the RADCAP facility is totally operational and includes system software to allow for operation in both a STARAN stand-alone mode and an integrated STARAN/Multics mode.

STARAN Parallel Processor

STARAN can perform search, arithmetic, and logical operations simultaneously on either all or selected words of its memory. Figure 2 shows the basic STARAN elements. The most important is the associative array and its unique multi-dimensional access capability which, along with the other elements, are described in more detail in referenced publications (1) (3) (4). Listed below are brief descriptions of the STARAN elements:

1. Associative array: provides multi-dimensional access, content-addressable memory with 65,536 (2^{16}) bits of storage and 256 processing elements; permits parallel arithmetic, search, and logical operations.

2. AP control: performs data manipulation within associative arrays as directed by program stored in AP control memory.

3. AP control memory: stores AP control instructions. Can also store data and act as buffer between AP control and other system elements.

4. Sequential controller and memory: performs maintenance and test functions, controls peripherals, maintains job control, provides means for operator communication between various STARAN elements and, assembles STARAN programs written in MAPPLE (Macro-Associative Processor Programming LanguagE).

5. External function: transfers control information among STARAN elements.

STARAN has been designed to provide a flexible I/O capability. The standard peripherals for STARAN are listed below, along with a typical list of optional peripherals:

1. Standard: cartridge disk drive and control, paper tape reader, paper tape punch, and keyboard printer.

2. Optional: line printer, card reader, magnetic tape, keyboard crt, and other peripherals, as desired, that are compatible with the Digital Equipment Corporation (DEC) PDP-11.

All these peripherals interface with the STARAN system's sequential controller, a PDP-11 mini-computer. STARAN also provides facilities for interfacing with other processors. The four buses provided, (see STARAN block diagram, Figure 2) are the direct memory access, the buffered I/O, external function, and parallel I/O.

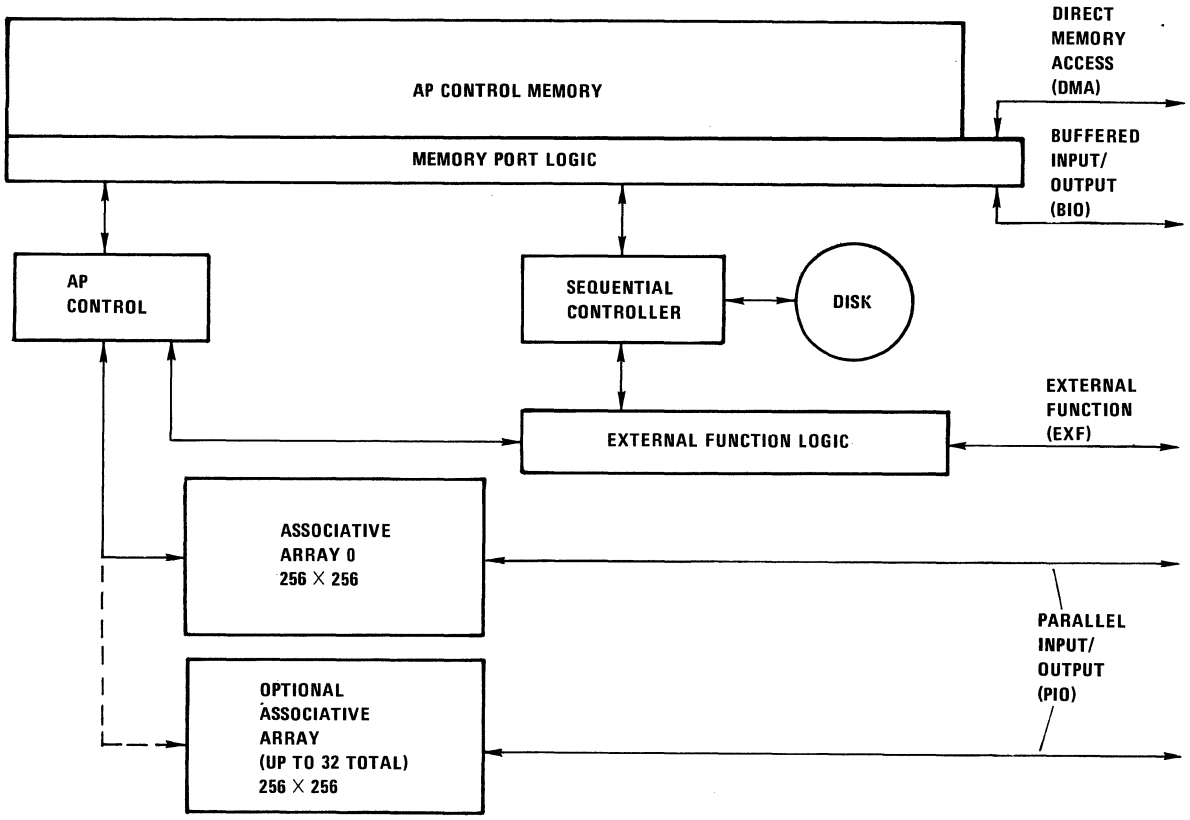


Figure 2. STARAN Block Diagram

The direct memory access is a 32-bit bus for STARAN to address external memory. The AP control or the sequential controller can access external memory at a rate dependent upon this memory's cycle time.

The buffered I/O is a 32-bit bus for processors to address STARAN. Depending upon which portion of control memory is accessed, the access rate is 0.4 to 1.0 microsec per 32-bit word.

The external function is a bus for exchange of control signals. Discrete signals and interrupts can be both generated and accepted across this bus.

The parallel I/O is a bus for STARAN array I/O. Up to 256 bits per array (e.g., one bit per array word) can be provided. If all 32 arrays are implemented, up to 8192 bits can be utilized in parallel at a transfer rate less than one microsecond, dependent upon the desired application.

STARAN Performance Summary

In a high-speed, asynchronous, pipe-line type processor such as STARAN, it is difficult to summarize performance since speeds vary with instruction types, types of loops, etc. Also, the overall effective speed depends upon the number of words in the arrays over which the simultaneous operations are occurring. However, an effort is made below to list the performance and features of a 256 x 256 associative array, the control unit, and the interface portion of STARAN:

Associative Array Features

Up to 32 Arrays per system

Multi-dimensional access (bit slice or word slice)

Array module speed:

- Typical search: 150 nsec/bit
- Typical add or subtract: 800 nsec/bit
- Read bit or word slice (256 bits): 150 nsec
- Write bit or word slice (256 bits): 300 nsec

Control Unit Features

Two separate processors: AP control, sequential controller

Solid-state control memory capacity: 2K x 32 standard, 4K x 32 maximum

Solid-state control memory speed: 150 nsec/instruction (typical)

Bulk core capability: 16K x 32 standard, 32K x 32 maximum

Bulk core speed: 1 microsec (read or write)

Interface Capabilities

STARAN to address external memory: rate-memory dependent

External processor to address STARAN: 0.4 to 1.0 microsec/32-bit word

Parallel I/O to/from associative arrays: less than 1.0 microsec/8192 bits (maximum)

Control signals and interrupts

Custom Input/Output Unit (CIOU)

Figure 3 shows a simplified block diagram of the STARAN/RADCAP custom input/output unit (CIOU). As indicated, the CIOU contains a parallel input/output (PIO) module, a 645 computer interface, and an internal performance monitor. The CIOU functions as a mini-processor much the same as the control unit portion of STARAN. Processing within one array module (e.g., under STARAN control) may be concurrent with I/O in another array module (e.g., under PIO control).

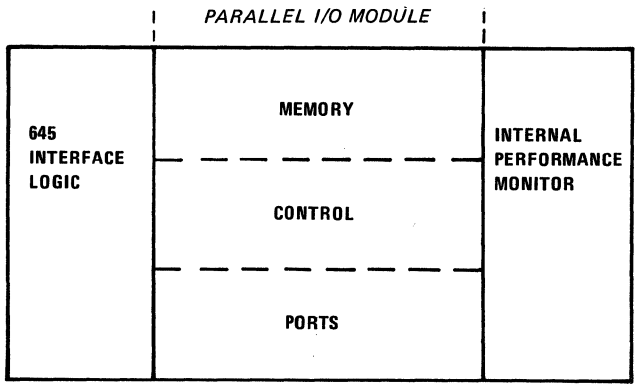


Figure 3. Simplified Block Diagram of Custom I/O Unit

As directed by instructions stored in PIO control memory, the optional PIO module manipulates data among and within the associative arrays concurrent with operations as directed by AP control. The PIO module contains eight ports, with 256 bits per port to accommodate associative array I/O and to permute data.

The 645 interface logic provides a communication path between the 645 computer and the STARAN system. This interface logic contains a 30-character queue and a 32-bit status register which are tied to a 645 I/O channel. The status register contains interface control signals, and the queue buffers data being transferred to or from the 645.

The internal performance monitor, although contained in the CIOU, is best discussed in the following description of the hardware performance monitor.

Hardware Performance Monitor

To help meet a RADCAP facility objective of measuring system performance, a hardware performance monitoring capability has been provided by an internal performance monitor in the CIOU cabinet and an external performance monitor system. Measurements can be made to determine instruction execution timing, control memory and bus utilization, array utilization, and activity in the pager, the PIO module, and the 645 interface.

The internal performance monitor is used exclusively for STARAN instruction execution times and instruction event times. The events counted and timed are the execution of flagged instructions in AP control. Between a start flag and an end flag, a timer increments at a 100-nsec rate. Overflows from this counter interrupt the sequential controller. In addition, the sequential controller can interrogate the event counter and timer.

The external performance monitor is a self-contained system that can monitor any point of STARAN or the custom I/O. Data are acquired via probes that detect logical signal changes in either an event count or elapsed time mode. Several probes can be logically connected via a patchboard to trigger a counter. At regular intervals, the contents of the counters are written as a record on a magnetic tape unit. The performance monitor software then evaluates the collected data and produces the results in the form of reports and graphs. The software for the performance monitor runs on the 645.

Physical Description of Hardware

All the elements shown in the STARAN block diagram (Figure 2), including the associative arrays, are built using dual-in-line IC's (integrated circuits) mounted on multi-layer printed circuit boards. Thus, the physical construction of STARAN and the CIOU is similar to that of typical high-speed sequential processors.

Figure 4 shows Goodyear Aerospace's STARAN demonstration and evaluation facility. Table 1 gives the approximate numbers of cabinets, boards, and IC's for the various STARAN models. These figures do not account for I/O logic, since this varies from one installation to another. The STARAN/RADCAP CIOU, which includes the parallel I/O option for all four arrays, contains approximately 200 boards and 8,000 IC's.

Figure 4. STARAN Demonstration and Evaluation Facility

Table 1. Approximate STARAN Component Count*

STARAN* Model	No. of Arrays	No. of Cabinets	No. of Printed Circuit Boards	No. of Integrated Circuits
S-250	1	3	220	9,000
S-500	2	3	276	11,500
S-750	3	3	332	14,100
S-1000	4	4	412	16,700
S-1250	5	4	468	19,300
S-1500	6	4	524	21,900
S-1750	7	5	604	24,900
S-2000	8	5	660	27,500
S-4000	16	8	1156	48,700

*Without input/output

Although up to three arrays can be packaged in one cabinet, the RADCAP configuration has two arrays per cabinet for symmetry. Figure 5 shows the equipment that was delivered to RADC. This includes a sequential control cabinet, an AP control cabinet, two AP memory cabinets for the four associative arrays, and a CIOU cabinet. The disk drive and line printer are mounted in separate cabinets. The keyboard/printer, the card reader, and the graphics display console can be mounted on table tops or pedestals. As mentioned earlier, the internal performance monitor is packaged within the CIOU cabinet. The external performance monitor, not shown in Figure 5, mounts on a table top.

Summary of System Software

The system software available for STARAN/RADCAP is capable of operating STARAN in a stand-alone mode or when integrated with the 645, in a STARAN/Multics configuration. The system software is based upon a disk operating system, which provides ready access to system programs, device independent I/O, and a file system. Operation of STARAN can be under direct control of the user at the control console or run in a batch mode with a control stream from an input device like the card reader.

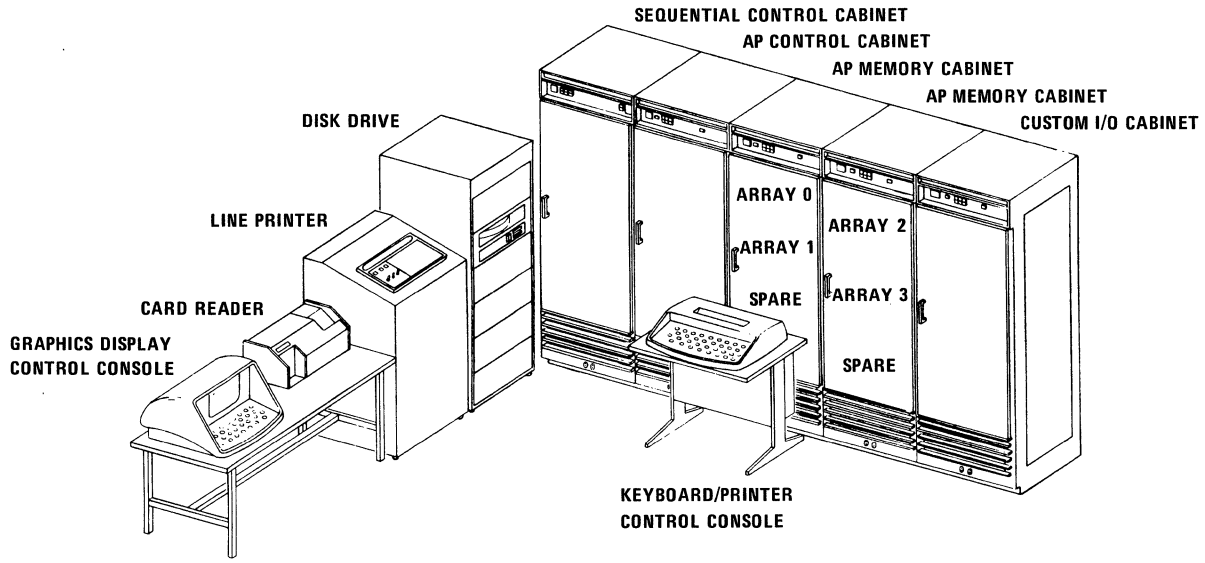


Figure 5. STARAN Complex at RADC

The total assembly package for STARAN has a macro language processor, an APPLE assembler, and a relocating linker. Programs are written in the APPLE and MAPPLE languages. Extensive string handling and substitution are implemented in the macro-preprocessor. APPLE is a symbolic language that includes mnemonics for parallel and associative operations. The linker combines separately assembled object modules by relocating code as necessary and resolving globally defined symbols.

Control of processing in STARAN is through interactive system routines. These routines are the interface between application program execution and the user. They allow the user to start and halt STARAN, to load programs and overlays, and to debug programs with trace, memory modification, and dump commands.

Diagnostic programs for STARAN hardware are disk resident. The programs can be called individually, in groups related to specific parts of the hardware, or as a total set for complete system testing. Fault detection and location are provided.

Additional software for the integrated STARAN/Multics operation is designed to handle the interface between the computers and the use of STARAN from Multics. For the interface, a special device driver module has been added to the STARAN disk operating system. This driver is similar to drivers used for peripherals. It has been specialized for Multics and can accommodate 16 open files simultaneously. A device interface module (DIM) has been added to Multics as the counterpart to the device driver. These two modules are basic parts of each machine's operating system and are transparent to the programmer.

STARAN can be operated from Multics by commands a user inputs at a terminal or from a file. File control procedures handle STARAN related keyboard inputs, and provide the interface between the DIM and the MULTICS storage system. With these procedures, a user process executing in the 645 can call for execution of a STARAN program.

To facilitate the assembly of STARAN programs, a cross assembler is provided for time-shared use in Multics. This assembler accepts MAPPLE and APPLE as inputs.

Objectives and Uses

The basic objective of the RADCAP facility is to explore the performance of a hybrid computer configuration (STARAN associative processor interfaced with a 645 sequential processor) on real-world, real-time problems. A specific goal is to determine the cost-effectiveness of associative/parallel processing in such an environment. Associative processing has been studied extensively in both theoretical and simulation studies, but no significant practical operating experience with them exists. Experimentation is necessary to provide "hard" data and fill in the presently existing void. Practical operating experience also is required so that a general-purpose associative processor configuration could be developed if results warrant it.

The RADCAP facility will be used in an experimental program to evaluate the internal performance of this hybrid computer configuration by means of hardware and/or software performance monitors to determine internal component utilization and system bottlenecks. Programming aspects of associative processing also will be investigated. Associative-processing programming is not well understood and represents radical departures from the traditional programming approach. The program loop is being replaced by hardware processing elements. This requires a whole new programming attitude. Programming languages suitable for associative processors probably will be quite different from present ones. This basic uncertainty must be explored and some practical operating experience gained. As a test problem, indicative of high data rate and real-time processing requirements, the data processing functions of an air surveillance system (AWACS) have been chosen. The primary functions to be investigated are tracking (both passive and active), display processing, and weapons control.

The scope of the research program can be described with the aid of Figure 6. The flow will begin with the development of associative-sequential algorithms for each of the AWACS data processing functions. As these algorithms are being developed, the application engineers will make known to a language and system software group those instruction level and system routine functions required to support the AWACS processing functions.

Based on this input, the language group will develop a language and implement this language on the RADCAP testbed. The system software activity will implement routines to support the command language. The applications program will then be run on the testbed using, where possible, nonsynthesized data as input. The machine activity will be monitored to gather statistics on utilization, identify system bottlenecks, and determine the efficiency with which the algorithms provide solution.

The data collected will then be analyzed to determine where cost effective improvements can be made to software and/or hardware in order to improve the cost-effective performance of the system. These changes will be incorporated into the system via micro program or software routines. If the change is to be a hardware design, that design will be made to the gate level so that performance and cost effectiveness determination can be made.

When the solution to the problem is finally refined, it will be contrasted with known sequential solutions.

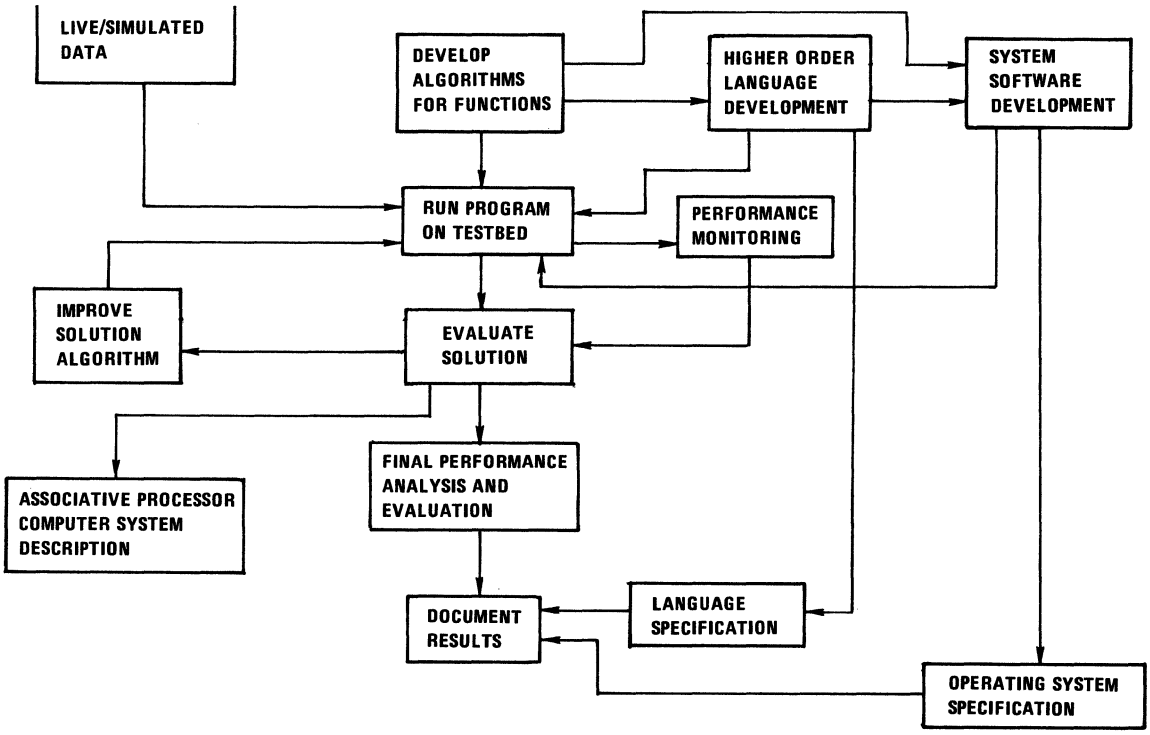


Figure 6. Flow of RADCAP Research Project

Initially each of the AWACS data processing functions will be treated separately. The final task will then be to develop a system executive and integrate all the functions to reflect the real world.

References

(1) K.E. Batcher, STARAN/RADCAP Hardware Architecture, GER-15947, Goodyear Aerospace Corporation (22 August 1973).

(2) E.W. Davis, Jr, STARAN/RADCAP System Software, GER-15948, Goodyear Aerospace Corporation (22 August 1973).

(3) K.E. Batcher, "Flexible Parallel Processing and STARAN," 1972 WESCON Technical Papers, Session 1.

(4) J.A. Rudolph, "A Production Implementation of an Associative Array Processor - STARAN," 1972 Fall Joint Computer Conference Proceedings, (December 1972), pp. 229 - 241.

STARAN/RADCAP HARDWARE ARCHITECTURE

Kenneth E. Batcher
Goodyear Aerospace Corporation
Akron, Ohio 44315

Summary: Hardware architecture is described for RADCAP, the operational associative array processor (AP) facility installed at Rome Air Development Center (RADC), N.Y. Basically, this facility consists of Goodyear Aerospace STARAN[a] parallel processor and various peripheral devices interfaced with a Honeywell Information Systems (HIS) 645 sequential computer, which runs under the Multics time-shared operating system. The hardware of STARAN/RADCAP is described with particular emphasis on the parallel processing elements.

Introduction

Companion papers presented at this conference describe the potential use of the RADCAP facility and its software (1) (2). The STARAN associative array (parallel) processor (3) employed in RADCAP has been modified to include a custom parallel input/output (PIO) unit and an interface to the 645 computer.

The parallel processing capability of STARAN resides in four array modules. Each array module contains 256 small processing elements (PE's). They communicate with a multi-dimensional access (MDA) memory through a "flip" network, which can permute a set of operands to allow inter-PE communication. This gives the programmer a great deal of freedom in using the processing capability of the PE's. At one stage of a program, he may apply this capability to many bits of one or a few items of data; at another stage, he may apply it to one or a few bits of many items of data.

The remainder of this paper deals with the MDA memories, the STARAN array modules, and the STARAN/RADCAP elements.

Multi-Dimensional Access (MDA) Memories

A common implementation of associative processing is to treat data in a bit-sequential manner. A small one-bit PE (processing element) is associated with each item or word of data in the store, and the set of PE's accesses the data store in bit-slices; a typical operation is to read Bit i of each data word into its associated PE or to write Bit i from its associated PE.

The memory for such an associative processor could be a simple random-access memory with the data rotated 90 deg, so that it is accessed by bit-slices instead of by words. Unfortunately, in most applications, data come in and leave the processor as items or words instead of as bit-slices. Hence, rotating the data in a random-access memory complicates data input and output.

To accommodate both bit-slice accesses for associative processing and word-slice accesses for STARAN input/output (I/O), the data are stored in a multi-dimensional access (MDA) memory (Figure 1). It has wide read and write busses for parallel access to a large number (256) of memory bits. The write-mask bus allows selective writing of memory bits. Memory accesses (both read and write accesses) are controlled by the address and access mode control inputs; the access mode selects a stencil pattern of 256 bits, while the address positions the stencil in memory.

For many applications, the MDA memory is treated as a square array of bits, 256 words with 256 bits in each word. The bit-slice access mode (Figure 2A) is used in the associative operations

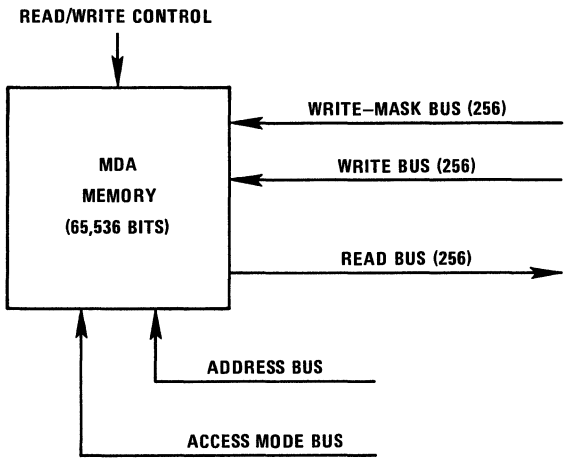


Figure 1. Multi-Dimensional Access Memory

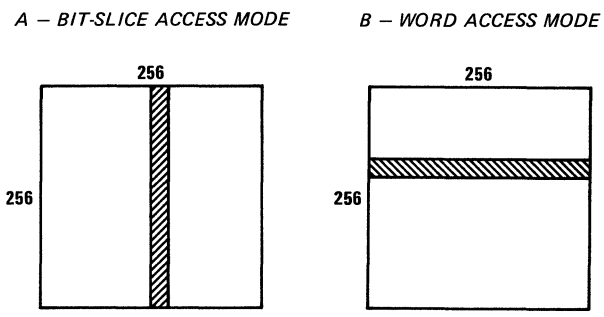


Figure 2. Bit-Slice and Word Access Modes

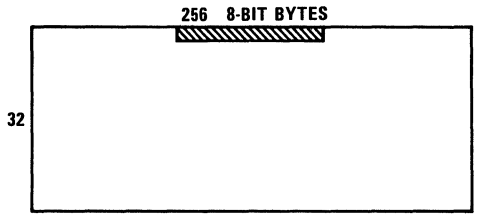
[a] TM, Goodyear Aerospace Corporation, Akron, Ohio.

to access one bit of all words in parallel, while the word access mode (Figure 2B) is used in the I/O operations to access several or all bits of one word in parallel.

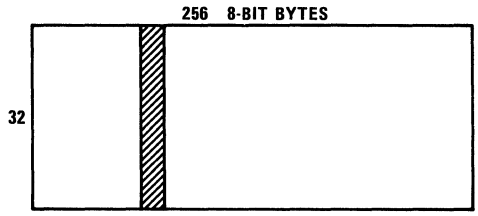
The MDA memory structure is not limited to a square array of 256 by 256. For example, the data may be formatted as records with 256 8-bit bytes in each record. Thirty-two such records can be stored in an MDA memory and accessed several ways. To input and output records, one can access 32 consecutive bytes of a record in parallel (Figure 3A). To search key fields of the data, one can access the corresponding bytes of all records in parallel (Figure 3B). To search a whole record for the presence of a particular byte, one can access a bit from each byte in parallel (Figure 3C).

The MDA memories in the RADCAP array modules are bipolar. They exhibit read cycle times of less than 150 nsec and write cycle times of less than 250 nsec.

A – ACCESS TO 32 CONSECUTIVE BYTES OF A RECORD



B – ACCESS TO CORRESPONDING BYTES OF ALL RECORDS



C – ACCESS TO ONE BIT OF EVERY BYTE IN A RECORD

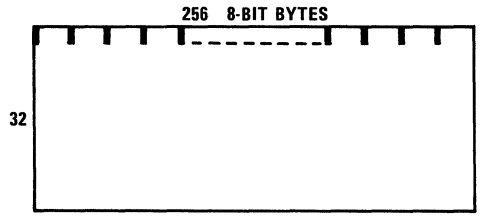


Figure 3. Accessing 256-Byte Records

STARAN Array Modules

A STARAN array module (Figure 4) contains a MDA memory communicating with three 256-bit registers (M, X, and Y) through a flip (permutation) network. One may think of an array module as having 256 small processing elements (PE's), where a PE contains one bit of the M register, one bit of the X register, and one bit of the Y register.

The M register drives the write mask bus of the MDA memory to select which of the MDA memory bits are modfied in a masked-write operation. The MDA memory also has an unmasked-write operation that ignores M and modifies all 256 accessed bits. The M register can be loaded from the other components of the array module.

In general, the logic associated with the X register can perform any of the 16 Boolean functions of two variables; that is, if x_i is the state of the i^{th} X-register bit, and f_i is the state of the i^{th} flip network output, then:

$$x_i \leftarrow \phi\,(x_i, f_i) \quad (i = 0, 1, \ldots, 255)$$

where ϕ is any Boolean function of two variables. Similarly, the logic associated with the Y-register can perform any Boolean function:

$$y_i \leftarrow \phi\,(y_i, f_i) \quad (i = 0, 1, \ldots, 255)$$

where y_i is the state of the i^{th} Y-register bit. The programmer is given the choice of operating X alone, Y alone, or X and Y together.

If X and Y are operated together, the same Boolean function, ϕ, is applied to both registers:

$$x_i \leftarrow \phi\,(x_i, f_i)$$

$$y_i \leftarrow \phi\,(y_i, f_i)$$

The programmer also can choose to operate on X selectively using Y as a mask:

$$x_i \leftarrow \phi\,(x_i, f_i) \qquad \text{(where } y_i = 1\text{)}$$

$$x_i \leftarrow x_i \qquad \text{(where } y_i = 0\text{)}$$

Another choice is to operate on X selectively while operating on Y:

$$x_i \leftarrow \phi\,(x_i, f_i) \qquad \text{(where } y_i = 1\text{)}$$

$$x_i \leftarrow x_i \qquad \text{(where } y_i = 0\text{)}$$

$$y_i \leftarrow \phi\,(y_i, f_i)$$

In this case, the old state of Y (before modification by ϕ) is used as the mask for the X operation.

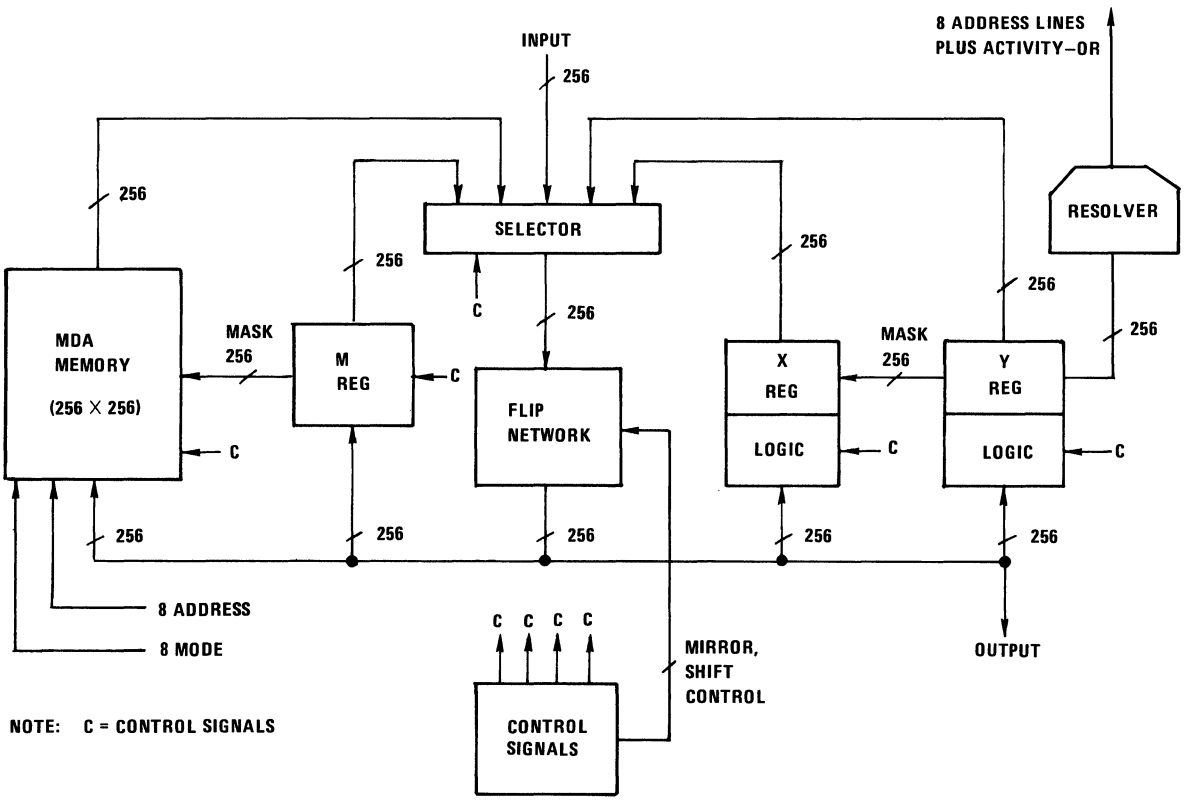


Figure 4. STARAN Array Module

For a programming example, the basic loop of an unmasked add fields operation is selected. This operation adds the contents of a Field A of all memory words to the contents of a Field B of the words and stores the sum in a Field S of the words. For n-bit fields, the operation executes the basic loop n times. During each execution of the loop, a bit-slice (a) of Field A is read from memory, a bit-slice (b) of Field B is read, and a bit-slice (s) of Field S is written into memory. The operation starts at the least significant bits of the fields and steps through the fields to the most significant bits. At the beginning of each loop execution, the carry (c) from the previous bits is stored in Y and X contains zeroes:

$$x_i = 0$$

$$y_i = c_i$$

The loop has four steps:

Step 1: Read Bit-slice a and exclusive-or ($\oplus$) it to X selectively and also to Y:

$$x_i \leftarrow x_i \oplus y_i a_i$$

$$y_i \leftarrow y_i \oplus a_i$$

The states of X and Y are now:

$$x_i = a_i c_i$$

$$y_i = a_i \oplus c_i$$

Step 2: Read Bit-slice b and exclusive-or it to X selectively and also to Y:

$$x_i \leftarrow x_i \oplus y_i b_i$$

$$y_i \leftarrow y_i \oplus b_i$$

Registers X and Y now contain the carry and sum bits:

$$x_i = a_i c_i \oplus a_i b_i \oplus b_i c_i = c'_i$$

$$y_i = a_i \oplus b_i \oplus c_i = s_i$$

Step 3: Write the sum bit from Y into Bit-slice s and also complement X selectively:

$$s_i \leftarrow y_i$$

$$x_i \leftarrow x_i \oplus y_i$$

The states of X and Y are now:

$$x_i = c'_i \oplus s_i$$

$$y_i = s_i$$

Step 4: Read the X-register and exclusive-or it into both X and Y:

$$x_i \leftarrow x_i \oplus x_i$$

$$y_i \leftarrow y_i \oplus x_i$$

This clears X and stores the carry bit into Y to prepare the registers for the next execution of the loop:

$$x_i = 0$$

$$y_i = c'_i.$$

Step 3 takes less than 250 nsec, while Steps 1, 2, and 4 each take less than 150 nsec. Hence, the time to execute the basic loop once is less than 700 nsec. If the field length is 32 bits, the add operation takes less than 22.4 microsec plus a small amount of setup time. The operation performs 256 additions in each array module. This amounts to 1024 additions, if all four array modules are enabled, to achieve a processing power of approximately 40 MIPS (million-instructions-per-second).

The array module components communicate through a network called the flip network. A selector chooses a 256-bit source item from the MDA memory read bus, the M register, the X register, the Y register, or an outside source. The bits of the source item travel through the flip network, which may shift and permute the bits in various ways. The permuted source item is presented to the MDA memory write bus, M register, X register, Y register, and an outside destination.

The permutations of the flip network allow inter-PE communication. A PE can read data from another PE either directly from its registers or indirectly from the MDA memory. One can permute the 256-bit data item as a whole or divide it into groups of 2, 4, 8, 16, 32, 64, or 128 bits and permute within groups.

The permutations allowed include shifts of 1, 2, 4, 8, 16, 32, 64, or 128 places. One also can mirror the bits of a group (invert the left-right order) while shifting it. A positive shift of mirrored data is equivalent to a negative shift of the unmirrored data. To shift data a number of places, multiple passes through the flip network may be required. Mirroring can be used to reduce the number of passes. For example, a shift of 31 places can be done in two passes: mirror and shift 1 place on the first pass, and then remirror and shift 32 places on the second pass.

The flip network permutations are particularly useful for fast-fourier transforms (FFT's). A 2^n point FFT requires n steps, where each step pairs the 2^n points in a certain way and operates on the two points of each pair arithmetically to form two new points. The flip network can be used to rearrange the pairings between steps. Bitonic sorting (4) and other algorithms (5) also find the permutations of the flip network useful.

Each array module contains a resolver reading the state of the Y register. One output of the resolver (activity-or) indicates if any Y bit is set. If some Y bits are set, the other output of the resolver indicates the index (address) of the first such bit. Since the result of an associative search is marked in the Y register, the resolver indicates which if any words respond to the search.

STARAN/RADCAP Elements

Each of the four array modules in STARAN/RADCAP (Figure 5) contains an assignment switch that connects its control inputs and data inputs and outputs to AP (associative processor) control or the PIO (parallel input/output) module.

The AP control unit contains the registers and logic necessary to exercise control over the array modules assigned to it. It receives instructions from the control memory and can transfer 32-bit data items to and from the control memory. Data busses communicate with the assigned array modules. The busses connect only to 32 bits of the 256-bit-wide input and output ports of the array modules (Figure 4), but the permutations of the array module flip networks allow communication with any part of the array. The AP control sends control signals and MDA memory addresses and access modes to the array modules and receives the resolver outputs from the array modules.

Registers in the AP control include:

1. An instruction register to hold the 32-bit instruction being executed.

2. A program status word to hold the control memory address of the next instruction to be executed and the program priority level.

3. A common register to hold a 32-bit search comparand, an operand to be broadcast to the array modules, or an operand output from an array module.

4. An array select register to select a subset of the assigned array modules to be operated on.

5. Four field pointers to hold MDA memory addresses and allow them to be incremented or decremented for stepping through the bit-slices of a field, the words of a group, etc.

6. Three counters to keep track of the number of executions of loops, etc.

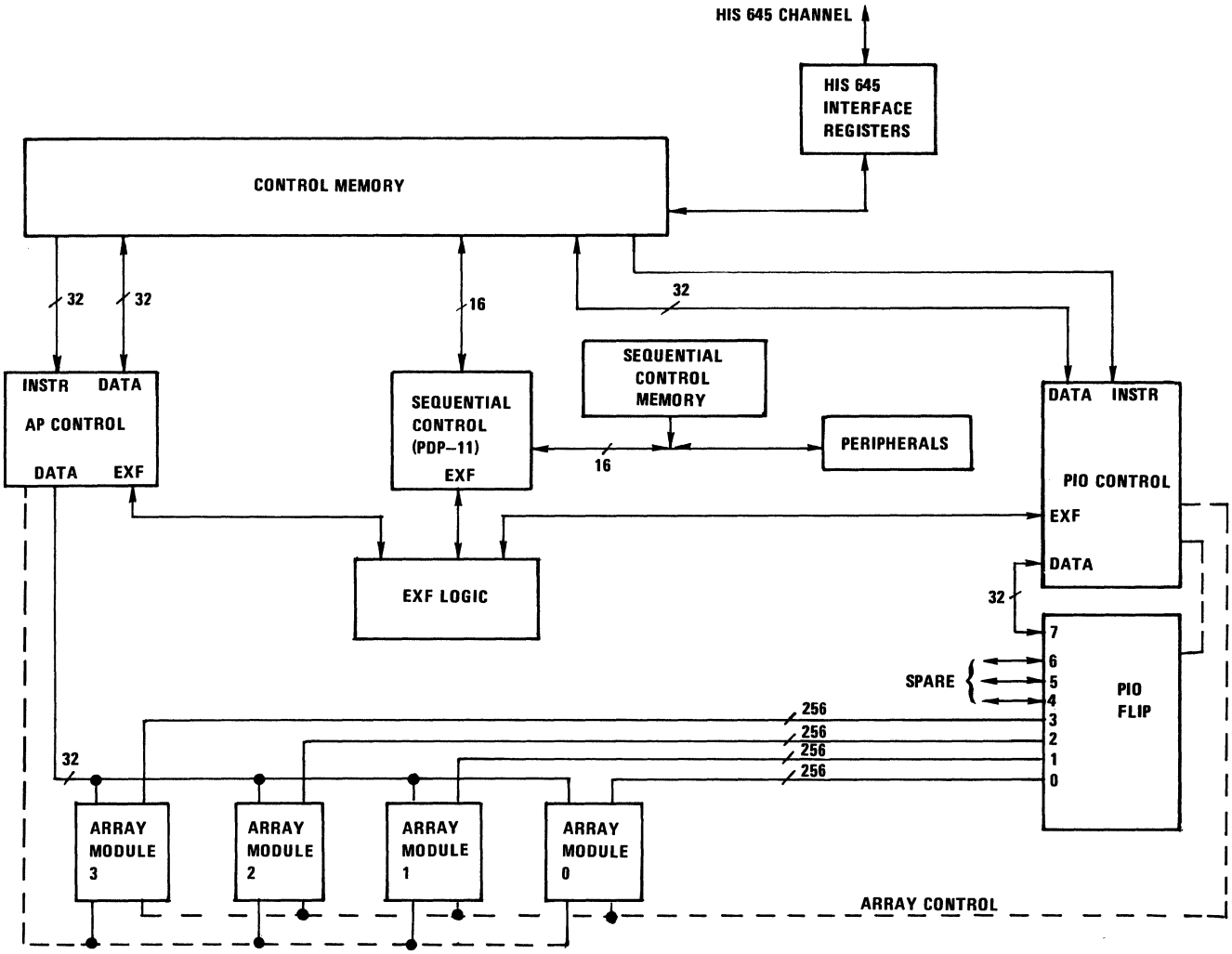


Figure 5. STARAN/RADCAP Block Diagram

7. A data pointer to allow stepping through a set of operands in control memory.

8. Two access mode registers to hold the MDA memory access modes.

The parallel input/output (PIO) module contains a PIO flip network and PIO control unit (Figure 5). It is used for high bandwidth I/O and inter-array transfers.

The PIO flip network permutes data between eight 256-bit ports. Ports 0 through 3 connect to the four array modules through buffer registers. Port 7 connects to a 32-bit data bus in the PIO control through a fan-in, fan-out switch. Ports 4, 5, and 6 are spare ports intended for future connections to high bandwidth peripherals, such as parallel-head disk stores, sophisticated displays, and radar video channels. The spare ports also could be used to handle additional array modules. High bandwidth inter-array data transfers up to 1024 bits in parallel are handled by permuting data between Ports 0, 1, 2 and 3. Array I/O is handled by permuting data between an array module port and an I/O port. The PIO flip network is controlled by the PIO control unit.

The PIO control unit controls the PIO flip network and the array modules assigned to it. While AP control is processing data in some array modules the PIO control can input and output data in the other array modules. Since most of the registers in the AP control are duplicated in PIO control, it can address the array modules associatively.

The control memory holds AP control programs, PIO control programs, and microprogram subroutines. To satisfy the high instruction fetch rate of the control units (up to 7.7 million instructions per second), the control memory has five banks of bipolar memory with 512 32-bit words in each bank. Each bank is expandable to 1024 words. To allow for storage of large programs, the control memory

also has a 16K-word core memory with a cycle time of 1 microsec. The core memory can be expanded to 32K words. Usually the main program resides in the core memory and the system microprogram subroutines reside in bipolar storage. For flexibility, users are given the option of changing the storage allocation and dynamically paging parts of the program into bipolar storage.

A Digital Equipment Corporation (DEC) PDP-11 minicomputer is included to handle the peripherals, control the system from console commands, and perform diagnostic functions. It is called sequential control to differentiate it from the STARAN parallel processing control units. The sequential control memory of 16K 16-bit words is augmented by a 8K × 16-bit "window" into the main control memory. By moving the window, sequential control can access any part of control memory. The window is moved by changing the contents of an addressable register.

The STARAN/RADCAP peripherals include a disk, card reader, line printer, paper-tape reader/punch, console typewriter, and a graphics console.

Synchronization of the three control units (AP control, sequential control, and PIO control) is maintained by the external function (EXF) logic. Control units issue commands to the EXF logic to cause system actions and read system states. Some of the system actions are: AP control start/stop/reset, PIO control start/stop/reset, AP control interrupts, sequential control interrupts, and array module assignment.

RADCAP connects to a common peripheral channel of a 645 computer. Channel characters are 6 bits wide. Instead of interfacing the channel to one of the three control units in RADCAP, the channel interface is assigned a set of control memory addresses so it can be addressed by any control unit. The interface has a 30-character first-in, first-out, (FIFO) queue to buffer the data transfer between the two machines. To reduce the number of queue accesses, the control units transfer queue data by character-pairs, 12 bits at a time.

References

(1) J. D. Feldman and O. A. Reimann, RADCAP: An Operational Parallel Processing Facility, GER-15946, Goodyear Aerospace Corporation and Rome Air Development Center (22 August 1973).

(2) E. W. Davis, Jr., STARAN/RADCAP System Software, GER-15948, Goodyear Aerospace Corporation (22 August 1973).

(3) K. E. Batcher, "Flexible Parallel Processing and STARAN," 1972 WESCON Technical Papers, Session 1.

(4) K. E. Batcher: "Sorting Networks and Their Applications," 1968 Spring Joint Computer Conference, AFIPS Proceedings, Vol 32, pp 307-314.

(5) H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Transactions on Computers, Vol C-20, No. 2, February 1971, pp 153-161.

STARAN/RADCAP SYSTEM SOFTWARE

Edward W. Davis
Goodyear Aerospace Corporation
Akron, Ohio 44315

Summary: System software is described for RADCAP, the operational associative array processor (AP) facility installed at Rome Air Development Center (RADC), N.Y. The description covers the software for the stand-alone operation of the Goodyear Aerospace STARAN[a] associative array (parallel) processor, which is supported by a disk operating system with a macro-assembler, a relocating linker and loader, an interactive debug package, and control procedures. Also described is the software for the STARAN processor when integrated with the Honeywell Information Systems (HIS) 645 sequential computer, which runs under the Multics time-shared operating system.

Introduction

The potential use of RADCAP and its hardware architecture are described in companion papers presented at this conference (1) (2). Basically, the RADCAP facility consists of an operational STARAN associative array (parallel) processor (2) (3) and various peripheral devices, all interfaced with a 645 computer.

There are two modes of RADCAP operation. First, STARAN can be operated as a stand-alone parallel processing system. Peripherals for this mode include a card reader, line printer, paper tape reader and punch, and cartridge type disk unit. Second, STARAN and the 645 can be operated in an integrated fashion. This means that (1) commands to the STARAN disk operating system can originate in Multics, (2) the Multics storage system is available to STARAN users for program or data storage, and (3) a single task can use both machines to satisfy its processing requirements. All peripherals belonging to a stand-alone STARAN and to the HIS 645 are available when the machines are integrated.

This paper describes the software for the STARAN stand-alone mode of operation, then covers the additional software used with the integrated mode.

Since the STARAN processor architecture is detailed in a companion paper (2), only a basic diagram is given in Figure 1. The multi-dimensional access associative arrays and their controls are the main architectural features. The sequential control, a Digital Equipment Corporation (DEC) PDP-11 minicomputer, has a minor role in the architecture, but is important for software considerations. Other architectural features are mentioned later in the paper.

[a]TM, Goodyear Aerospace Corporation, Akron, Ohio.

Software For STARAN Stand-Alone Mode

Software for the STARAN stand-alone mode of operation can be discussed from the standpoints of the operating system, language processing, and operational software.

Batch Disk Operating System

In this paper, an operating system means the collection of routines that give the user appropriate control of the computing system, inform him of system status, provide input/output (I/O) facilities, and provide access to system programs. STARAN features a disk operating system (DOS) and has a batch processing capability. The batch command stream can be assigned to any ASCII character input device, allowing control to originate at the control console or from a user's file on the batch device.

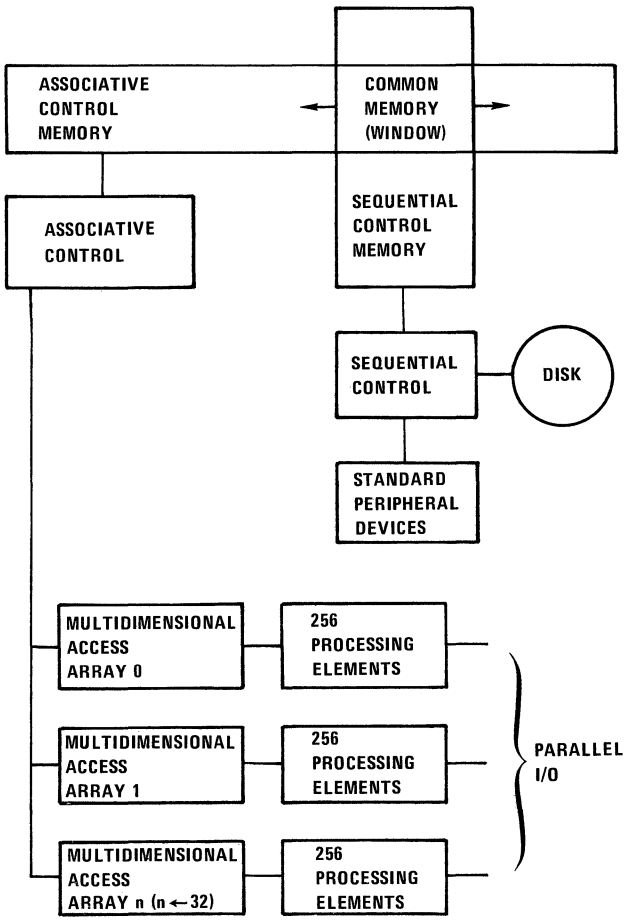


Figure 1. STARAN Block Diagram

The disk is a file structured bulk storage medium. All system software is resident on the device for easy, rapid access by the user.

Listed below are the standard programs supplied with the DEC PDP-11 batch system:

Program Name	Function
MACRO	Macro-assembler
LINK	Linker
LIBR	Librarian
PIP	File utility package
EDIT	Text editor
ODT	On-line debugging package
FORTRAN	Fortran compiler

These programs are not covered in detail since primary emphasis in this paper is on the STARAN-related software that has been added to the above list to build the STARAN disk operating system.

One general rule used in software development was to avoid changes to the basic DEC batch system. This rule was intended to simplify any future change to a new DEC release.

Language Processing

APPLE. Programs for STARAN are written in the APPLE[b] assembly language (Associative Processor Programming LanguagE). This language has some mnemonics that generate one machine language instruction and others that generate a sequence of machine instructions (5). The one-to-many mnemonics generally implement a parallel algorithm for arithmetic or search operations using the arrays. Thus, APPLE is at a higher level than sequential machine assembly languages.

APPLE produces relocatable or absolute program sections and has a conditional assembly capability. Groups of instructions in the language are listed below:

1. Assembler directives
2. Branch instructions
3. Register load and store
4. Associative instructions
   a. Loads
   b. Stores
   c. Parallel searches
   d. Parallel moves
   e. Parallel arithmetic operations
5. Control and test instructions
6. Input/output (I/O) instructions

Most of these groups of instructions resemble those of other typical assemblers. The unique group - associative instructions - deals with operations on the multi-dimensional access arrays and the registers in their processing elements (PE).

Some general comments apply to all the associative instructions listed above. Operations take place only on arrays enabled by the array select register (2). Fields are of variable length within each array word and are defined for various instructions by field pointers and length counters. The common register, a part of associative control, can contain an operand which is used in common by all selected array words.

More detail is presented below on the associative instructions; i.e., loads, stores, parallel searches, parallel moves, and parallel arithmetic operations.

The load associative instructions load the processing element (PE) registers or the common register with data from the arrays. Logical operations may be performed between the current PE register contents and the array data. The language has mnemonics for the common logical operations, while the machine supports all 16 functions of two logical variables. A given load instruction can increment, decrement, or leave as is an array field pointer. Thus, a single one of these instructions can load registers, perform logic, and change pointer values. Operations to set, clear, or rotate the PE register are included in this group.

The store associative instructions are used to move PE or common register data into the arrays. A mask feature is provided that allows writing only in mask enabled array words. As with the load instructions, logical operations may be performed between the current PE register contents and the array data. Also, the array field pointer can be incremented, decremented, or left unchanged.

The parallel search associative instructions allow the programmer to search for particular conditions in the arrays. Only those words enabled by the mask register take part in the searches. Searches can be performed that compare a value in the common register with a value in a field of all array words. Another variety of search compares one field of a word with a second field of the same word for all array words. Comparisons can be made for such conditions as equal, not equal, greater than, greater than or equal, etc. Maximum and minimum searches also can be performed. Combinations of searches yield such functions as between limits and next higher. Additional mnemonics in this group are provided to resolve multiple responders to the searches.

The parallel move instructions are provided to move an array memory field to another field within the same array word. As with searches, a word is active for this instruction only when enabled by the mask register. Types of moves are direct, complement the field, increment or decrement the field, and move the absolute value.

The parallel arithmetic operation associative instructions allow the programmer to perform such parallel operations in the arrays. These operations are subject to mask register word enabling.

[b] TM, Goodyear Aerospace Corporation, Akron, Ohio.

Arithmetic can use a value in the common register as one operand and a value in a field of all array words as the parallel operand. Alternatively, one field of a word can be arithmetically combined with a second field of the same word for all array words. Operations supplied by APPLE are add, subtract, multiply, divide, and square root.

Macro. A macro language is provided to increase the user's flexibility at assembly time (6). The macro language has a large set of arithmetic, logical, relational, and string manipulation operators. Adding macro variable symbol handling, conditional expansion capability, and ability to nest macro calls make it possible to write powerful macro instructions. A system macro library feature has been implemented.

Benefits to the user are the ability to define new mnemonics, redefine existing mnemonics, and conveniently generate standard instruction sequences.

Mnemonics have been added to the basic APPLE language for RADCAP by writing macros and putting them in the system library. Primarily, the added mnemonics are floating point instructions. They are fixed field length operations in both single and double precision.

Building Load Modules. Software used to convert source language programs into executable load modules includes an APPLE assembler, macro-preprocessor, and relocating linker. Figure 2 shows this software and the flow of programs or modules through it.

Building load modules begins with the original program written in APPLE. This source program may contain macro instructions. Translation of the source into a machine language object module is by MAPPLE, (APPLE assembler with Macro-preprocessor on the front end). If it is known that the source program does not contain macro instructions, it is possible to input the source directly to the APPLE assembler.

A relocatable object module is converted to an absolute load module by the STARAN linker. Multiple object modules may be input to the linker since it has the function of resolving symbols defined across object module boundaries (global symbols) as well as adjusting addresses for relocation.

Use of the language processing software is fully described in the STARAN user's guide (7).

Operational Software

Operational software is discussed below from the standpoints of loading, executing, and debugging programs on STARAN. Four modules are involved: loader plus STARAN program supervisor, debug module, and control module.

Loader. Output of the STARAN linker is shown in Figure 2 as an absolute load module. The loader has the straightforward task of moving a load module into STARAN control memory beginning at the address specified in a text block. Options on loading are to load and not execute or to load and begin execution either at an address given with the load module or at one given with the load command.

The load module can be linked with a user program to enable calling for a load from an executing program. This means that overlay modules can be brought in dynamically.

STARAN Program Supervisor (SPS). The SPS is the software interface between the associative and sequential portions of STARAN. This module has services for system users when programming in APPLE and when programming a PDP-11 routine to interact with an APPLE program.

For the APPLE program, SPS makes the I/O instructions of the disk operating system (DOS) available, provides a program overlay capability, and provides a programmable interrupt to a PDP-11 routine. The PDP-11 routine interacts through

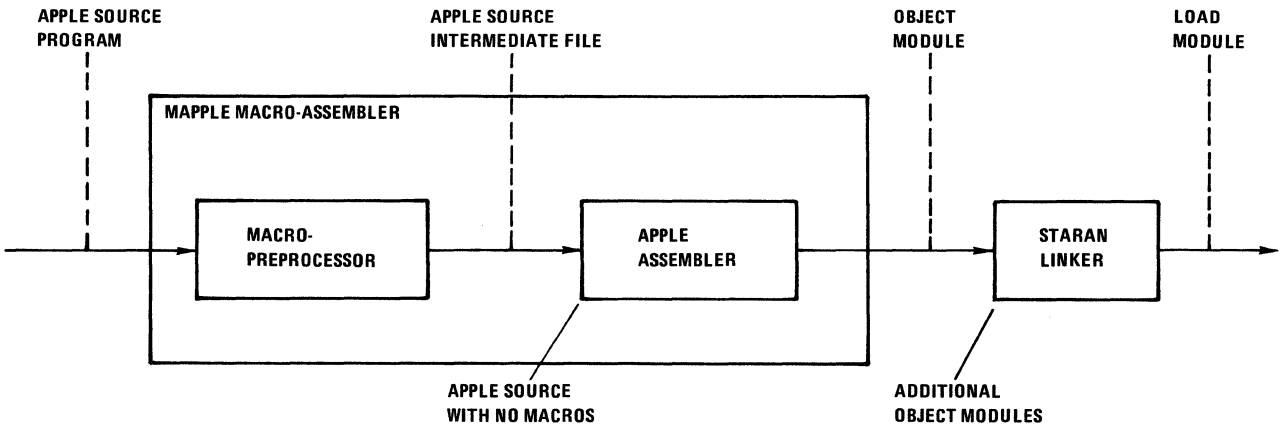


Figure 2. Language Processing Software

a software link, which receives the APPLE interrupts, and through the issuing of control information to the associative control logic.

In addition, the SPS supplies interface services. It transfers data between associative and sequential memory through the common memory window (Figure 1). The SPS also fields associative processor error interrupts.

Concurrent execution of associative and sequential routines, with interaction, is made possible by the SPS.

STARAN Debug Module (SDM). The SDM helps get rid of bugs in APPLE programs by giving the user control of the execution of the program being debugged, and access to memory and registers. Such features as single step, trace, and breakpoint provide good execution control. Dumps of all memory areas can be taken, with both word slice and bit slice available for the multi-dimensional access arrays. All memory locations also can be modified.

STARAN Control Module (SCM). This final operational module is the interface between the user and execution of a STARAN program. By running the SCM, the user enters a mode in which STARAN related commands are recognized. Such commands as start, halt, and continue execution are processed directly by the SCM. When the load command is used, the SCM passes control to the loader for that function. If debug aids are needed, a simple command adds all debug module features to the SCM.

All the operational software modules are described more fully in the STARAN user's guide (7).

Software for STARAN/645 Multics Mode

General

In the RADCAP facility, the integrated use of the STARAN parallel processor and the 645 sequential computer makes additional software necessary. One major concern is the interface between the computers; this requires a software module in both machines. A second concern involves reasonable ease of use for the integrated mode; four procedure packages that execute totally in the 645 were added to satisfy this concern.

Figure 3 shows the relationship between software modules in STARAN and the 645. As indicated, the Multics time-shared operating system of the 645 contains three categories of software: command level, user process, and system related. Command level software is brought into execution by user-supplied commands, as from a Multics terminal. User process software consists essentially of subroutines called from a user program. System-related software is the collection of routines that support use of the system, such as handling input and output, and are usually called indirectly by the user program.

Additional details on the design and use of software are described in the STARAN/645 user's guide (8).

Interface Modules

The two modules for the interface, shown in Figure 3, are the 645 device driver in the STARAN batch disk operating system (DOS) and the STARAN device interface module (DIM). These modules are discussed below.

645 Device Driver. This driver provides the interface between the DOS monitor and the 645 computer. It communicates with the monitor as do other device drivers for standard peripherals. If the device looks like an input for character information, then batch commands can come from it. The batch stream can be assigned to the device. This is the significance, for Multics, of the batch feature on the DOS.

In reality, the device treated by the 645 driver is used for much more than character input. The 645 appears as three logical devices with unit numbers 0, 1, and 2.

Unit 0 looks like the disk, logically. Before transferring data, it is necessary to "open" a file using a file name and extension in the DOS format. The driver supports both ASCII and binary transfer modes, both formatted and unformatted. A data-set remains open until a "close" call is issued. At any one time, up to 14 data-sets may be open on unit 0.

Unit 1 looks like a card reader, logically. It is a read-only device with an ASCII transfer mode. This unit serves as the batch command stream input so a Multics user can control the system.

Unit 2 looks like a paper tape punch, logically. It is a write-only device with ASCII and binary transfer modes. Job log output, in the integrated mode, is always assigned to this unit.

Because of the nature of the 645 device and its expected usage, the device driver has two custom functions built in. An "idle" function is used to tell Multics when the command stream file has been processed. A "detach" function, called when a Multics user detaches from STARAN, performs cleanup and makes STARAN ready for a new Multics user.

STARAN DIM. In Multics terminology, a device interface module (DIM) coordinates communications with a particular physical device. The four major functions are performed by the DIM are: (1) attachment, (2) read command from STARAN, (3) respond to STARAN command, and (4) detachment.

Attachment is the function through which a user process gains access to STARAN. The interface is initialized by a call to the attachment entry point in the DIM. STARAN is available as a Multics system resource to only one process at a time.

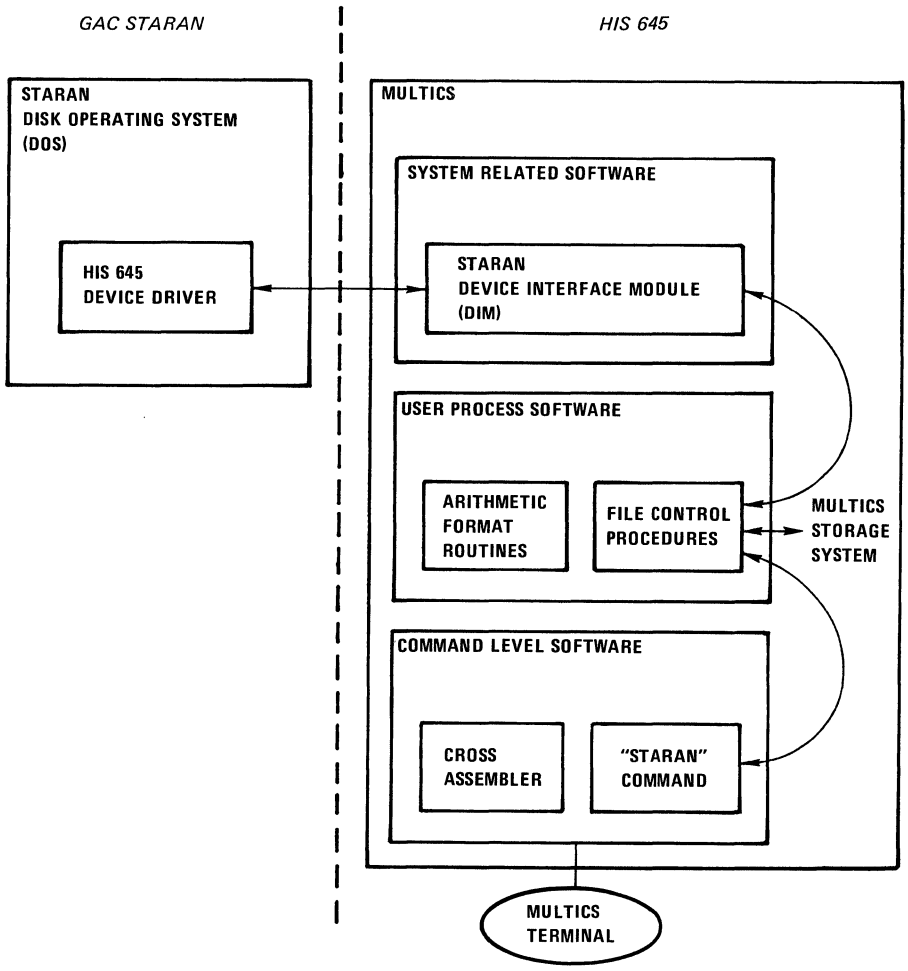


Figure 3. STARAN/645 System Software Relationship

Therefore, further calls to the DIM can be made only after a successful attachment.

The function of reading a command from STARAN occurs after attachment. The first command should be to read ASCII on the command stream input device. As noted above, Unit 1 of the 645 device driver is provided for this purpose.

Once the initial sequence is past, the DIM must respond to STARAN commands. The call made by the Multics process is determined by the previous STARAN command. For example, if STARAN issues a read call, Multics must write.

Finally, the detachment function severs the link between the user process and STARAN.

Data manipulation by the DIM assumes all Multics data is in character form. It converts characters into the form needed for output to STARAN and converts data received from STARAN into Multics character form. This means, for example, that Multics arithmetic data must be converted to a character form prior to output, and from characters following input. The conversion is done by a procedure superior to the DIM. The DIM also handles retransmission of bad data and reports a failure to its caller after a specific number of unsuccessful tries on the same data.

In the Multics software structure, the DIM is located in a position inferior to the file control procedures, shown in Figure 3 and described in the next part of this paper.

System Use Modules

File Control Procedure (FCP). The FCP greatly simplifies operation of STARAN from Multics. It enables a Multics user process (program) to interact with STARAN by initializing the interface, handling communication between the machines, and terminating the interface. The FCP also makes the necessary calls to the DIM to

initialize and terminate the interface. Communication is described in the following paragraphs.

Once the interface has been established, Multics appears to STARAN as a set of three logical devices, defined above as Units 0, 1, and 2.

Unit 0 is like a disk. All operations on this file-structured device are initiated within STARAN by I/O instructions and are performed within Multics. The FCP represents the interface between the DIM and Multics storage for all file operations. It handles the opening and closing of files, makes file names known to Multics, and issues appropriate calls to the DIM for read and write operations.

Unit 1 is like a card reader. It is the source of batch stream commands to the STARAN operating system. The FCP must recognize requests for these commands, read the commands from the source in Multics, and write them to STARAN. The source can be either a Multics terminal or named file. All calls to the DIM are made by the FCP.

Unit 2 is the destination of job log output. The FCP sorts this out and directs it to a Multics terminal or named file. Again, all calls to the DIM are handled by the FCP.

With FCP, a user process, executing in the 645, can call for STARAN, and it can pass commands, programs, and data to STARAN. The FCP raises the point at which the user becomes involved from sequences of calls to the DIM to a more symbolic call to FCP routines from the user process.

STARAN Command. User involvement in the interface to STARAN is raised still higher from the user process to the Multics command level by a "STARAN" module. Essentially, this module is a supplied user process that passes parameters used in the terminal command to the FCP. The parameters identify the STARAN batch command stream input and output devices. The module calls appropriate FCP routines to establish interaction with STARAN.

In typical operation of STARAN from a terminal, this Multics command is used with STARAN commands also coming from the terminal. Initializing and terminating the interface are not a concern of the user. The Multics terminal becomes very similar to the STARAN control console when this module is used.

Arithmetic Format Routines (AFR). STARAN and the 645 differ in the lengths of their data representations. STARAN has a 32-bit control memory, while the 645 has a 36-bit word length. Arithmetic format routines are provided to convert either integer or floating point data between the 645 format and the format used by the DIM for transmission to STARAN.

In the Multics to STARAN direction, integer data are converted by truncating the most significant four bits. A check is made to verify that the integer can be represented in 32 bits. Floating point data are converted by truncating the least significant bits of the mantissa.

From STARAN to Multics, integer conversion is done by extending the sign bit. Floating point conversion is done by filling the low order mantissa bits with zeros.

Cross Assembler. This is a functionally equivalent version of the MAPPLE assembler, written in PL/1, to be run in Multics. It is available to terminal users on a time-shared basis. It accepts APPLE and macro statements and produces STARAN object code in the Multics character format required by the DIM for transmission to STARAN.

Conclusion

A brief description has been given of the software that makes up the operating system for operational STARAN associative array processor installed in the RADCAP facility. Also described is the additional software that makes STARAN operational when integrated with 645 sequential computer. The goal of all the software is to provide tools to use STARAN in the stand alone and integrated modes. The tools are intended to increase convenience for the user and improve total system throughput.

Many modules have been discussed. Some of these are essentially transparent to the user, some may not be needed by certain users, and some may be required by all users. For stand-alone STARAN operation, the programmer must know APPLE and the use of the assembler and linker. He must be able to run the control module and load programs. He will probably be interested in the debug module. The STARAN program supervisor is transparent for most users. It is not necessary to know any of the sequential control program or languages.

To use STARAN from a Multics terminal, the only additional requirement is to know how to connect STARAN and Multics using the Multics "STARAN" command. If the user wishes to have a Multics user process (i.e., a program) interact with STARAN, then the calls to the file control procedures and use of the arithmetic format routines become important. The 645 device driver and the STARAN DIM are transparent to users. The cross assembler is a convenience for Multics users and may be used instead of the assembler in STARAN.

References

(1) J.D. Feldman and O.A. Reimann, RADCAP: An Operational Parallel Processing Facility, GER-15946, Goodyear Aerospace Corporation and Rome Air Development Center (22 August 1973)

(2) K.E. Batcher, STARAN/RADCAP Hardware Architecture, GER-15947, Goodyear Aerospace Corporation (22 August 1973).

(3) J.A. Rudolph, "A Production Implementation of an Associative Array Processor - STARAN," 1972 Fall Joint Computer Conference Proceedings, (December, 1972), pp.229-241.

(4) E.I. Organick, The Multics System, MIT Press, 1972.

(5) STARAN S APPLE Programming Manual, GER-15637A, Goodyear Aerospace Corporation (August, 1973).

(6) STARAN S MACRO Programming Manual, GER-15643, Goodyear Aerospace Corporation (August, 1973).

(7) STARAN Users Guide, GER-15644, Goodyear Aerospace Corporation (August, 1973).

(8) STARAN/HIS-645 Users Guide, GER-15641, Goodyear Aerospace Corporation (August, 1973).

APPLICATION OF STARAN TO SUPPORT REGION ANALYSIS FOR A MECHANICAL ROBOT

J. M. Plante and D. J. Gondek
Rome Air Development Center (IRDA)
Griffiss Air Force Base, New York 13441

Summary

For the past seven years the Advanced Research Projects Agency (ARPA) has sponsored research at Stanford Research Institute (SRI) in the area of artificial intelligence. The primary goal of this project has been to investigate techniques in artificial intelligence applied to the control of a mobile automaton (robot) in a real environment. The main emphasis has been on the design of a hierarchy of algorithms that will accept visual and other sensory information gathered by the automaton. Specifically algorithms are developed to support the analysis of the controlled environment in which the automaton resides (1). The potential application of STARAN to support a selected subset of these algorithms (i.e. Region Analysis) was investigated and is summarized in this paper.

The Region Analysis algorithm uses a decision tree. Nodes in the tree correspond to an operator to be applied, and branches emanating from a node correspond to the results of that operation. Any path through the decision tree eventually leads to a terminal node corresponding to a description of the location, and possibly the identification of an object in the scene. Repeated passes through the tree produce a list of such information describing the scene.

The Region Analysis algorithm is designed to:

(1) Assign region numbers and identify related neighbors within the overall environment.

(2) Assign scores for "Best Guess Region Type". This information is derived from the afore mentioned Scan and Merging Heuristic Algorithms.

(3) Object identification within regions, as related to the overall environment.

The data (identified regions) is then used as input for further Scene Analysis before being passed to the main body of robot programs (i.e. question-answering, navigation/route plotting, problem solving, etc.)

The Scene Analysis program as executed on a sequential computer, uses a number of special purpose subroutines to extract evidence from or to apply to a picture/scene. These low-level routines operate in a quasi-intelligent fashion, in that they perform some operation and return an answer based on previous results and the sensibleness of their answers.

The highly iterative Region Analysis algorithm (which are a subset of the Scene Analysis algorithms) have not been currently implemented on any conventional sequential machines due to the excessive computational time required to execute them. Since this particular subtask performs many repetitive sequential operations which collect very similar samples/packets of related data elements, parallel processing techniques for performing the Region Analysis functions were investigated. The conclusion of the study was that the application of the STARAN Associative Processor is a viable solution which readily lends itself to this programming task.

For information concerning the design and operation of the mechanical robot and supporting programming subtasks consult references (1) and (2).

For supporting technical data on the STARAN Associative Processor System, the reader is directed to reference (3).

References

(1) R. O. Duda, "Some Current Techniques for Scene Analysis", Stanford Research Institute, A.I. Group, Tech Note 46, Project 8259, (October, 1970), pp. 1-20.

(2) C.R. Brice, and C.L. Fennema, "Scene Analysis Using Regions", Stanford Research Institute, A.I. Group, Tech Note 17, Projects 7494 and 8259, (April, 1970), pp. 1-19.

(3) Goodyear Aerospace Corporation, STARAN Development Division, Akron, Ohio 44315.

A DATA MANAGEMENT SYSTEM UTILIZING THE STARAN ASSOCIATIVE PROCESSOR

Richard Moulder
Digital Systems, D/472
Goodyear Aerospace Corporation
Akron, Ohio 44315

SUMMARY

An on-line data base management system (DBMS) utilizing the STARAN Associative Processor has been designed and implemented at Goodyear Aerospace. The hardware configuration is composed of Goodyear's STARAN S-1000 with a parallel head-per-track disc (PHD) and a Xerox Data Systems Sigma 5 computer. Communications between the two computers is via Direct Memory Access (DMA). The PHD is for peripheral data storage and consists of a single disc with 64 tracks. Each track has a head and read/write electronics. This design allows data to be read into or out of the associative arrays over a communications channel which is 64 bits wide.

A four level hierarchical data base was selected and implemented in our DBMS. The technique used for actually storing the data on the PHD was the Associative Normal Form (ANF) suggested by DeFiore and others [1]. Employing ANF we developed a data base having no external indices and no organization by record type. This allowed a significant saving in peripheral storage with little or no degradation in query or update response times. This was made possible because of the parallel input/output and parallel content searching capabilities of STARAN. The benefits of a fully inverted data base were achieved without the attendant increase in peripheral storage.

The software system was composed of four basic modules. These modules can be found in most DBMS and are the Define, Create, Interrogate, and Update Modules. The Define module describes the logical data structure to the computer system. In our implementation, the Define module was similar to IBM's GIS/2 [2]. The Create module populates the data base by mapping the logical data structure to the ANF and writing the data to the PHD. The most used modules are the Interrogate and Update modules. These modules are used via a graphic display console to query and change the data base. A non-procedural language tailored after SDC's SACCS Data Management System [3] was employed. Any data item or attribute of a record can participate in the search criteria with multiple criteria being permitted. Besides the standard query and update functions that were provided, an additional function called "Move" was introduced. This command allowed the restructuring of the hierarchical data base without going through a "Delete" and an "Add."

A user's request is typed on the graphic display terminal and then transmitted to the Sigma 5. The request is passed through an input validation software module. Following validation, the request is processed by a translation module. This translation includes the restructuring of the selection criteria according to the rules for Reverse Polish Notation. A task list of I/O functions involving the search criteria is constructed and transmitted via DMA to the STARAN. The task list is executed and records that satisfy the search criteria are transmitted back to the Sigma 5. Information is extracted from the records, formatted, and displayed.

Our results to date show that associative processors working in concert with sequential processors performing in a DBMS environment are an excellent marriage of two computer concepts. With multiprocessing capabilities, greater throughput can be achieved. Timing results show that for the implemented data base, query and update times are nearly equal. Our results also show that a DBMS employing Associative Processors will require less software. This is due to the simplicity of the data storage techniques. For a more detailed description of the Data Management System implemented on STARAN, the reader is directed to references [4] and [5].

REFERENCES

[1] C.R. DeFiore, N.J. Stillman, P.B. Berra, "Associative Techniques in the Solution of Data Management Problems," Proceedings of ACM (1971), pp. 28-36.

[2] Generalized Information System Version 2 (GIS/2) Application Description, IBM.

[3] R.G. Curtis, Description of the SACCS Data Management System, Mitre Corporation, ESD-TR-70-295, (Sept. 1970) 114 pp.

[4] R. Moulder, "An Implementation of a Data Management System on an Associative Processor," Proceedings of AFIPS, NCC 73 (1973), pp. 171-176.

[5] R. Moulder, C. Bruno, J. Ernst, P. Gilmore, "Associative Processor Data Management Research and Development", Goodyear Aerospace Corporation, GER-15806, (Dec. 1972), 138 pp.

INTRODUCTION TO THE ARCHITECTURE OF A 288-ELEMENT PEPE

Alf J. Evensen & James L. Troy
Huntsville Operations
System Development Corporation
Huntsville, Alabama 35805

Abstract -- The PEPE (Parallel Element Processing Ensemble) is a parallel-associative processor which can attain order-of-magnitude performance and cost-effectiveness improvements over conventional machines when employed on problems containing inherent parallism. This paper describes the architectural features of a new large-scale PEPE system now being constructed to operate with a CDC 7600 Host.

General Description

When compared with conventional sequential multi-processing computers, PEPE provides much faster data processing rates. It does this at relatively low cost and with inherent reliability since its architecture is made up largely of disconnectable, rather simple but triple-processing element modules which are replicated many times throughout the design. Failure in any one element affects neither the remaining hardware nor the software.

Each PEPE element may simultaneously respond to instruction execution microsteps from each of three control units. Therefore, a 288-element PEPE may effectively execute up to 864 instructions simultaneously.

Elements may be added to the configuration if required, with no effect on the software. The capability of associative addressing allows the software to be indifferent to the number of elements that are present. Individual elements are activated or deactivated from participation in the execution of an algorithm based upon comparisons of sequential and/or parallel data.

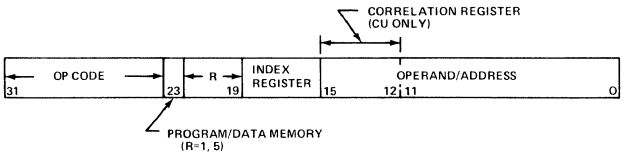
PEPE uniquely puts its parallel processing capabilities to work by providing completely overlapped input and output functions. The current large-scale PEPE model provides architecture to interface the parallel processing environment with a computing world which is sequentially oriented. Input/output conversion units, an input correlation control unit and an associative output control unit are utilized to allow the parallel arithmetic architecture to execute virtually without I/O overhead. Within the correlation and associative output control units data are block transferred from and to external devices simultaneously with the transfer of other data into or out of selected elements. The PEPE then is a complete parallel data processing system providing an unrestricted throughput relative to its parallel arithmetic capabilities (see Figure 6). [1] [2]

Host Interface

Although the current model PEPE will contain its own instructions, programs, interrupt mechanisms, clocks, etc., a close interface with a standard sequential computing system is desirable for quickly processing non-array-oriented portions of problems and for peripheral device control. This sequential system may also be used for utility functions such as compiling PEPE programs. For these purposes, the current PEPE configuration will utilize the ABMDA Research Center CDC 7600 computer which is connected to PEPE through three MUX (Input/Output Multiplexor) channels.(a)

PEPE Instructions

Both sequential control and parallel instructions can be intermixed in a program unit. The sequential instruction repertoire is required for program control functions and includes branching, I/O, active element count, and a limited data conversion capability (shift, mask, integer arithmetic). The parallel instruction repertoire includes two types of instructions: those which select element activity, and functional instructions such as floating point and integer arithmetic, shift and mask. The floating point capability in the Arithmetic Units includes floating point - integer conversion instructions and a square root instruction (see Figure 1). The 32-



R - ROUTING CODE

SEQUENTIAL INSTRUCTIONS:

0 - INSTRUCTION (23-0) MODIFICATION
1 - OPERAND FROM PROGRAM/DATA MEMORY
2 - OPERAND FROM INSTRUCTION (15-0)

PARALLEL INSTRUCTIONS:

3. OPERAND FROM ELEMENT MEMORY
4. OPERAND FROM CONTROL UNIT A-REGISTER
5. OPERAND FROM PROGRAM/DATA MEMORY
6. OPERAND FROM INSTRUCTION
7. OPERAND FROM PROGRAM MEMORY (PROGRAM COUNTER PLUS ONE)

Fig. 1. PEPE Instruction Format

(a) 7600 PPUs (Peripheral Processing Units) are not utilized for this connection because of execution time penalties.

This work was supported by the U.S. Army Advanced Ballistic Missile Defense Agency (ABMDA), Huntsville, Ala., under Contract DAHC60-73-C-0060.

bit instruction format has an 8-bit op code field (O), a 1-bit memory unit selection field (M), a 3-bit routing field (R), a 4-bit index register field (X) (there are 15 index registers in each control unit), and a 16-bit address field (A). The routing field determines the source of the operand and whether the instruction is sequential or parallel. The instruction "Load A-register," for instance, can cause the sequential control unit A-register or one or more parallel A-registers to be loaded depending upon the routing field setting. If a parallel routing is specified, parallel element A-registers will be loaded only in "active" elements set by a previous "select" instruction.

Detailed Description

Physical Configuration

The PEPE design will accommodate 288 processing elements partitioned into eight element bays. The element bays are installed radially to reduce cable length. Current plans call for the installation of only one element bay containing 36 processing elements. All processing element operations are controlled from the control console which also provides the interfaces required for connection to the CDC 7600 and test and maintenance equipment. PEPE will be implemented with standard emitter-coupled logic (ECL) contained in

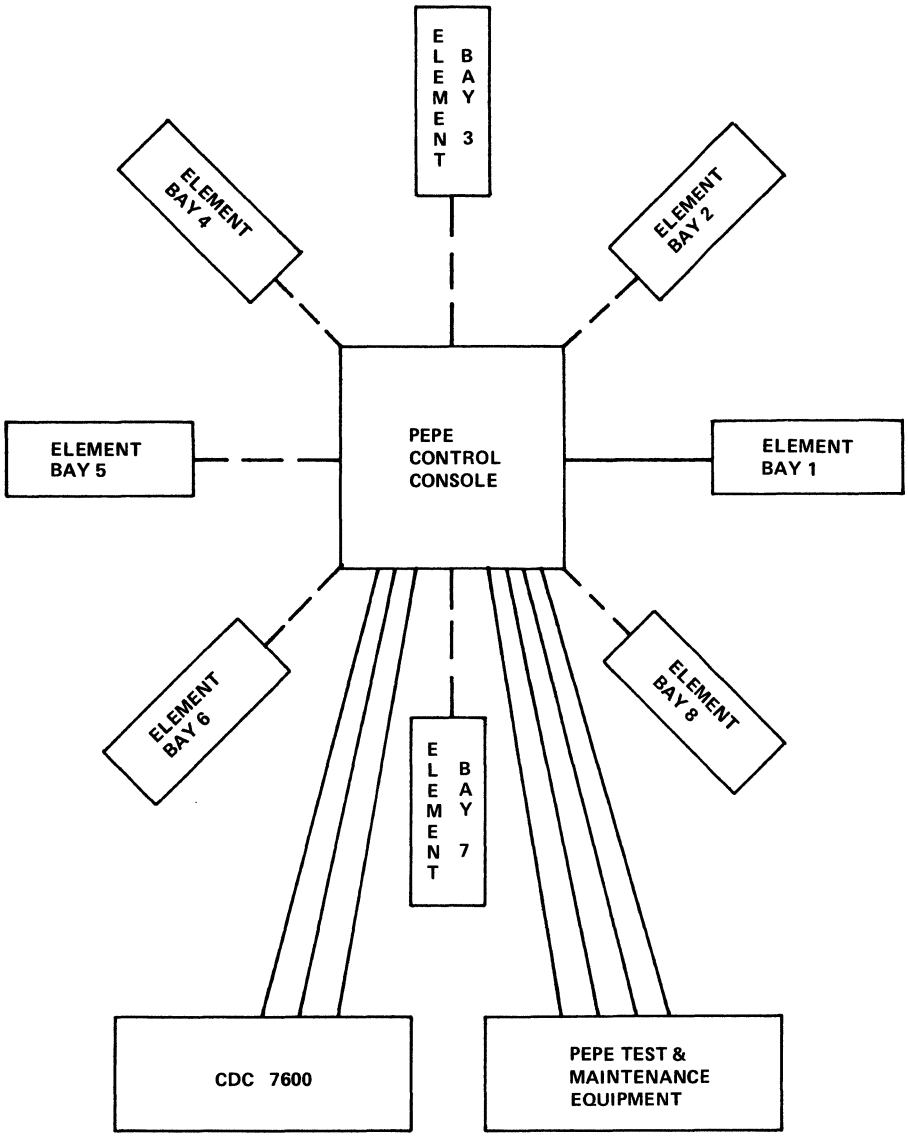


Fig. 2. PEPE Physical Configuration

Dual In-Line Packages (DIPs) which are mounted on multilayer printed circuit boards. The printed circuit boards will be approximately 16" x 18" and will have an average component density of 275 DIPs in the element bay and a maximum of 150 DIPs in the control console. PEPE will be cooled by means of forced air and chilled water. Figure 2 illustrates the PEPE physical configuration.

Control Console. The control console dimensions are approximately:

80" high
50" wide
26" deep

Power dissipation is approximately 6000 watts.

Element Bay. The element bay dimensions are approximately:

80" high
78" wide
26" deep

Power dissipation is approximately 30,000 watts per element bay.

Test & Maintenance Equipment. A Burroughs B1714 computer will be utilized for dynamic test and maintenance of the PEPE system and its individual printed circuit boards and processing elements.

PEPE Processing Element

Each processing element (PE) contains an Arithmetic Unit, Associative Output Unit, Correlation Unit and Element Memory as shown in Figure 3. The PE contains no instruction execution control logic and must receive all timing and control signals from the control console.

The ensemble of processing element units receives timing and control signals from corresponding control console execution units as follows:

PE Unit	Control Console Unit
Arithmetic Unit	Arithmetic Control Unit
Associative Output Unit	Associative Output Control Unit
Correlation Unit	Correlation Control Unit
Element Memory	Element Memory Control

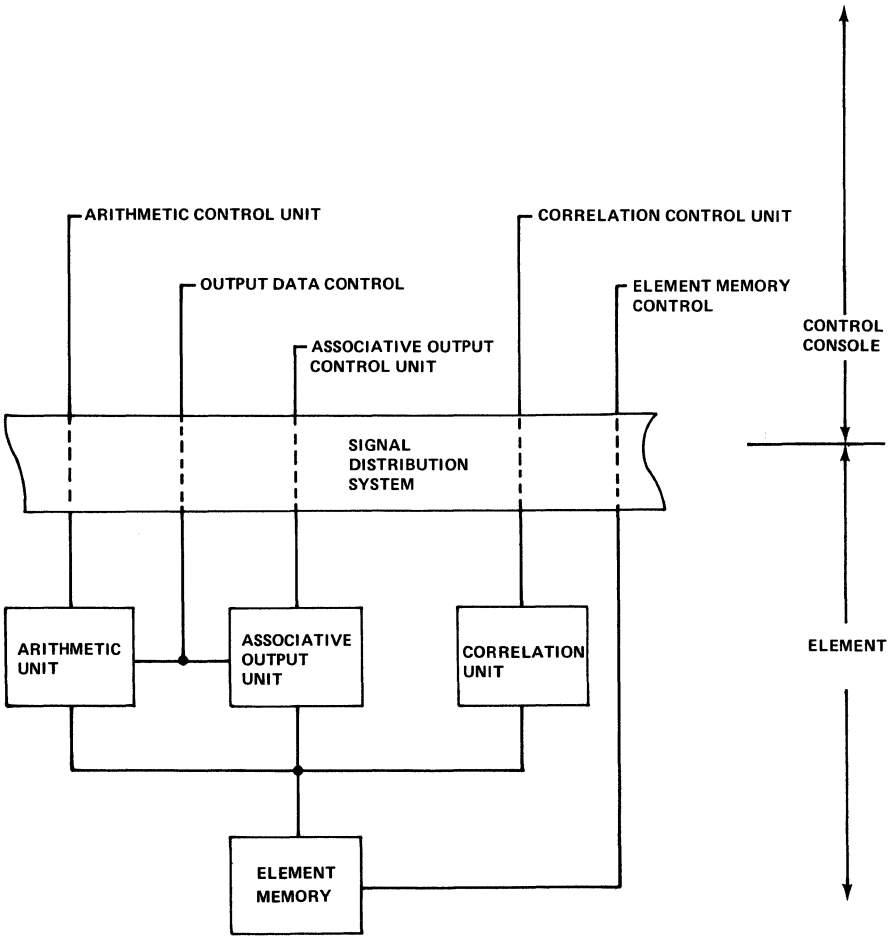


Fig. 3. PEPE Processing Element

Each PE unit (except element memory) contains an Activity register (one bit). When a control unit performs a parallel instruction all corresponding active (Activity register = "1") PE units respond so that a maximum of 288 PE units may simultaneously execute that instruction. Since the processing element contains three computational units corresponding to three independent control units, an ensemble of 288 PEs may be responding to three simultaneous and independent parallel instructions thereby effectively executing 864 simultaneous (subject to element memory conflicts) instructions.

An Activity Stack has been added to the Arithmetic Unit and Associative Output Unit. It is a 21-level hardware implemented "push-pop" stack connected to the Activity register. The Activity Stack is used to save and restore multiple subsets of PE units.

All processing element units contain an 8-bit, bit addressed Tag register which is used to perform associative matches on data received from the control unit.

Arithmetic Unit. Each Arithmetic Unit (AU) contains conventional A (accumulator), B (operand) and Q (quotient or product) registers which support execution of the parallel integer, logical and floating point instructions. The AU A-register is additionally utilized to provide associative output to its control unit via a data bus shared with the Associative Output Units. Various "select" instructions operate upon the AU Activity register, Activity Stack and Tag register to determine which Arithmetic Units participate in subsequent parallel instructions and to remember and restore previously active element sets.

Associative Output Unit. Each Associative Output Unit (AOU) contains conventional A and B registers which support execution of parallel integer and logical instructions. The AOU A-register is additionally utilized to provide associative output to its control unit via a data bus shared with the Arithmetic Units. Various "select" instructions operate upon the AOU registers exactly as in the AUs.

Correlation Unit. Each Correlation Unit (CU) contains a B-register and 16 Correlation registers (contained in a 16-word ECL RAM). These registers support execution of parallel integer and logical instructions. Correlation register-to-register operations are permitted. No means are provided for the CU to output data to its control unit. Various "select" instructions operate upon the CU Activity register and Tag register to determine which Correlation Units participate in subsequent parallel instructions. There is no Activity Stack in the CU since the correlation process requires the rapid identification of elements in which to store new data, rather than maintenance of a history of previous sets of activity as in the AUs and AOUs.

Element Memory. Each element memory (EM) consists of 1K words of ECL storage and receives address and mode information from the control console Element Memory Control (EMC). All ensemble EMs receive identical information from EMC during execution of a particular parallel instruction. EM is connected to the AU, AOU, and CU by means of a common data bus and consequently EMC directs the sharing of element memory with the following priority assignments: (1) CU, (2) AOU, (3) AU. This priority scheme has been established since the CU instructions tend to be short (200-300 nanoseconds) and the AU instructions tend to be considerably longer (floating point multiply requires 1.9 microseconds). Program execution times are expected to increase by no more than 5% due to element memory conflicts. Simulation experiments have shown that reversing the priority order greatly increases program execution times.

PEPE Control Console

The control console provides instruction execution control for the entire PEPE. It contains three control units (see Figure 4) which are connected to the ensemble of processing elements as described above. Additionally, the control console contains functional units which support the following operations:

- Inter-control unit interrupts
- Error recovery
- Processing element output
- Element memory conflict resolution
- Maintenance and diagnostic tasks
- Input/Output data conversion

The system function of each control unit is:

- ACU - Manipulates the parallel data base contained in the ensemble of element memories.
- AOCU - Outputs data resulting from parallel data base manipulations.
- CCU - Inputs new data.

Control Units. The three control units (ACU, AOCU, CCU) are of a common design which is functionally configured as shown in Figure 5. Each control unit has its own program and data memory. Communication between control units may occur via the Intercommunication Logic Unit (ICL) as illustrated in Figure 4.

Programs are executed from the program memory and consist of any sequence of:

- sequential instructions to be executed in the Sequential Control Logic (SCL)
- parallel instructions which are routed through the SCL to the Parallel Instruction Control Unit (PICU)

The SCL contains conventional A,B,Q and index registers which support execution of sequential integer, logical and branch instruc-

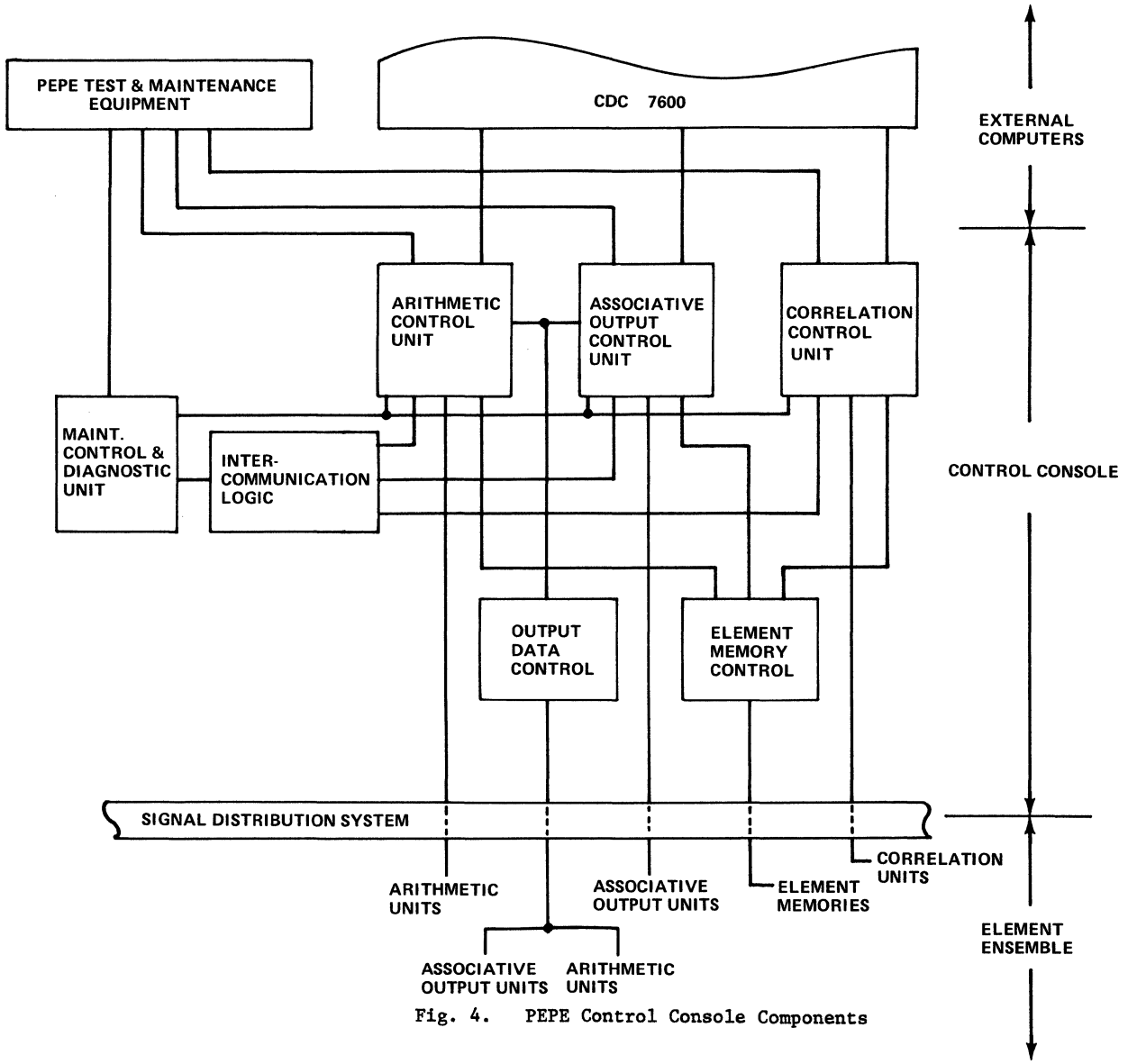


Fig. 4. PEPE Control Console Components

tions. It also responds to parallel instructions which:

- cause output from the PE (except CCU)
- allow branching based upon element activity
- cause inter-control unit interrupts
- support error recovery

Sequential instruction operands may be contained either in the instruction or in program/data memory as specified by the appropriate instruction fields.

Parallel instructions (with indexable operands) are routed to the PICU which is a microprogrammed execution unit in which the microprogram memory outputs are utilized to control the switching networks in the processing element. When required during execution of a parallel instruction the PICU transmits address, request and mode data to element memory control. It then transmits a data strobe to the PE when an acknowledge is received from EMC indicating that the PICU has been selected for EM service. Parallel instruction operands may be contained either in the instruction or in element memory as specified by the appropriate instruction fields.

Because of its large (32K) program memory, ACU cycle time is a relatively slow 200-300 ns (other program/data memory cycle times are 100ns). Moreover, the ACU has responsibility for execution of relatively slow parallel floating-point instructions, so the ACU parallel instructions are routed through a 16-step queue (Parallel Instruction Queue) prior to execution in the ACU-PICU. This queue effectively speeds up average ACU instruction rate.

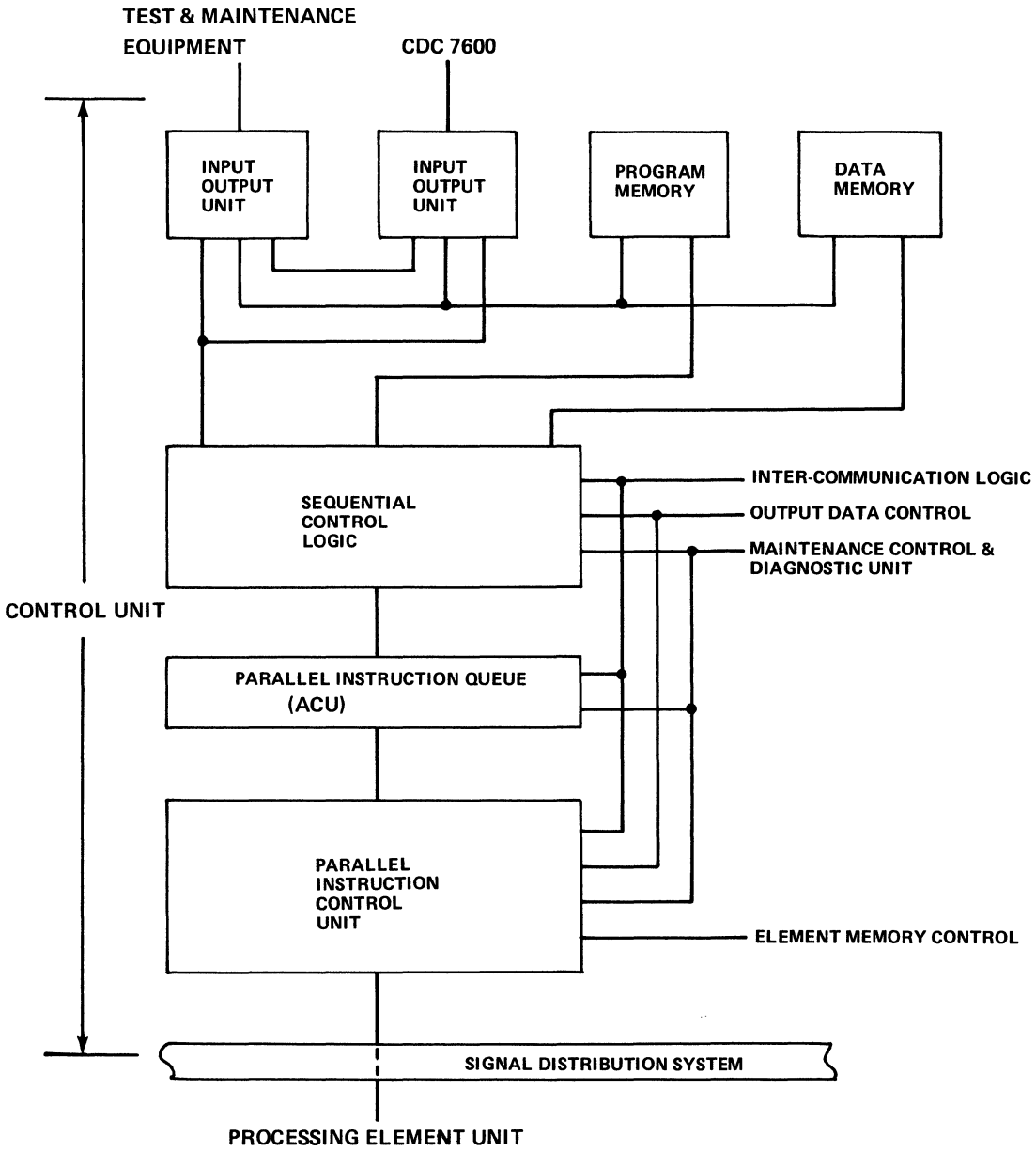


Fig. 5. PEPE Control Unit

Input/Output. Each control unit has two Input/Output Units (IOU) to provide for control and fully duplexed data transfer to and from the CDC 7600 and test and maintenance equipment. These IOUs are capable of:

- Block transfer of data to control unit program/data memory initiated by CDC 7600 (T&M equipment)
- Block transfer of data from (to) control unit program/data memory to (from) CDC 7600 T&M equipment) initiated by sequential instruction execution
- Control unit interrupts
- Control unit start/stop (master clock)

IOU capability has been expanded to allow overlap of IOU data transfer with parallel/sequential instruction execution. This feature alone is responsible for halving the time it takes to correlate new data received by the CCU with existing data residing in element memory.

Element Memory Control (EMC). EMC receives requests from the three control unit PICUs for element memory service. It performs any needed conflict resolution, transmits required control information to the ensemble EMs and responds to the PICU when the selected EMs have been properly switched to service the AU, AOU, or CU.

Output Data Control (ODC). ODC receives requests from the ACU/AOCU to transfer AU/AOU A-register contents to the A-register in the ACU/AOCU SCL. It performs conflict resolution and places the active AU/AOU A-register contents on a common data bus to the control console. ODC then transmits an acknowledge to the ACU/AOCU SCL to achieve the data transfer. More than one AU/AOU in an active state will cause an error condition to be processed by the Inter Communication Logic.

Inter Communication Logic (ICL). ICL provides the mechanism for:

- AOCU interrupt of the ACU
- CCU interrupt of the ACU
- Control unit interrupts from IOU
- ACU control of control unit registers
- Error interrupts from ACU
- Real-Time Clock
- Interval Timer
- System data collection

Neither the AOCU nor the CCU have floating point instructions. Therefore, they have been given the capability to interrupt the ACU in order to execute subroutines which require floating point manipulations. The ICL prevents interrupt "nesting" by either AOCU or CCU, and contains four registers (two each for the CCU and the AOCU) which may be utilized for inter-control-unit interrupt data transfer. Provision has been made for the inclusion of a 1K-word ICL memory in the event that extensive inter-control unit communication becomes necessary.

Each Input/Output Unit transmits control unit interrupt requests to the ICL. Three registers (one for each control unit) provide the means of transmitting an interrupt message to the control units with each interrupt request from either the CDC 7600 or the test and maintenance equipment.

Error conditions within the PEPE signal the ICL to generate an error interrupt to the ACU. An error identification code is placed in an ICL register.

The ACU SCL error-recovery software utilizes supervisory instructions which can read and write all control unit registers to and from the A-register in the ACU SCL.

A Real-Time Clock (46 bits) and Interval Timer (24 bits) are contained in the ICL. Both count with 100ns granularity and are fully accessible from all control units. An ACU interrupt may be generated when:

- The Interval Timer decrements to zero
- The Real-Time Clock equals the value contained in the Real-Time Clock Buffer (fully accessible from all control units)

Eight counters (24 bits) are available in the ICL for monitoring software/hardware performance.

Maintenance Control and Diagnostic Unit (MCDU). The MCDU is the diagnostic interface which couples the test and maintenance equipment, the PEPE control console, and the maintenance technician.

Future Development

Although the PEPE Program is continuing under the direction of ABMDA for the purpose of developing an advanced ballistic missile defense system, nonmilitary applications for PEPE have been studied with the permission of ABMDA. These applications could include air traffic control, satellite tracking, auto traffic control and weather data processing.

References

[1] J.A. Cornell, "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE," COMPCON 72 Computer Conference, 1972

[2] J.A. Cornell, "PEPE Application and Support Software," Western Electronic Show and Convention, 1972

[3] M.D. Johnson, "The Architecture and Implementation of a Parallel Element Processing Ensemble," Western Electronics Show and Convention, 1972

[4] C.Y. Lee, "Intercommunicating Cells, Basis for a Distributed Logic Computer," Proc. Fall Joint Computer Conference, 1962

[5] C.Y. Lee and M.C. Paull, "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," Proc. of the IEEE, June 1964

[6] B.A. Crane and J.A. Githens, "Bulk Processing in Distributed Logic Memory," Proc. of the IEEE, April 1965

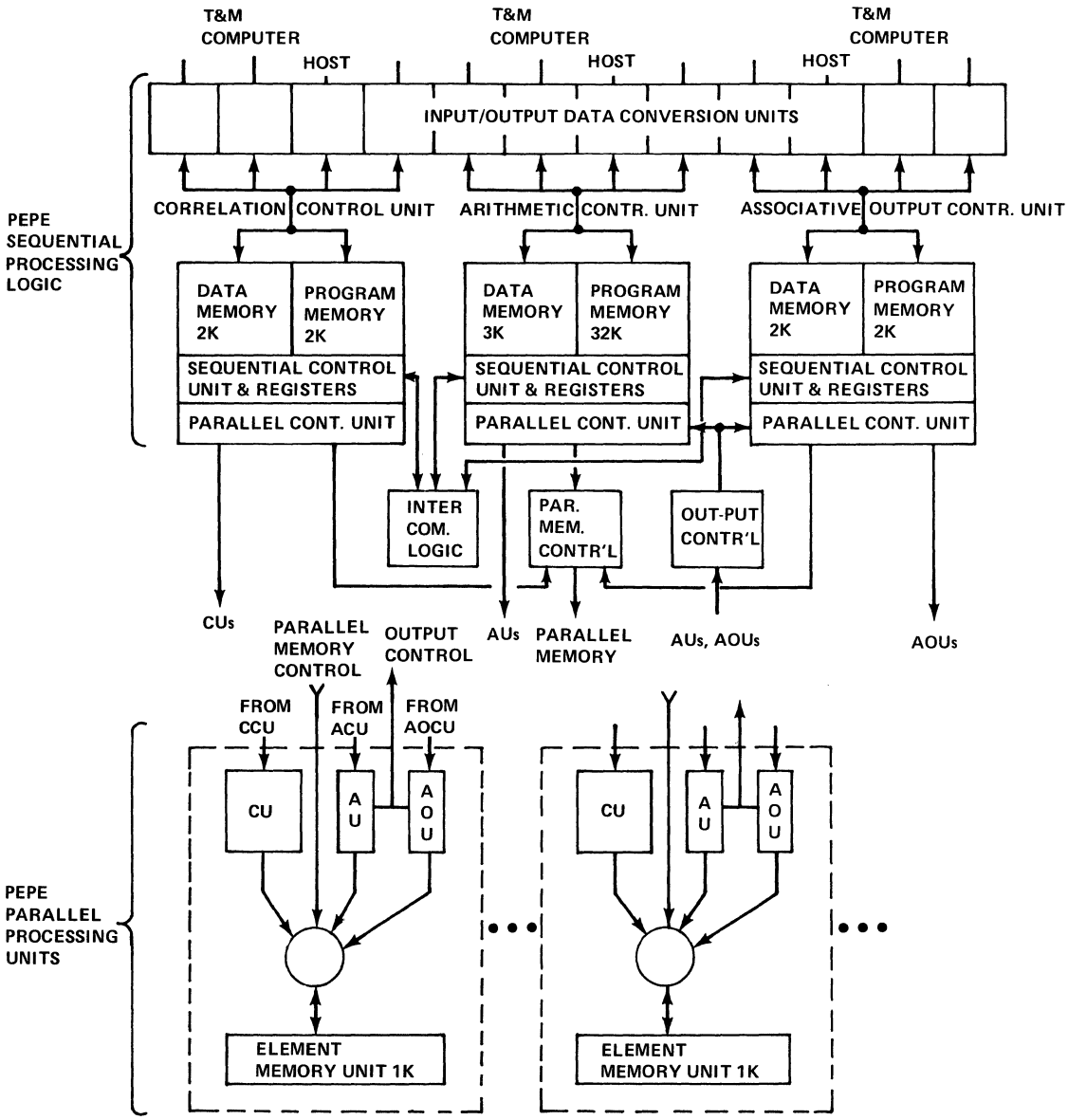


Fig. 6. PEPE Architecture

OPERATING SYSTEM AND SUPPORT SOFTWARE FOR PEPE

J.R. Dingeldine, H.G. Martin, W.M. Patterson
Huntsville Operations
System Development Corporation
Huntsville, Alabama 35805

Abstract -- Software for the CDC 7600-PEPE configuration consists of a constructable real-time tactical process and the support software required to develop and execute the real-time process. This paper discusses: 1) the development and the real-time characteristics of the Operating System; 2) the procedure oriented language, Parallel FORTRAN (PFOR), used to develop tactical programs; and 3) the PFOR Translation System. A PEPE instruction level simulator and the process constructor are covered by other papers in this set. [3] [4]

Operating System Software

Figure 1 is a simplified picture of the PEPE system and its host. Three bi-directional communications paths connect the host computer (CDC 7600) with each of the PEPE controllers. The system is a network of controllers each of which requires compatible parts of the operating system. These interfacing parts, together with real-time executive functions, are the subject of this section. Comments are made on design goals, real-time executive functions, process execution control tables, and system performance under functional simulations.

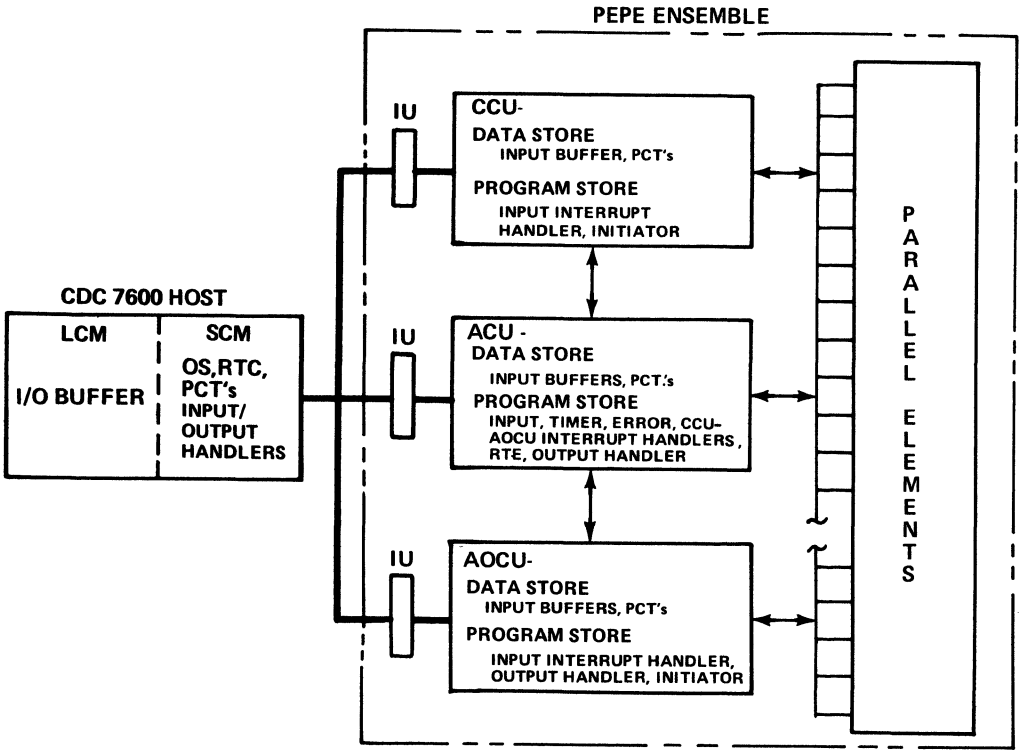


Fig. 1. PEPE/CDC 7600 Computer Network

This work was supported by the U.S. Army Advanced Ballistic Missile Defense Agency (ABMDA), Huntsville, Ala., under Contract DAHC60-73-C-0060.

1. SIMPLICITY TO PROMOTE EASE OF USE
2. RESPONSIVE TO EXTERNAL STIMULI
3. FLEXIBLE
4. TABLE DRIVEN FOR CONSTRUCTABILITY
5. TASK TRIGGERING IN RESPONSE TO
   - TIME EVENTS
   - DATA EVENTS
   - SETTINGS OF SETS OF CONDITIONS
   - DIRECTLY BY OTHER TASKS (DYNAMIC)

Fig. 2. Real-Time Executive Design Goals

Following experience with the previous PEPE feasibility model and study of other existing and proposed multi-computer operating systems, a preliminary operating-system model was designed with the goals listed in Figure 2 in mind. A functional simulation model was generated to aid in evaluating the system's effectiveness. The results from the runs revealed excessive executive and interrupt handling times. Response to external stimuli was poor. The basic design was flexible enough and table driven as required, but the interrupt and overhead tasks were time consuming.

A second simulation model was generated with emphasis on simplicity in the hope that flexibility and responsiveness would follow. The current system is an outgrowth of the second model. Figure 3 is a flow chart of the real-time executive loop. It has only three steps: (1) If there has been a change in the status of any condition, make any indicated enablements using process control tables; (2) Select the highest priority task; (3) If a task is selected, clear the software interrupt flag, and call the task. When the task is completed, it returns control to the executive and the cycle is repeated. This basic executive cycle is supported by interrupt handlers and output routines which accomplish process control table changes when messages requiring actions are intercepted.

Timing tests on CDC 7600 code for the executive produced favorable results. Conditional enablements, step (1), were made in 4.620 microseconds using a three entry table. Task selection, step (2), required 4.950 microseconds, while task initiation, step (3), used 4.263 micro-seconds. So, a task may be running in the host within 10 microseconds after it is enabled. This quick response is due to the simple structure of the process control tables, Figure 4.

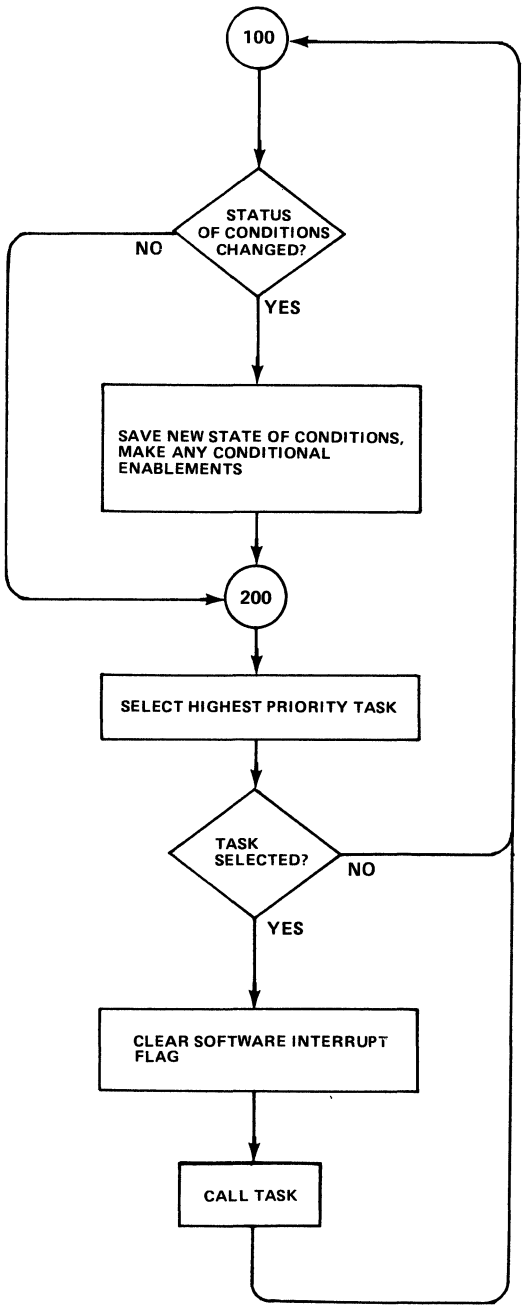


Fig. 3. PEPE/CDC 7600 Real-Time Executive

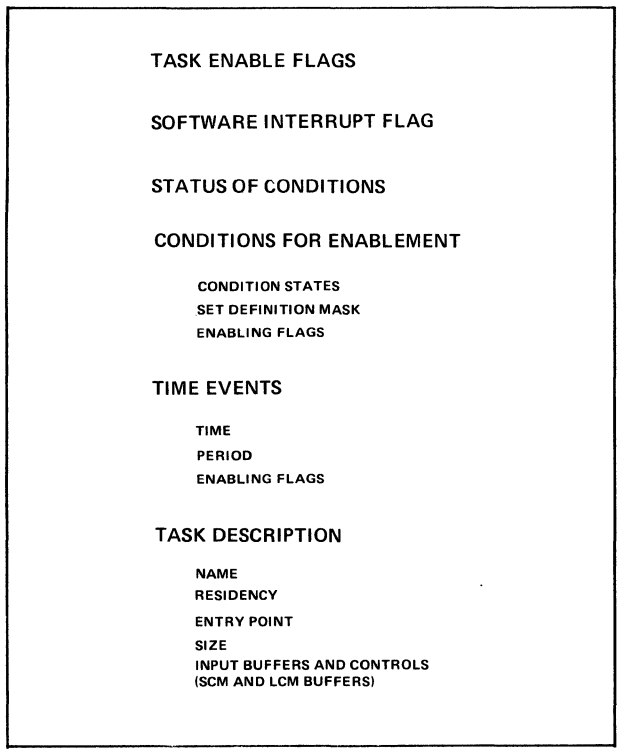


Fig. 4. PEPE Process Control Tables

The Task Enable Flags are arranged in priority order. The first flag found in the On State represents the next task to run. The Software Interrupt Flag is set any time a task is enabled which has a higher priority than a running task. Any task which runs longer than the software interrupt interval (say 250 microseconds) must enable itself, adjust controls for continuing its operation later, and return control to the executive. The status of conditions is maintained by the running tasks. The Conditions for Enablement Table has three parts per entry: a set of condition states, a mask for selecting the set of conditions, and task enabling flags (tasks are identified by flag positions as in the Task Enable Flags Table). All entries in the table are processed when a change in the status of conditions is detected. The Time Events Table is a chronological list of scheduled time events with associated periods between events and task enabling flags. The Task Description Table identifies the task name, size, and location in task priority (number) order. A buffer pointer leads the executing task to its input data.

The process control tables are accessible to the running task, the executive, and all interrupt handlers. Tasks are triggered in other controllers by messages in standard form. Time Event Change and Task Enable messages are completely processed by the message handlers.

The simplicity of the real-time control process permits similar control mechanisms in all system controllers (see Figure 1). The capabilities of the controllers dictate the amount of local executive control. The PEPE Arithmetic Controller has full interrupt features with an interval timer and error interrupts. It, therefore, has full real-time executive controls. The other two PEPE controllers have only input interrupt, task initiation controls, and output handler features.

It is interesting to note that with the redesign of PEPE to include program storage, the total system executes as a sequential computer network. The only operational difference is shorter execution times. Thus, changes to the host's commercial operating system are required only to support the input/output channels and for the addition of a real-time interval timer.

The real-time controls as described accomplish the original design goals (Figure 2) favorably. The simplicity is illustrated by the executive (Figure 3) itself. Responsiveness results from the simple requirement of the interrupt handlers. The time controls are maintained in time order to eliminate time consuming searches or sorts. Up to 48 tasks may be enabled by one condition table entry. The task triggering methods together with the rapid response to enablements permits efficient calls to scheduling algorithms or deadline functions. For example, a deadline task may be time enabled when the deadlined task is scheduled. If the task executes before the deadline, it simply deletes the time table entry which would trigger the deadline action. The table structure obviously permits many process construction forms such as enable tasks, set time event, set/reset conditions, etc. Simulation model testing and actual instruction timings conducted at the ABMDA Research Center in Huntsville substantiate these statements.

The simplicity of the real-time controls coupled with the ease of operating with the redesigned PEPE appear to have produced an efficient and effective real-time system.

Support System Software - Parallel FORTRAN (PFOR)

Overview

The Parallel FORTRAN (PFOR) sections of this paper emphasize language extensions and changes to the PFOR Translation System developed since the presentation of papers on the Parallel Element Processing Ensemble (PEPE) and its support software at COMPCON 72 and WESCON 72 [1], [2]. The referenced papers describe the basic PFOR language and language processors as implemented on the laboratory PEPE IC (Integrated Circuit) model used to demonstrate the feasibility of PEPE in a Ballistic Missile Defense (BMD) environment.

PFOR Language

PFOR is a procedure oriented, higher order FORTRAN-like language tailored to the new PEPE MSI (Medium Scale Integration) model. The language consists of: 1) PEPE FORTRAN, the minimal subset

of standard FORTRAN required for the sequential control of parallel algorithms in the PEPE Sequential Control Logic (SCL) hardware, 2) Parallel FORTRAN or PFOR, the extensions to FORTRAN for the declaration of data and the parallel and associative processing of data in the PEPE elements, and 3) PEPE Assembly Language (PAL) machine instructions, extended mnemonics, and pseudo operations.

PFOR is currently being used to develop PEPE tactical processes. It is the sole source language for the three PEPE control units with unique machine code generated by the compiler for each control unit; whereas for the PEPE IC model, PFOR was used to program only the Arithmetic Control Unit (ACU). A macro assembly language (CUAL) was used to program the Correlation Control Unit (CCU) and sequential control was exercised in the host IBM S/360-65. The IC model hardware did not contain an Associative Output Control Unit (AOCU). The capability of intermixing PFOR, FORTRAN, and PAL statements in a source program has been retained. For the MSI model the PAL assembly language statements are bracketed by the PFOR primitives MODE(DIRECT) and MODE(PFOR). Each block of PAL code is processed as a single PFOR source statement.

PEPE FORTRAN

The FORTRAN declarative statements, imperative statements, and logical, relational, and arithmetic operators defined for sequential execution in PEPE are listed in Figures 5 and 6.

The arithmetic operators * and / are not defined since the sequential portion of the PEPE hardware supports only 24-bit integer addition and subtraction. Address (16-bit) multiplication is implemented in the sequential hardware which allows the compiler to generate efficient code for array references which contain variables in the array subscript.

A minimal subset of standard FORTRAN required to exercise sequential control of parallel, tactical processes has been defined for PEPE.

FORTRAN REAL variables are limited in use since the SCL hardware does not perform floating point operations. These variables are used during data transfer between the host and the ensemble.

PFOR Extensions

The PFOR language has previously been described by Wilson [1] and Cornell [2] at COMPCON 72 and WESCON 72. The basic PEPE IC model PFOR primitives and operators which have been retained for implementation on the MSI model are listed in Figure 7.

Several features have been added to the PFOR language for the PEPE MSI model. To support parallel double precision integer arithmetic, the data description forms PAR DOUBLE (element memory double word) and PAR COR DOUBLE (correlation

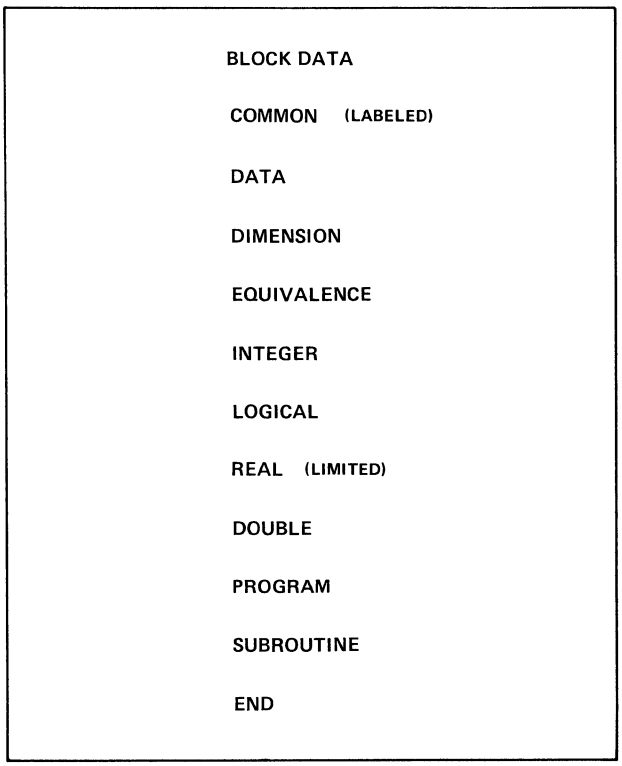


Fig. 5. PEPE FORTRAN Subset Declaratives

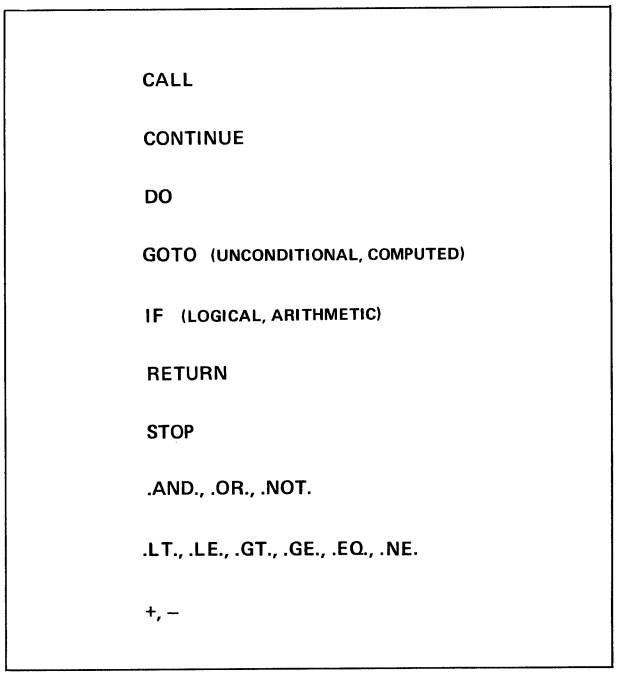


Fig. 6.
PEPE FORTRAN Subset Imperatives and Operators

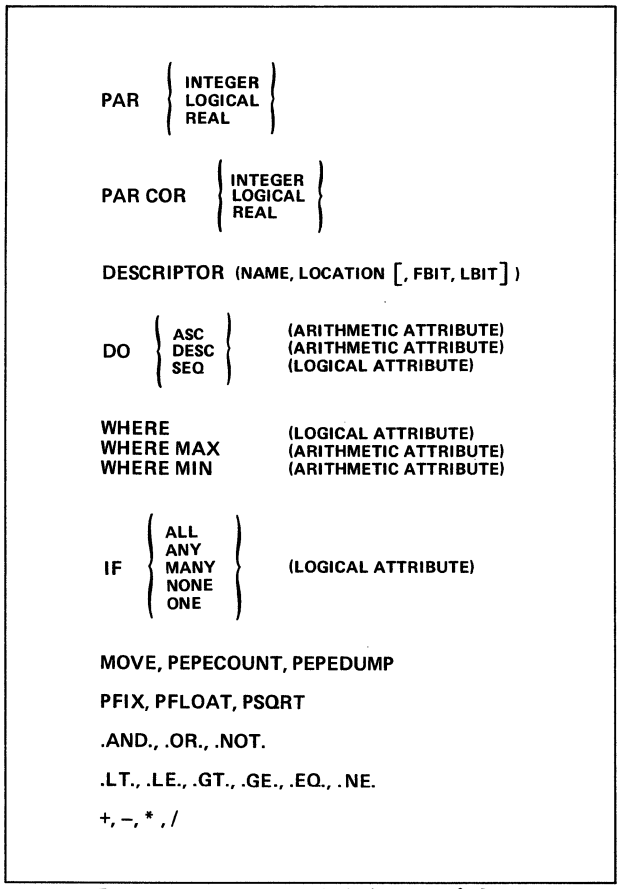


Fig. 7. Basic PFOR Primitives and Operators

register file double word) have been added. For example,

```
PAR DOUBLE  PV
```

declares PV to be a double precision, integer parallel variable. In all examples in this paper, names prefixed by the letter P denote parallel variables.

The set of PFOR statements which allow ordered selection of PEPE elements in sequential, ascending, and descending fashion (DO SEQ, DO ASC, and DO DESC) select the elements one at a time. These have been augmented by the DO UP and DO DOWN constructs which allow sets of elements to be utilized in an ascending or descending manner. The code sequence

```
      WHERE  (PTEST)  100
      DO UP  100  (PV)  10,I
100   PW = I
```

acts like a DO ASC statement if, in the elements passing the test (the set of elements remaining active by virtue of the parallel logical variable PTEST being true), the ascending algebraic values of PV are unique. In this case in the active element where PV has the lowest algebraic value, PW is set to one; in the active element where PV has the next lowest value, PW is set equal to two, etc. However, if PV is duplicated in one or more of the elements passing the test, a "tie" exists. Assume for example the two lowest algebraic values of PV are identical in the active set of elements. The DO ASC construct causes PW to be set equal to one in the first physically available active element of the two. PW is set to two in the other element and then looping continues with PW = 3, PW = 4, etc., until at most ten elements are looped over. Upon completion of the DO ASC loop, I is set to indicate the number of elements involved in the processing (i.e., elements where PW has been set) for there may be less than ten elements which passed the test.

For the DO UP construct, PW is set to one for both elements where the lowest value of PV is identical; then PW is set to two in the active element where PV has the next lowest value, etc.; until at most ten sets of elements are looped over. Upon completion of the loop, I is set to indicate the number of sets of elements involved in the processing (sequentially tagged in the example) for there may be less than ten sets of elements which passed the test.

The WHERE class of statements (WHERE, WHERE MAX, WHERE MIN), which are used to specify a content-addressed subset of PEPE elements, have been augmented by the addition of WHERE FIRST, WHERE NOT, WHERE SET, and CONVERGE constructs. The general form

```
WHERE (logical attribute)  s#
```

causes the subset of elements satisfying the given attribute to remain active and to participate in processing through the range of the statement labeled s#.

The WHERE class of statements illustrates the explicit associative aspects and the implicit parallel aspects of the PFOR language. The attribute parameter explicitly denotes the associative or content based addressing. The parallel execution of the statements in the range of the WHERE statement (to and including the statement labeled s#) is implicit.

The statements in the range of a WHERE FIRST statement (through s#) are executed only in the first physically available element of the active set. No attribute is specified. For example, suppose we wanted to calculate $(PZ)^2$ in one and only one element where PX is greater than PY. This could be realized by the program sequence

```
      WHERE (PX .GT. PY)   10
      WHERE FIRST  10
10    PZ = PZ * PZ
```

which calculates a new value of PZ in the first physically available element of the subset of elements which pass the test.

A WHERE NOT statement must reside within the range of a WHERE, WHERE MAX, or WHERE MIN statement and does not require specification of an attribute. The statements in the range of a WHERE NOT statement (through s#) are executed only in those elements made inactive by the preceding WHERE statement. Upon execution of the range terminating statement (labeled s#), the element activity reverts to the set of elements active by virtue of the preceding WHERE-type statement. Assuming a 288-element ensemble where 100 elements are active at the time the simple WHERE statement is executed and 75 elements of the 100 pass the test in the simple WHERE statement, the program sequence

```
      WHERE (PX .GT. PY)  10
      PFLAG = 1
      WHERE NOT  5
 5    PZ = PZ + 1
10    PZ = PZ * PZ
```

sets PFLAG equal to one and PZ = $(PZ)^2$ in the set of 75 active elements where PX is greater than PY and sets PZ equal to PZ plus one in the set of 25 elements made inactive by virtue of PY≥PX.

The WHERE SET construct allows the user to temporarily activate a set of elements. This may expand (or contract) the set of active elements as opposed to the typical nesting of WHERE statements which subset elements into smaller sets and reinstate the element activity level by level as the program reverts from inner levels to outer levels. Assuming a 288-element ensemble where 100 elements are active, 75 remain active by virtue of passing the test in the simple WHERE statement, and PW is greater than zero in 250 of the 288 elements in the ensemble, the program sequence

```
      WHERE (PX .GT. PY)  10
      PZ = PZ * PZ + 1
      WHERE SET (PW .GT. 0)   5
 5    PX = 0
      PY = 1
10    CONTINUE
```

computes a new value of PZ in the 75 elements of the set of 100 where PX is algebraicly greater than PY, sets PX equal to zero in all elements of the ensemble where PW is greater than zero, sets PY equal to one in the 75 elements of the set of 100, and then reverts the ensemble activity back so that the original 100 are active. The normal nesting of WHERE statements allows subsetting of element activity and, when the range terminator statement labeled s# has been executed, the previously active set of elements becomes reactivated.

A simpler, faster-executing content-addressable method of subsetting element activity has been implemented for CCU-targeted program units in the form of the CONVERGE construct. The CONVERGE statement is used for typical correlation algorithms which involve short iterations of code.

The general form of the CONVERGE statement is

CONVERGE (logical attribute) s#

CONVERGE statements may be nested but the range terminator statement labeled s# must be identical for each CONVERGE statement in the nested set. The element activity reverts to its previous state once for each set of nested CONVERGE statements; whereas, for WHERE-type statements the element activity is restored following the range terminator statement of each WHERE in the nested set. A set of nested WHERE statements may also have the same range terminator statement label but, for each nested WHERE, code must be generated to revert the element activity step by step from inner level to outer level. The CONVERGE statement executes faster only when nested with other CONVERGE statements. The element subsetting of CONVERGE and WHERE statements can be illustrated by the following code sequences where the numbers in parentheses indicate the number of active elements:

```
          (100)                    (100)
   CONVERGE(PY .GT. PX) 15   WHERE(PY .GT. PX) 15
          (50)                     (50)
   CONVERGE(PM .LE. PN) 15   WHERE(PM .LE. PN) 10
          (30)                     (30)
15 CONTINUE                10 CONTINUE
          (100)                    (50)
                           15 CONTINUE
                                   (100)
```

The INHIBIT INTERRUPT and ALLOW INTERRUPT constructs are used to bracket short, fast-executing code sequences in ACU-targeted program units to inhibit Host, CCU, and AOCU interrupts. This feature has been implemented because PFOR programs are not re-entrant.

The MODE (processor) and MODE OFF (processor) constructs are used to bracket subordinate code sequences targeted for execution in another processor. The WHILE construct is used to specify a primary code sequence which is to continue execution in an overlap mode while the subordinate code sequence executes in another processor. In, for example, a CCU-targeted program unit, the program sequence

```
MODE (ACU)
PV = PX + PY
MODE OFF (ACU)
```

causes the ACU to be interrupted, the activity state of the CUs (Correlation Units) to be transferred to the AUs (Arithmetic Units), and the bracketed code sequence to be executed by the ACU under the control of the ACU resident Real-Time Executive Interrupt Handler. Also, in a CCU-targeted source program, the program sequence

```
 MODE (ACU)
PV = PX + PY
WHILE
CALL CORLAT
MODE OFF (ACU)
```

causes the ACU to be interrupted and the code sequence bracketed by MODE (ACU) and WHILE to be executed in the ACU. In an overlap fashion CCU resident subroutine CORLAT is called and executed. The statement following the MODE OFF (ACU) statement is executed only after both the ACU code sequence (bracketed by the MODE (ACU) and WHILE statements) and the CCU code sequence (bracketed by the WHILE and MODE OFF (ACU) statements) have been executed to completion.

The function PABS is used to obtain the absolute value of a parallel expression. The PEPESTAT construct allows the programmer to transfer the status of element activity from one control unit to another via the shared element memory and allows an ACU or AOCU targeted program unit to extend element subsetting beyond the normal limit of 21 levels.

The READ, WRITE, and WRITE-with-End-of-Record (WRITER) constructs are used to pass data between the CDC 7600 host and the PEPE SCL data memory. These constructs are PEPE oriented in that 1) no FORTRAN FORMAT capability is required, 2) data are converted by the hardware from CDC format to PEPE format or vice versa as specified in a PEPE resident control word referenced by the PEPE I/O machine instructions, and 3) data conversions are performed based on the PFOR type specification statements for the variables. Figure 8 lists the extensions to PFOR being implemented for the PEPE MSI model.

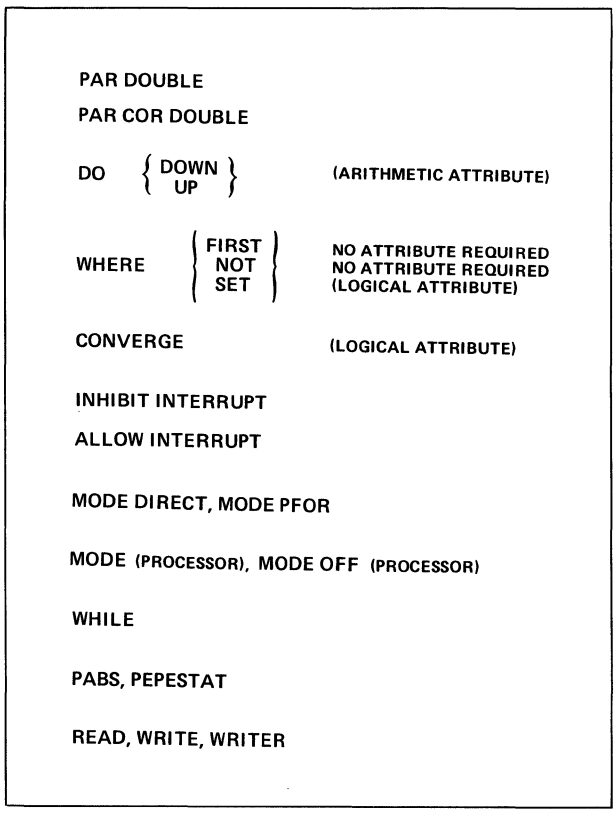


Fig. 8. PFOR Extensions for PEPE MSI Model

Additional features incorporated in the PEPE MSI model PFOR language allow ACU and AOCU targeted subroutines containing parallel statements to be nested to any level. Also, PFOR statements may be extended as continuation lines in the FORTRAN fashion.

The advantages of the PFOR language are that it is easy to learn (like FORTRAN), the associative aspects are explicit, the parallel aspects are implicit, and it is used as a common source language for the three PEPE processors.

Minor constraints include certain limitations on usage; i.e., some language forms cannot be utilized in all three processors because each processor has unique hardware designed for its particular application (input, processing, output). Moreover, the utilization of mixed parallel and sequential forms in a single source program is sometimes a source of confusion to programmers oriented towards sequential processing computer systems.

PFOR Translation System

Background

The PFOR language translation system for the laboratory PEPE IC model resided on the IBM S/360-65. It consisted of 1) PFOR Monitor, 2) PFOR precompiler, 3) PAL assembler, 4) S/360 FORTRAN compiler, and 5) S/360 assembler. The PFOR precompiler was a preprocessor which converted the PFOR source language to FORTRAN for execution in the host and PAL for execution in PEPE. FORTRAN source text was passed without error checking except to ensure that statement label references (GOTO,DO, etc.) did not conflict with PFOR construct context rules. The PFOR language translation system has been described in detail by Wilson [1]. Following is a brief overview. The PFOR preprocessor converted PFOR source text to FORTRAN and PAL. FORTRAN and PAL source text were not modified. The PAL assembler converted blocks of PAL code to S/360 assembly language named data sets and generated a FORTRAN call statement to a run time PEPE initiator routine (PINIT) for each named data set. The intermixed input and generated FORTRAN statements were passed to the S/360 FORTRAN compiler and the generated assembly language CSECT and DC pseudo operations representing the named data sets containing the blocks of PAL code were passed to the S/360 assembler. At run time, under control of the code segments executing in the host, blocks of PEPE instructions were streamed over a selector channel to the ACU for execution and PEPE data were returned to the host via the same channel following host invocation of the PINIT interface routine. The stream of instructions received by the ACU were decoded and broadcast one by one for simultaneous execution in the ensemble Arithmetic Units (AUs). The PFOR translation system processed only PEPE code destined for ACU execution. A separate Correlation Unit Assembly Language (CUAL) implemented as S/360 assembler language macros was used to generate blocks of PEPE code (as S/360 named

data sets) for transmission over another selector channel to the CCU.

Current Implementation Design

The PFOR language translation system for the PEPE MSI model is designed to execute on the CDC 7600 under the control of SCOPE 2.0. A common source language for the three PEPE control units is accepted, and the source text is converted to a triple object language, namely, unique parallel and sequential code for each target controller (ACU, CCU, and AOCU).

The PFOR compiler consists of two passes operating under the control of the PFOR monitor. The monitor performs control statement cracking to determine, for example, if the program unit (or batch of program units) is destined for execution in the ACU, CCU, or AOCU. Pass 1 of the compiler contains the first pass of a conventional two-pass assembler. Pass 1 also performs syntax analysis and grammar checking on the source text. A source listing, error diagnostics and a PEPE memory map are output to a list file. A file containing an encoded pseudo binary unit record for each assembly language statement (PAL source input or generated PAL) and a dictionary or symbol table file are prepared by Pass 1 for input to Pass 2, the assembly pass. The assembler generates an object listing and a cross reference listing of symbol utilization. A relocatable binary object module is generated which can be placed in a library file or directly input to the Process Consolidator for linkage and binding into a core image absolute binary load module suitable for loading (from the CDC 7600) and execution in PEPE.

ACU-executing code sequences embedded in a CCU or AOCU targeted source program (parent program) are placed on disk by Pass 1. Parallel variable entries in the symbol table are saved in compiler memory. When the compilation of the parent program is complete, each subordinate code sequence destined for ACU execution is processed as a separate, unique compilation utilizing the saved symbol table containing entries for parallel variables which reside in element memory. In PEPE a single element memory is shared by the three control units so parallel variables can be declared and referenced in both parent programs and subordinate code sequences. An overview of the PFOR Translation System is depicted in Figure 9.

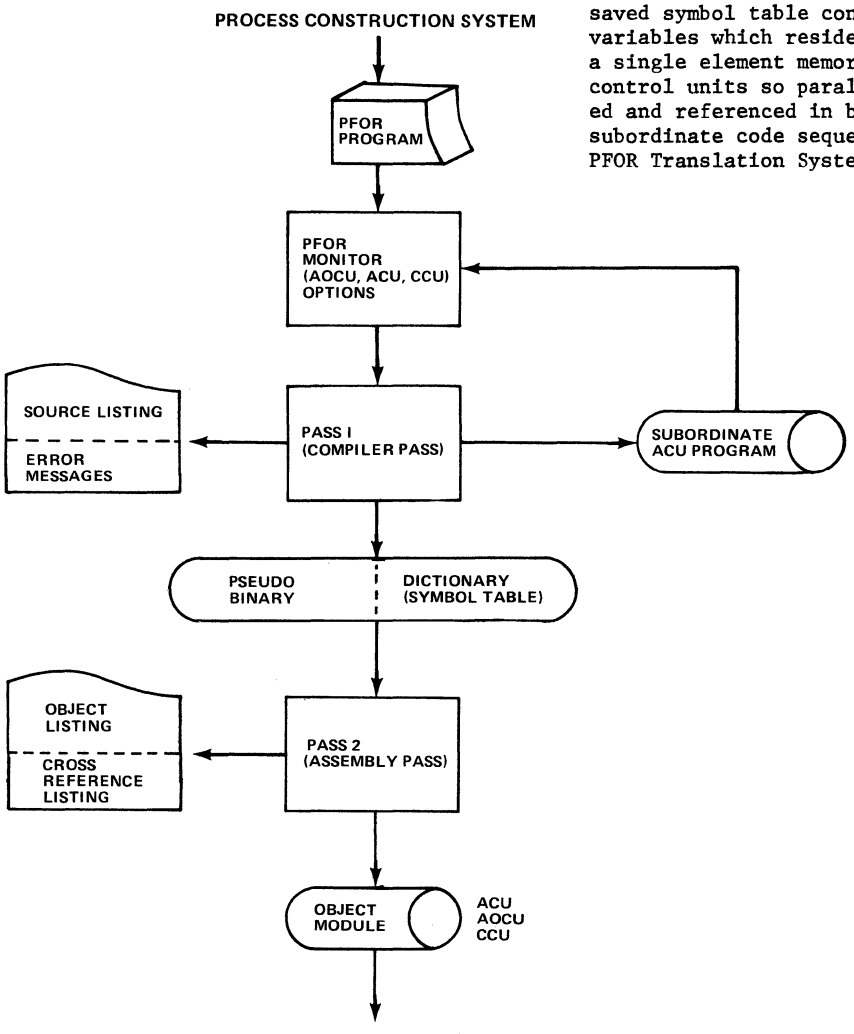


Fig. 9. PFOR Translation System

Summary and Conclusions

The PEPE MSI model PFOR compiler accepts a single source language and generates a triple object language. Language forms have been expanded to provide more flexibility to the tactical applications programmer. Using PEPE, an increase in data volume (as opposed to an increase in the complexity or sophistication of data manipulation) can be straightforwardly handled by increasing the number of elements in the ensemble. The PEPE software will still perform effectively with no changes; i.e., an increase in system capability obtained by adding more hardware is not necessarily accompanied by software breakage problems. This latter point is illustrated by the fact that in the PEPE IC model, tactical software for tracking targets was checked out using a 16-element simulator, run on the 16-element IC model hardware, and then transferred to a 100-element simulated ensemble with no changes.

References

[1] D.E. Wilson, "The PEPE Support Software System," IEEE Compcon 72 Digest (Sept, 1972), pp. 61-64.

[2] J.A. Cornell, "PEPE Applications and Support Software," IEEE Wescon 72 Digest (Sept, 1972).

[3] J.L. Troy, "Computer Simulation of PEPE and its Host at the Instruction Level," 1973 Sagamore Computer Conference on Parallel Processing.

[4] A.L. Barrett, "Process-Construction for a Parallel-Sequential Computer Architecture," 1973 Sagamore Computer Conference on Parallel Processing.

PROCESS-CONSTRUCTION FOR A PARALLEL-SEQUENTIAL COMPUTER ARCHITECTURE [1]

Arthur L. Barrett
Huntsville Operations
System Development Corporation
Huntsville, Alabama 35805

Summary

The purpose of process construction is to facilitate the transition from process design to operating process. Five successive states comprise this transition: modification, translation, compilation, consolidation, and operation.

The PEPE process constructor currently excludes the modification stage; it is performed by a commercial utility routine. Statements of design and implementation are updated and sorted to produce a file of process definitions and a file of source statements for the components of the object process. These are inputs to the translation stage.

Definitions are translated first to produce object statements defining the process data base and to provide information for the translation routines to use in handling the operative statements. The latter are translated as they are detected in the ensuing examination of the source file. Unrecognizable statements in that file are passed in proper sequence to either the PFOR [2] task file or the FORTRAN task file. This stage also produces a control file for the consolidation stage.

The process constructor invokes each of the compilers to produce a file of object modules; one for the PEPE and one for the host. The consolidation stage reads these and other files, such as the system subroutine libraries. Directives to the consolidator from the translator and information included in the object modules enable this stage to create modules to be loaded into each memory of the PEPE-host configuration and operation is begun.

The approach to tactical software development in the PEPE program is one of evolution from process design and functional simulation to live operation. As an aid to this approach the constructor uses the Software Development Language (SDL). This language meets the requirements of flexibility and ease of use through its syntax; keyword followed by parameter. Statements are formed from sequences of keyword-parameter sets. Three field delimiters each have the same meaning and are used interchangeably for readability. The simple, rigorous syntax enables the table-driven SDL translator to be highly generalized, thus the language is open-ended, requiring only changes or additions to the data in the translator's control tables for modification or addition of a statement to the language.

A tactical process design is described in SDL statements and such PFOR and FORTRAN statements as are needed to manipulate data for a functional simulation. The process is then constructed for simulation, merging the simulation package with the process. As the various routines of the process are implemented, their code is added directly to the source library. The code thus becomes part of the process and is executed during operation, but it does not affect the functional simulation. When the tactical code is fully implemented the process can be constructed for live operation. The translator recognizes statements that are peculiar to a simulation and removes them or, in some cases, provides a translation more appropriate to a live process.

The basic data entities are PARCELs (parallel cells grouped into PARTITIONs) in PEPE ensemble memories, and ELEMENTs. Both translate into variables and arrays of up to three dimensions, aside from the PARCEL's innate vector across the ensemble. The PARCEL also may be positioned and packed in bit groups smaller than word size with more than one parcel per word. ELEMENTs comprise QUEUEs in host secondary storage, accessed by a data manager utility. They also appear as members of SETs, translating into labelled common in host primary memory and the data memories of the respective PEPE Control Units.

To date the application of process construction methods directly to the PEPE-related segments of the object processes to be built is extremely limited. Because of this it is not clear to what extent such methods will aid implementation of parallel processes. Also there are capabilities now available, or soon to be available, that will allow the description of a process to be stated in "neutral" terms; neither specifically sequential nor parallel. Then, via directives to the process constructor the designer can alter the distribution of data and functions over the parallel-sequential architecture to determine the optimal assignment of functions.

References

[1] A.J. Evensen, and J.L. Troy, "Introduction to the Architecture of A 288-Element PEPE," 1973 Sagamore Computer Conference on Parallel Processing.

[2] J.R. Dingeldine, H.G. Martin, and W.M. Patterson, "Support & Operating System Software for PEPE," 1973 Sagamore Computer Conference on Parallel Processing.

This work was supported by the U.S. Army Advanced Ballistic Missile Defense Agency (ABMDA), Huntsville, Ala., under Contract DAHC60-73-C-0060.

A COMPARISON OF A PARALLEL AND SERIAL IMPLEMENTATION OF A LARGE REAL TIME PROBLEM

Peter T. Alexander
Richard O. Parker
Science & Technology Division
General Research Corporation
Santa Barbara, California 93105

Abstract -- The mapping of an existing large real-time application onto PEPE is discussed. A set of measures describing the utilization of hardware by software and the match of the combination of hardware and software to the problem are suggested. The tools used to gather data for both the serial and the parallel implementation are described. Preliminary data is presented.

Introduction

SETS[1] is a computer program currently implemented on a Control Data Corporation 7600 consisting of approximately 150 modules and 120,000 machine instructions. It is designed to execute in real time and to model the complete environment external to a tactical data processor in a ballistic missile defense scenario. The primary task of SETS is to generate realistic radar returns in response to radar commands which are communicated through an interface with another computer. The salient characteristics of this problem are: (1) there is a large, time-varying data base describing the changing environment which must be input and maintained; (2) as many as 5000 radar commands per second may cross the interface; (3) the amount of processing required to generate a return is highly dependent upon the changing environment; and (4) average response times as short as 200 microseconds are required. This is a large real time problem characterized by high data rates, a dynamically changing data base, unpredictable computational requirements, and short response times. Some of these characteristics are common to other real time problems (e.g., Air Traffic Control and command and control systems). This particular application can be considered, in general terms, as a problem which requires that a dynamic data base be maintained and that the data base be accessed to respond, in real time, to questions related to that data base.

In an effort to extend the capacity and fidelity of the simulation, parts of the simulation are being implemented on PEPE, which will serve as an adjunct to the CDC 7600. The resulting configuration will be one in which PEPE and the CDC 7600 cooperate to respond to an inquiry.

The following two sections of the paper will summarize the current serial implementation and describe the mapping of the problem onto PEPE. The final sections will present some preliminary thoughts and data describing the performance of each implementation.

The reader's familiarity with the PEPE architecture and nomenclature is assumed. The preceding papers in this session should provide sufficient background and references. A knowledge of the CDC 7600 architecture is needed to understand some of the measurement data.

The Serial Implementation

Serial implementation of this problem have been designed, executed, measured and refined for three years [2,3]. In the current version, a single inquiry is processed to completion before starting work on the next. Although off-line preprocessing is used to increase throughput, no attempt is made to anticipate an inquiry. The relevant parts of the data base are updated as part of the inquiry processing.

Data Structures

The basic data structure used is the linked list. This was chosen because of the varying storage requirements of different scenarios and the logical interconnections of the data.

Control Structures

The process is data driven by the presence of new inquiries. Computations are initiated under the control of a time ordered task list. Input/output interrupts are transparent to the applications code.

Input/Output Structures

The data base is double buffered into Large Core Memory (LCM) at approximately one second intervals. The data is then transferred into Small Core Memory (SCM) as needed by the applications program. References to data are made through a Dynamic Storage Allocation system (DSA) [4] providing memory management that is transparent to the user. Inquiries and responses reside in circular buffers. The management of these buffers and the interface between computers is performed by special purpose interrupt handlers and Peripheral Processor Unit (PPU) routines transparent to the applications code.

This research was supported by the Advanced Ballistic Missile Defense Agency under contract DAHC60-73-C-0037.

The Parallel Implementation

The parallel implementation consists basically of two tasks: (1) responding to inquiries and (2) updating the data base. The division is in contrast to the serial implementation where the relevant portions of the data base are updated in response to each inquiry. These tasks have several subtasks. The PEPE implementation requires that the subtasks be assigned to the PEPE units so as to (1) maximize the number of simultaneously active instruction streams, (2) have access to an instruction set which is suited for the subtask, and (3) maximize the number of distinct data streams for parallel subtasks.

Description of the Major Subtask Distribution Among Units

Input of the data base is performed by the ACU/AU and the data is primarily stored in element memory. This choice was dictated by the need for coordination between data base input and data base updates.

Inquiries are input into a circular buffer in the CCU data memory (SCDATA) under the control of the CDC 7600 or a special interface computer. The CCU transfers this data from SCDATA into the appropriate Element Memories and handles the associated bookkeeping. This assignment was based upon the estimated utilization of the units which indicated that the CCU would be underutilized and thus available for this essentially serial process.

Decoding of the inquiries, selecting relevant parts of the data base and linear data base updates are performed by the ACU/AU. The ACU/AU is assigned to this task by an interrupt from the CCU/CU. These functions would probably have been assigned to the CCU/CU if fast shift instructions and a fixed point multiply were available.

The generation of responses and output functions are assigned to the AOCU/AOU. The generation function is primarily fixed point arithmetic and associative computations. Outputting of the data requires serially moving data from element memory to the AOCU data memory (SODATA) and its subsequent transfer to the CDC 7600. The high speed AOCU data and program memories appear well suited to these functions. The lack of a fixed point multiply might become significant in the future, if the generation function becomes more complex.

Data base maintenance is a parallel numerical task which is a background ACU/AU operation. This task keeps the data base current enough to allow rapid generation of responses.

Data Structures

The data structures are fixed arrays and circular buffers. The associative properties of PEPE and the natural partitioning derived by assigning data to different elements within the ensemble obviates the need for a software equivalent of linked lists.

Control Structures

The CCU is data driven by the presence of inquiries in SCDATA. Their presence is detected by the application software. The AOCU is also data driven by the presence of decoded inquiries in element memory. The ACU participates in the generation of a response to an inquiry when it is interrupted by the CCU. In addition, the ACU periodically inputs data base information and updates this information as a background process.

Comparison of the Serial and Parallel Implementation

Introduction

If both machines were executing functionally identical software, it would be meaningful to compare the execution times for representative sets of input data. This is not possible at the present time. Also, the approach yields little insight about the relationship of the hardware, the software, and the problem. We propose to discuss some preliminary attempts to determine, for each of the two machines, how well the software is matched to the hardware, and how well the combination of hardware and software is matched to the problem.

In the following paragraphs we will describe these comparisons. We will then outline the tools which are available to gather the data. Finally, we will present the preliminary data that is available.

Basis of Comparison

In principle, the applications software may be dichotomized into those pieces of code which are performing the computations (arithmetic and logical) specified by the functional description of the problem (call it problem code) and those pieces performing such functions as controlling the flow of the computational process, accessing data, and maintaining data structures (support code). Execution times will be influenced by both software design and the machine architecture that the software runs on. The ratio of problem code execution time to the total execution time (problem code plus support code) is a measure of the match between the hardware and software and the original problem. More generally stated, this is a measure of the resources required by a set of algorithms divided by the resources of the problem solution in which the algorithms are embedded.

Although we cannot give a precise quantification of this measure, some of the available data does provide a basis for an initial estimate. In the serial implementation, most of the data access, data transfer, and data structure maintenance functions are performed via FORTRAN subroutine calls to a Dynamic Storage Allocation system (DSA) and thus are identifiable. There are two limitations to using DSA execution time

as a measure of the support code time. First, many of the variables used in the arithmetic computations are singly or doubly indexed. The time necessary to perform the index arithmetic should be included in the data access time, but is not included in the measure of DSA. The second limitation is that some of the logic of the computations is simplified by the data structure. This time, which is included in the DSA total, should be charged to problem code. These two omissions bias the answer in opposite ways and thus, for a zero order approximation, can be ignored.

In our particular code--the skeletal PEPE SETS code--the problem code is executed in the ensemble and most of the support code is executed in the sequential control logic (SCL). The SCL code is controlling the process, is transferring data from Element Memory into the control unit data memories, and is performing address arithmetic. The parallel support code is primarily concerned with maintaining the Activity Stack and inputting inquiries.

A commonly used measure of the match of software to computer architecture is the amount of parallelism actually achieved, compared to the amount of parallelism inherent in the hardware. (For example, a system profile obtained with a hardware monitor is often used for this purpose.) We will consider this in the context of PEPE and the CDC 7600 central processing unit.

In the 7600 there is the potential for instruction fetch and execution overlap, and simultaneous execution of multiple instructions. The latter is accomplished with multiple functional units, most of which are segmented for pipelined operation. The maximum execution rate is one instruction each cycle.

Three types of parallelism must be considered for PEPE: the simultaneity of instruction streams, the presence of multiple, independent data streams, and the overlap of instruction fetch, routing, and execution. Many of the PEPE instructions execute in one or two clock periods. The degree to which instruction fetching, routing, and execution are overlapped can affect the execution rate by 100% or more.

Measurement tools have been developed and are being used to gather data describing the achieved parallelism on each of the computers.

Tools

The task of comparing the two implementations is just beginning. The data is sparse, and the conclusions tentative. There is a simulation of the 7600 [5] to evaluate the serial implementation. A software monitor package [6] exists to gather timing and execution path data for 7600 programs. As a check on the 7600 simulator there is timing and dynamic instruction mix data gathered with a hardware monitor [7] using a CDC 6400 executing a non-real time version of the application program.

The CDC 7600 simulator (SIM7600) is a program developed by General Research Corporation which simulates hardware functions of the CDC 7600 at a clock cycle level. The SIM7600 program is executed as an ordinary batch job under the control of the operating system on either CDC 6000 or 7000 series equipment. SIM7600 models the Central Processor Unit (CPU), the first level Peripheral Processors Unit (PPU), the Maintenance Control Unit (MCU), a variety of external equipment, and the connecting communication channels.

The CDC 7600 software monitor instruments object code to record the sequence of entries, exists and execution times of selected program modules. Reports are then generated describing the module characteristics and the relationships between modules.

The hardware monitor experiments investigated the characteristics of selected programs running on a CDC 6400 computer system. The goals of the study were to evaluate the use of hardware monitors for measuring the performance of real time computer systems, and to investigate the characteristic use of the CDC 6400 by the SETS program.

A PEPE computer simulation has been developed in order to aid the design and testing of program code, to provide insight into the operation and interaction of the various control and computational elements of the system, and to establish preliminary timing estimates for the algorithms which are being developed. The simulator represents current PEPE specifications [8] and contains all of the salient characteristics of the real hardware design.

The micro-code sequences were not modeled for each instruction algorithm. However, register contents, control signal values, execution delays and overall timing have been faithfully observed. Since the time-base of the simulation clock has the same granularity as the clock period in the PEPE system (100 ns), substantial data are available for collection and evaluation. The following data are currently being collected:

Clock cycle of instruction issue to each of the sequential and parallel units (relative to the beginning of simulation).

Element activity counts at each clock cycle.

Total of instruction issues for each unit (sequential and parallel).

Distribution of issued instructions by major (high order 5-bit) instruction category.

Count of references to element memory for each unit (AOU, AU, CU).

Count of the cycles of concurrent execution for parallel and sequential unit combinations (AOCU/AOU, ACU/AU, CCU/CU).

Accumulated cycles of concurrent idle time for sequential and parallel unit combinations (AOCU/AOU, ACU/AU, CCU/CU).

Calculated effective instruction rate.

In addition, a detailed trace facility has been incorporated to provide register contents, control signal status and element activity on a clock cycle basis. The trace facility may be enabled or disabled under object program control by utilizing one of the unused test and maintenance instructions.

In order to facilitate writing object code to be measured with the simulation tool described above, a cross assembler was developed to translate PEPE mnemonic instruction formats to appropriate bit-field definitions. The assembler permits data as well as instructions to be generated for any of the global (program or data) memories and, in addition, permits data to be preset into element memory. The cross assembler is implemented with the COMPASS assembly language of the CDC 6000-7000 computer systems. It allows the use of all of the standard features of the COMPASS assembler.

Serial Implementation Data

Using the software monitor we measured the elapsed central processor time, operating system services time, wall clock time, and the number of invocations for each module and major sequence of modules in the SETS code. Only the central processor time is considered for two limiting cases to derive the ratio of support code time to total execution time. The first case was one in which the computations required to generate a response to an inquiry were minimal, in the other case the number of computations were maximized. The support code data, presented in Figure 1, is the sum of the time spent in the data access and data structure maintenance routines (primarily DSA) and the time spent in routines which transfer data within the central processor memory systems. The high support code values for the minimum case is indicative of the amount of data handling activities performed in processing an inquiry independent of the complexity of the response.

Figure 2 presents some preliminary measurements which describe the parallelism achieved by the CDC 7600 CPU for a particular execution of the SETS program. The SETS program was in this case responding to a typical inquiry. This data was derived by running SETS with the CDC 7600 simulation.

The measures used are millions of instructions issued per second (MIPS), the fraction of cycles waiting to issue the next instruction, and the ratio of execution time to the execution time of an equivalent serial instruction stream computed by summing the execution time of all instructions issued. Instruction issue is delayed by contentions for registers, certain functional units, and the unavailability of an operand or instruction word being fetched from memory.

The CDC 7600 has a maximum instruction issue rate of one instruction every 27.5 nanoseconds. On this basis the SETS code issues instructions at approximately one-third the maximum rate. In the sample code, almost 15% of the total CPU time was spent initiating the fetch of instruction words from memory. In an additional 47% of the machine cycles no instruction issue occurred due to the delays mentioned in the preceding paragraph. Although it appears that much of this delay time is spent waiting for operands to be fetched from memory, more work is required to fully understand the mechanisms. It is expected that the CDC 7600 simulator will supply the data necessary to illuminate the causes of the delays.

Parallel Implementation Data

In the serial implementation the majority of the code is concerned with small pieces of logic arising from the need to consider a wide range of input scenarios. This diversity is lacking in the current PEPE code. We believe that these omissions cause the data base update to have more simultaneous data streams (active elements) and simpler data base update algorithms than would exist in the complete code. The complete code will probably require some additional associative operations to select the relevant portions of the data base for a given inquiry. In addition, the complete code will probably include a background process in the CCU/CU as part of the data base maintenance process. It is our guarded belief that the skeletal version does accurately represent the degree of interaction between the units.

The support code/problem code data for one case was derived from the instruction trace output of the PEPE simulator. The input for this example consisted of only one inquiry. The result is that the AOCU and CCU spend most of the time waiting for the arrival of data. In the full PEPE SETS there would be background tasks assigned to these units as well as a steady flow of inquiries. The effect of the limited inquiry data is to bias the execution towards a high percentage of support code. The support code consumed 44.8% of the total execution time.

Figure 3 presents a summary of the execution characteristics of a 100 microsecond time interval which included the complete processing of an inquiry. Polling for new inquiries and data base maintenance were background processes. The effective instruction rate is the sum of the average number of instructions issued each second to the three bodies of Sequential Control Logic (SCL) and the three Parallel Instruction Control Units (PICU). Thus any backlog remaining in the Parallel Instruction Queue (PIQ) at the termination of a run is not included in the total.

Regarding instruction issues as a measure of execution rate bypasses the problem of scaling

SERIAL IMPLEMENTATION
SUPPORT CODE TIMING
(Percent of Total CPU Execution)

	Minimum Response Computation	Maximum Response Computation
Data Acess and Structuring (DSA)	17.2	5.5
Data Transfer	17.5	7.1
	34.7	12.6

Figure 1.

PEPE EXECUTION SUMMARY

(Measurement = 1000 Clock Periods)

NUMBER OF INSTRUCTIONS ISSUED

AOCU	AOU	ACU	AU	CCU	CU
227	280	86	194	437	14

NUMBER OF REFERENCES TO ELEMENT MEMORY

AOU	AU	CU
64	132	6

EFFECTIVE INSTRUCTION RATE
(Issues x 10^6 per second)

12.38

Figure 3.

PRELIMINARY MEASUREMENTS
OF CDC 7600 CPU PARALLELISM

% Cycles Waiting to Issue	47.5%
% Cycles Used Initiating the Fetch of a New Instruction Word	14.8%
MIPS	13.7%
$\frac{\text{Elapsed Time}}{\text{Equivalent Serial Execution Time}}$	0.59%

Figure 2.

PEPE INSTRUCTION STREAM ANALYSIS

INDIVIDUAL UNIT ACTIVITY
(Percent of Total Time)

AOCU	AOU	ACU	AU	CCU	CU
38.1	49.5	9.5	65.6	50.3	2.9

OVERLAPPED UNIT ACTIVITY
(Percent of Total Time)

AOCU/AOU	ACU/AU	CCU/CU
5.3	5.8	0.7

OVERLAPPED UNIT IDLE PERIODS
(Percent of Total Time)

AOCU/AOU	ACU/AU	CCU/CU
17.7	30.7	47.5

Figure 4.

the parallel instruction execution rates by the number of active elements to derive a MIPS estimate. We chose this approach for two reasons. First, the utility and accuracy of the above MIPS estimate is questionable. Second, the number of active elements is often a function of the input data. We are attempting to consider the relationship between the hardware and software independent of the particular set of input data whenever possible.

The data in Figure 4 is part of the summary output from the PEPE simulation for the example considered in the preceding paragraph. The individual unit activity represents the percentage of time that each unit was executing instructions. (For the SCL the S_SCLF flags were monitored to determine activity for each cycle. For the PICU the S_IREQ flag was used.) The overlapped activity is the percentage of time that a sequential unit and its associated parallel unit were both active. The overlapped idle time is the percentage of time that a sequential and parallel unit were simultaneously not executing instructions. We present a preliminary interpretation of some of these values in order to give some information concerning the operation of PEPE and to show the types of analysis which can be performed using the simulation output. The interpretations are based upon the summary output and upon the cycle-by-cycle instruction trace, which is not shown.

Individual Activity

The low utilization of the ACU and the CUs is due to the few instructions in the skeletal code which execute there. The high utilization of the AUs is due in part to the relatively long execution times of many of the AU instructions. It is also the result of the lack of delay between successive AU instructions. This in turn is due to the PIQ which mitigates the effects of the relatively slow (300 ns) ACU program memory (SAPRGM).

Overlapped Activity

The low overlap of the AOCU/AOU activity, in spite of the many instructions executing there, is due to the short instruction executions times and the absence of a PIQ. However since the AOCU/AOU code sequences tend to be short and the presence of OTA instructions (output from an element A-register to the sequential A-register) frequent, a PIQ would probably provide little additional throughput. The relatively high overlap between the ACU and the AUs is due to the PIQ. The CCU/CU overlap is low because of the inactivity of the CUs and the factors considered for the AOCU/AOU.

Overlapped Idle

Since the CUs are idle most of the time, the CCU/CU idle time is primarily due to the non-overlap of instruction fetch with SCL instruction execution. This effect is less prominant in the AOCU/AOU due to the greater percentage of parallel instructions and thus increased overlap of fetch and execution.

Space limitations prevent the inclusion of a copy of the dynamic instruction trace or full execution trace. However the instruction trace is useful to determine the occurrence and duration of execution delays such as those caused by non-overlapped routing and execution, waiting for the PIQ to empty before OTA or branch instructions, or waiting for interrupts to be completed or accepted. (In the current code all of these delays tended to be of short duration, no more than 3 cycles. In test cases, however, delays in excess of 10 cycles have been observed.) Periods of inactivity for a particular unit and the effects of code segment rearrangement are easily seen on the trace. The inclusion in the trace of instruction issues to the PIQ, as well as to the PICU and SCL, provides insight into PIQ/SCL dynamics. The full execution trace has proved a valuable aid in debugging the PEPE code.

Summary

Continued work is planned in several areas. The PEPE SETS code will be expanded. At the same time the PEPE simulation will be enhanced by the addition of models for the interfaces between Input/Output Units (IOU) and external computers. We will continue to measure the characteristics of the PEPE code and develop methods for comparing them with those of the CDC 7600 implementation of SETS. The measurements will also be extended to consider the effects of input/output on the performance of both versions of SETS.

References

[1] T. O. Sullivan, P. T. Alexander, N. W. Hill, Jr., H. D. Wade, ABMDA SETS 1 Program (U), General Research Corporation CR-10-245, January 1973 (SECRET).

[2] K. E. Takacs, et al., The Simulation for the Analysis of Computer Systems (SACS) (three volumes), General Research Corporation, TM-1563, March 1972.

[3] R. L. Stone, et al., The Test Bed Demonstration Simulation System, General Research Corporation, TM-1301, July 1970.

[4] R. L. Stone, A Dynamic Storage Allocation System for FORTRAN Programs, General Research Corporation IMR-1249, January 1970.

[5] N. B. Brooks, SIM 7600 User Guide, General Research Corporation, July 1973.

[6] E. E. Balkovich, The Design and Application of a Software Monitor for the CDC 7600, General Research Corporation (in preparation) August 1973.

[7] E. E. Balkovich, P. T. Alexander, Hardware Monitor Measurements of SETS Performance, General Research Corporation (Working Notes) 1973.

[8] PEPE System Functional Design Specification, Vol. II, Hardware Specification, System Development Corporation, July 1973.

COMPUTER SIMULATION OF PEPE AND ITS HOST AT THE INSTRUCTION LEVEL

James L. Troy
Huntsville Operations
System Development Corporation
Huntsville, Alabama 35805

Summary

A serious problem emerges when an attempt is made to simulate parallel processing at the instruction level: execution time may be impractically slow. The faster a parallel processor executes, the slower will its instruction level simulator execute, and PEPE is very fast. The first attempt to simulate a 100-element PEPE at the instruction level (on an IBM 360/65, executing a BMD problem) produced a snail's-pace real-to-simulated time expansion of 11,000 to 1. This ratio was quickly reduced to a more practical 1000 to 1 by reducing the size of simulated element memory to fit the available core space. But because of the increased complexity and power of the current larger scale PEPE new solutions to the problem of excessive time expansion were sought.[1] Adding to the problem, however, were new requirements: the executions of all three control units were to be simulated "simultaneously" to accurately measure inter-unit memory access conflicts; element expandability (from 36 to 800 elements) was to be provided with particular emphasis on the efficient simulation of a 288-element PEPE; and instruction time was to be accurately modeled. A CDC 7600 was selected to execute this simulation since it is also being used as PEPE's "Host" to execute sequentially-oriented system tasks. Its relatively fast execution speed and large core storage are helpful in alleviating some of the problems inherent with sequential machines simulating PEPE; i.e., the use of such machines requires looping through many arrays which represent element data. The 7600, however, requires time-consuming data conversions between its 60-bit 1's complement and PEPE's 32-bit 2's complement formats.

The main approach taken to reduce execution time has been to eliminate code. There are, for instance, very few error checks. Erroneous conditions (such as, in PEPE, arithmetic operations with unnormalized floating point numbers) are not simulated where these conditions would surely lead to a program abort anyway. A preprocessing scheme eliminates several thousand word tables that would have otherwise been required in the online environment to provide instruction routing, execution times, legal field combinations and other information. This core space savings translates into considerable time saved due to Large Core Memory access-time characteristics of the CDC-7600. Dynamic instruction modification is disallowed by the software, so some instruction execution tasks can be preprocessed. Data, such as illegal instruction flag, execution time, address field size (which varies) and traps for parameter testing, are stored during a preprocessing pass into the 28 remaining bits of the 60-bit CDC 7600 word reserved in the load module for each 32-bit PEPE instruction.

FORTRAN was chosen as the programming language though code generation has been monitored closely to avoid inefficient object code. Extended FORTRAN for the CDC 7600 provides the necessary shift and mask statements to manage packed data. It was felt that the code generated by a modern compiler is efficient enough and the programming time thus saved is better spent interpreting the complexities of parallel hardware.

The PEPE simulator is instruction-driven and time is incremented following the occurrence of events which effect time. When PEPE simulation is interrupted due to I/O or interrupts between PEPE and external equipment, control is temporarily returned to a simulation control program (SDC's PEPSIE) which is event/time driven and in charge of coordinating I/O transactions between PEPE and the outside world.

The element expandability requirement produces a data variance of a million words, far too varied for one all-encompassing FORTRAN data block. So, three simulator versions are being produced in which up to 36, 300 or 800 elements can be modeled. If fewer elements are desired the space required for the maximum is blocked but not used. The 800-element version contains a disc-paging algorithm in which a block of contiguous addresses of element memory (for all elements) is maintained in core. This method was chosen based upon tests which showed that subsequent element memory accesses tend to stay in one "neighborhood" of memory for relatively long durations. The 36-element simulator is expected to reside in 7600 core at all times along with "Host" programs, the simulation controller, and executive programs. Disc transfer is expected to be required for the 300-element configuration only between the execution of major program segments.

Through the use of these techniques the PEPE instruction level simulator is expected to be a valuable tool in checking out the software utility package for the MSI Model PEPE currently being constructed and to validate the hardware design of the current model or of future large-scale integration PEPE models.

Reference

[1] A.J. Evensen, and J.L. Troy, "Introduction to the Architecture of A 288-Element PEPE," 1973 Sagamore Computer Conference on Parallel Processing.

This work was supported by the U.S. Army Advanced Ballistic Missile Defense Agency (ABMDA), Huntsville, Ala., under Contract DAHC60-73-C-0060.

AUTHOR INDEX

REVIEWERS

Prof. J. L. Baer	University of Washington
Dr. Kenneth E. Batcher	Goodyear Aerospace
Prof. H. C. Brearley	Iowa State University
Prof. P. Bruce Berra	Syracuse University
Mr. Wei-tih Cheng	Syracuse University
Mr. J. A. Cornell	System Development Corp.
Dr. H. R. Downs	Systems Control, Inc.
Dr. Philip H. Enslow, Jr.	Office of the Communication Policy
Prof. Michael J. Flynn	The Johns Hopkins University
Prof. Garth H. Foster	Syracuse University
Prof. Bernard A. Galler	University of Michigan
Dr. Mario Gonzalez, Jr.	Texas Instruments
Mr. Dale C. Gunderson	Honeywell, Inc.
Prof. Richard E. Horton	Iowa State University
Mr. Chao P. Hsieh	Syracuse University
Prof. M. Hu	Syracuse University
Prof. Keki B. Irani	University of Michigan
Prof. Robert M. Keller	Princeton University
Dr. Alan R. Klayton	Rome Air Development Center
Dr. Peter M. Kogge	IBM
Prof. David Kuck	University of Illinois
Prof. Duncan H. Lawrie	University of Illinois
Mr. Chung C. Lee	Syracuse University
Prof. Gerald J. Lipovski	University of Florida
Prof. John G. Marzolf	LeMoyne College
Mr. David McIntyre	University of Illinois
Prof. Shuhas Patil	Massachusetts Institute of Technology
Dr. William W. Patterson	Rome Air Development Center
Mr. James L. Previte	Rome Air Development Center
Prof. C. V. Ramamoorthy	University of California, Berkeley
Dr. Greg Schmitz	Honeywell, Inc.
Prof. Edward P. Stabler	Syracuse University
Prof. Harold S. Stone	Stanford University
Dr. Ken Thurber	Honeywell, Inc.
Mr. Ross A. Towle	University of Illinois
Prof. Roy J. Zingg	Iowa State University

COMMITTEES

Program Committee

Col. Philip H. Enslow	Office of the Communciation Policy
Prof. Tse-yun Feng	Syracuse University
Prof. Bernard A. Galler	University of Michigan
Dr. Peter M. Kogge	IBM
Prof. G. Jack Lipovski	University of Florida
Mr. James L. Previte	RADC
Prof. Harold S. Stone	Stanford University

Executive Committee

Prof. Tse-yun Feng	Syracuse University
Prof. Garth H. Foster	Syracuse Universtiy
Prof. Ming K. Hu	Syracuse University
Prof. Edward P. Stabler	Syracuse University

Local Arrangement

Miss Diane Sims	Syracuse University
Miss Anne Woods	Syracuse University