Technical Report 1248

Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines

Andrew Andai Chien

MIT Artificial Intelligence Laboratory

Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines

by
Andrew Andai Chien

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1990, in partial fulfillment of the
requirements for the degree of
Doctor of Science

Abstract

Fine-grained parallel machines exhibit great potential for high speed computation. Several machines such as the J-machine [42] and the Mosaic C [13] are projected to provide peak performance of 100's of billions of instructions per second in an air-cooled computer that fits in a cubic meter. While the hardware technology to build fine-grained machines is available, significant challenges remain in developing software systems to harness their computational power.

To program massively-concurrent MIMD machines, programmers need tools for managing complexity. One important tool used in the sequential programming world is hierarchies of abstractions. Unfortunately, most concurrent object-oriented languages construct hierarchical abstractions from objects that serialize – serializing the abstractions. In machines with tens of thousands of processors, this unnecessary serialization can cause significant loss of concurrency. Tools for managing concurrency should not restrict or reduce program concurrency.

Concurrent Aggregates (CA) provides multiple-access data abstraction tools, *Aggregates*, for managing program complexity. These tools can be used to implement abstractions with virtually unlimited potential for concurrency. Such tools allow programmers to modularize programs without reducing concurrency. These aggregates can be composed hierarchies of abstractions, allowing the structuring of a program to be highly concurrent at all levels.

In this thesis I describe the design and motivation of the Concurrent Aggregates language. I have used this language to construct a number of application programs. I present this programming experience with Concurrent Aggregates. Multi-access data abstractions are found to be useful for structuring applications without restricting concurrency. The tools provided to build such abstractions in CA allow the expression of a number of different styles of concurrency, including both data and control parallelism. I also present experience with the implementation of Concurrent Aggregates. A detailed evaluation of the efficiency of Concurrent Aggregates and the potential for further improving that efficiency is presented.

Thesis Supervisor: William J. Dally
Title: Associate Professor, EECS

Acknowledgments

I am especially thankful to my thesis advisor, Bill Dally, whose can-do spirit and enthusiasm have made the CVA Group a fun place to work. His encouragement and support of my research over the past three years is greatly appreciated. Bill's careful reading and comments have greatly improved this thesis. Thanks also to my other thesis readers, Bill Weihl and Dave Gifford, who have given me good advice (on many topics), support, and critical feedback throughout my thesis project. My thesis advisors, Bill Dally, Arvind, and Jim Melcher, have each in their own way helped me to learn what research is all about.

I would also like to thank Linda Chao, Stuart Fiske, Waldemar Horwat, John Keen, Jerry Larivee, Rich Lethin, Mike Noakes, Peter Nuth, Paul Song, Brian Totty, Deborah Wallach, Scott Wills, and the other members of the CVA Group and Computer Architecture Group for making the sixth floor an exciting place to learn. Your die hard attitudes and fun-loving spirit have made working here great fun.

Thanks also to the many friends who have provided kindly distraction from the seemingly never ending work at hand. In particular, I thank Julia Bernard, Gino Maa, Ken Traub, Earl Waldin, the Chaos Nets (you know who you are), and the members of the CSC-A Volleyball team.

My family have been a source of endless support and encouragement. I am indebted to my father who at an early age filled me with the joy of exploration and discovery, and my mother who suffered through its occasionally destructive consequences. Thanks Mom, Emily, Tony and Steve for your love, concern, encouragement, and confidence in me.

I cannot begin to express my joy and appreciation for the love and encouragement a special friend, Ellen. Thanks for hanging in through all those late nights in the lab!

Most of all, I give thanks to God for giving me the ability, perseverance and opportunity to complete this work.

To my father and mother

X

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the early 1980's, much has been written on the impact of VLSI and the impending revolution of parallel computing. It is currently possible to purchase integrated circuits with 1-4 million devices, yet massively parallel computer systems (with a few notable exceptions [93, 78]) have not had widespread impact. The dream of exceeding the performance of a CRAY [82] with an army of microprocessors has not yet been realized.

The major impediment to widespread use of massively concurrent machines is the lack of general-purpose programming systems. The development of circuit technology and our ability to design complex chips have both progressed quite rapidly. In contrast, our ability to program these machines has progressed much more slowly. Thus, their impact has been limited by the difficulty in developing effective programming systems. By almost any measure, massively parallel MIMD machines remain difficult to program.

Programmers of concurrent machines need tools for managing complexity. One important tool that has been used in the sequential programming world is hierarchies of abstractions. Unfortunately, most concurrent object-oriented languages construct hierarchical abstractions from objects that serialize – serializing the abstractions. In machines with tens of thousands of processors, unnecessary serialization of this sort can cause significant loss of concurrency. Tools for managing concurrency should not restrict or reduce program concurrency.

Concurrent Aggregates (CA) provides multiple-access data abstraction tools, aggregates, for managing program complexity. A multiple-access data abstraction is an abstraction with a well-defined message interface that can also process many messages *simultaneously.* Aggregates can be used to implement abstractions with virtually unlimited potential for concurrency. Such tools allow programmers to modularize programs without reducing concurrency. These aggregates can be composed to form hierarchies of abstractions, allowing the structuring of a program to be highly concurrent at all levels. CA supports expression of both control and data parallelism in programs.

My thesis is that multiple-access data abstractions can be used to manage the complexity of programming massively concurrent ensembles of processors [42, 13]. I call these multiple-access data abstractions Concurrent Aggregates. I have designed a programming language, also called Concurrent Aggregates, to explore this thesis. I examine motivation for the language's design and experience with its implementation and use.

To evaluate the programming language, I wrote a number of significant application programs. I present the performance of these application programs on fine-grained message-passing machines. In addition, I describe our programming experience: how features were used and how useful they were. Finally, I consider the efficiency improvement possible through better compilation and minor language changes in Concurrent Aggregates.

1.1 Original Results

The following results are the major original contributions of this thesis:

- The design of a language that incorporates multi-access (non-serializing) abstraction tools as its focus. We demonstrate how these multi-access abstractions can be used to organize programs, to implement massively concurrent data abstractions and data-parallel collections.

- The design of a language that allows programmers to explicitly manage consistency and replication within the programming model. This capability is used to implement abstractions ranging from strongly consistent to loosely consistent to inconsistent with a single set of mechanisms.

- The integration of first class continuations and user-defined continuations in a concurrent programming language. First class continuations are shown to be useful in improving program efficiency. The utility of user continuations is demonstrated with synchronization abstractions that suggest an efficient way to compose concurrent programs.

- The utility of first class messages is demonstrated. They can be used to implement meta-programs. In addition, they can be used as partial applications – allowing efficient support of SIMD concurrency or data parallelism in a multiple control flow language.

- Development of a novel type of delegation that allows message interfaces to be constructed incrementally from a number of other message interfaces (abstractions). This form of delegation allows the same interface to be constructed from shorter delegation chains.

- Programming evaluation of Concurrent Aggregates showing how these novel features can be used in real application programs.

- Implementation evaluation of Concurrent Aggregates programs. These studies show that highly concurrent programs can be expressed. A number of interesting characteristics of these programs are presented.
- Analysis of the achievable effiency of a Concurrent Aggregates implementation. Careful analysis shows that our original implementation can be improved by 30% to 50%. At that level, the message traffic of Concurrent Aggregates programs appears comparable to other approaches.

1.2 Multi-access Data Abstractions

Parallel programming is important because it offers a means for solving larger and more complex problems. Several research groups are building multiprocessors with thousands of powerful processors. The aggregate computing potential of these machines will exceed 500 billion instructions per second [42, 13]. If parallel programs can take advantage of the massive hardware concurrency we can afford to build (and make work reliably), such programs will be able to solve more complex problems, including many that are economically or practically impossible to solve with existing computer systems.

We are concerned with utilizing the computing potential of large ensembles of processors. At least two major obstacles stand between us and that goal.[1] First, programming should be relatively easy – it must not be so difficult that programs take years to construct and debug. As in large sequential programs, it must be possible to use abstraction (and hierarchies of abstractions) to relegate details to the appropriate level in a program. In fact, the importance of abstraction techniques is perhaps greater in parallel systems as the presence of nondeterministic behavior can complicate debugging. Second, the language must allow us to express sufficient concurrency to utilize the machine. In an ensemble of $10,000 - 100,000$ processors, unnecessary serialization may dramatically reduce the achievable performance.[2]

Most concurrent object-oriented languages serialize hierarchical abstractions. This leaves programmers with the choice of reduced concurrency or working without useful levels of abstraction. Going without these levels of abstraction makes programs more difficult to write, understand, and debug.

Concurrent Aggregates allows programmers to build hierarchical abstractions without serialization as shown in Figure 1.1. The message rates shown are normalized to the maximum message reception rate of a single object. CA programmers can build hierarchical abstractions from aggregates. Each aggregate is multi-access and therefore can receive many messages simultaneously. By using appropriately sized aggregates in the upper levels of the hierarchy, we can increase the message rate for lower levels in the hierarchy.

[1] There are several others such as load balancing and concurrent I/O but we will not discuss them here.

[2] A simple argument based on Amdahl's law [5] confirms this.

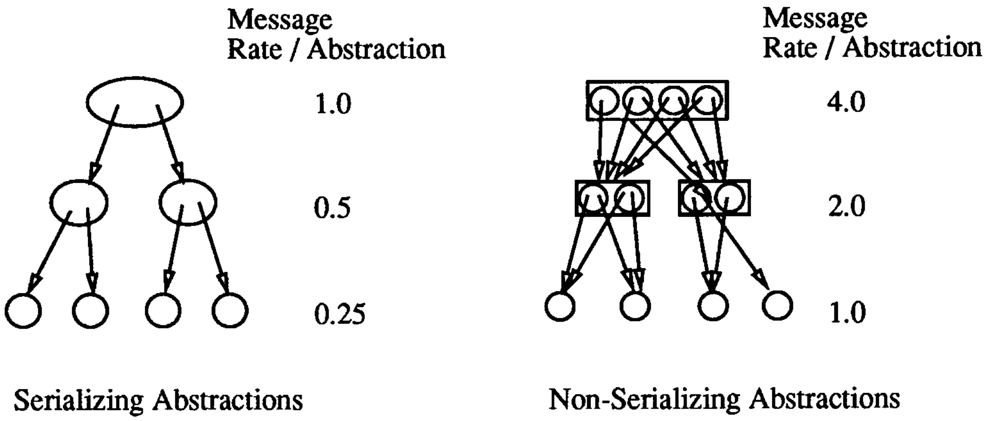


Figure 1.1: Maximum Message Rates in Abstraction Hierarchies

Concurrent Aggregates incorporates four additional important concepts that help programmers to construct abstractions from collections of objects and aggregates.

- Intra-aggregate Addressing
- Delegation
- First Class Messages
- First Class and User-defined Continuations

Intra-aggregate addressing allows representatives of an aggregate to compute the names of other parts of the aggregate. This facilitates cooperation in implementing abstractions by allowing representatives to pass messages to each other conveniently. For efficient communication, internal aggregate names can be used to link representatives directly to other objects in the same or in other aggregates. Externally, the aggregate can be manipulated with a single name – in a manner identical to an ordinary single object. This enables aggregates and single objects to be used interchangeably.

Delegation can be used to piece together (structurally compose) one aggregate's behavior from the behavior of others. In CA, an aggregate can delegate the handling of one or more messages to another aggregate. An aggregate can also delegate the handling of all messages not handled locally to another aggregate – implementing a more traditional form of delegation [69].

First class messages allow programmers to write message manipulation abstractions. Such abstractions can be used to implement control structures that perform message reordering or implement data parallel[3] operations on aggregates. Such aggregate operations are an important source of concurrency in our programs.

CA treats continuations [81] as first class objects. This enables programs to code synchronizing abstractions such as futures [61]. Further, ordinary objects or aggregates can be used as continuations (assuming they handle the reply message) making it possible for programmers to construct complex continuation structures such as a barrier. These user-defined synchronization structures can be cleanly factored from the remainder of the program. These four features – intra-aggregate addressing, delegation, first class messages and first class continuations – aid a programmer in constructing complex, concurrent aggregate behaviors.

1.3 Thesis Overview

In this thesis, we describe a language for programming fine-grained message-passing machines. We first describe the unique programming challenges presented by fine-grained machines in Chapter 2. These challenges are described in the context of a particular machine project, the J-machine. Some care is taken to describe the hardware support for programming systems in this machine. In addition, we summarize the related work in programming languages and systems. In Chapter 3, we motivate and describe the design of the Concurrent Aggregates language. Particular attention is paid to the novel features in CA, including delegation, first class continuations, user continuations, and first class messages. Chapter 4 documents our programming experience with Concurrent Aggregates. The structure of our application programs and particularly the use of aggregates and other novel CA language features is described in detail. The dynamic behavior of the programs is examined via a message-passing machine simulator. In Chapter 5, we examine implementation issues in the Concurrent Aggregates language. The cost of novel features such as first class messages and first class continuations is shown to be quite low. Two schemes for supporting aggregate name translation are described. Three types of compiler optimizations and their performance impact are studied. The potential performance improvement is measured.

[3] This differs from the usage of the term "Data Parallel" in the context of SIMD machines. We mean the parallelism arising from operations on large sets of data, be it heterogenous or homogeneous. No global synchronization is implied.

Chapter 2

Background

In this Chapter, we describe the context in which this research was performed. First, we consider the class of machines for which Concurrent Aggregates was designed. These machines have very different cost structures than typical von Neumann sequential computers. Second, we discuss the lingual ancestry of Concurrent Aggregates. The operational basis of the language and closely-related languages are described. Finally, we briefly discuss some efforts to program massively parallel machines, based on very different programming approaches.

2.1 Fine-Grained Concurrent Computers

Fine-grained parallel machines exhibit great potential for high speed computation. Several such machines [42, 13] are projected to provide peak performance of 100's of billions of instructions per second in an air-cooled computer that fits in a cubic meter. While the hardware technology to build fine-grained machines is available, significant challenges remain in developing software systems to harness their computational power.

Fine-grained concurrent computers differ from sequential computers and most other concurrent computers in several important ways. First, the tremendous computing potential of fine-grained machines comes from massive concurrency – thousands of processors operating concurrently. A high level of concurrency such as 10^5 is only possible with very small computing nodes. The fine-grained computers we consider differ from existing fine-grained SIMD[1] machines [55, 16] in their MIMD control structure. Second, these machines have very fast networks that can transmit a message from one corner of the machine to the farthest corner in a few microseconds [43]($\approx 10 - 20$ instruction times). Third, fine-grained concurrent computers can support very small tasks (≈ 20 instructions) efficiently. This allows us to break a computation into very small pieces – exposing greater concurrency for

[1] Single instruction, multiple data.

exploitation. Fourth, fine-grained machines have radically less memory capacity per unit of computation power than sequential machines or contemporary small scale multiprocessors. In such machines, a typical rule of thumb for a computer designers is "One megabyte of memory per MIP of computing power." In contrast, first generation fine-grained concurrent computers such as the J-machine [42, 38] and Mosaic C [86, 13] will have approximately one kiloword of memory per MIP of computing power[2].

Fine-grained machines have very different resource costs than more familiar sequential or small scale parallel machines. Computation cycles are very cheap as we have thousands of powerful processing units instead just a few. Memory seems relatively expensive as its ratio to computing power has changed dramatically. In fact, memory remains the primary cost in these systems. Effective use of the communication bandwidth in these machines is the key issue. As the scale of fine-grained machines is increased to $10^5 - 10^6$ nodes, how efficiently we can use the network will determine whether we continue to get performance improvements.

We summarize the significant distinguishing features of Fine-grained Concurrent Computers below:

- Massive Concurrency ($10^4 - 10^6$ nodes)
- Low-latency Networks ($2 - 3\mu$ seconds or $10 - 20$ instruction times corner to corner)
- Support Small Tasks ($10 - 100$ instructions)
- Radically Lower Memory capacity to compute power ratio ($\approx$ 1 Kword per MIP)

The dramatic performance potential and unique characteristics of fine-grained machines demand the development of new software systems. Existing operating systems developed for sequential computers or small-scale multiprocessors are inappropriate because they do not support extremely fine-grained tasks. Furthermore, such operating systems typically require far more memory per processor than is available in fine-grain machines.

2.2 Concurrent Object-Oriented Programming

2.2.1 Why Object-Oriented Programming?

Object-oriented models [36, 49, 72] provide an attractive base for building a concurrent programming system. Aggregation of procedure and data in such systems allow us to associate them – providing natural locality of reference. The grouping of data into semantic units allows us to make placement and migration decisions (for objects) based on their

[2] It is interesting to note that even the CM-2 [93] has $\approx$ 128 kilowords per MIP. This is based on 16μseconds for a 32-bit operation [80] and 256 kilobits of memory per node.

types. Object-oriented programming has been proven as an efficient and natural way to write large programs.

Object-oriented approaches are especially attractive for fine-grained MIMD machines as they encourage programs with many small pieces. The grain size of such programs is typically individual methods. Object-oriented programming models also lend themselves well to the introduction of concurrency – the addition of non-blocking sends fits very cleanly into the object-oriented paradigm.

2.2.2 Language Background

The Actor model [2] and Concurrent Smalltalk (CST) [57, 38] have had significant impact on the design of Concurrent Aggregates. The Actor model, developed by Agha, Hewitt and Clinger [33], is a clean semantic model for the execution of concurrent object-oriented programs. It has become the basis for a large number of concurrent object-oriented programming languages [12, 74, 105, 48, 106]. The basic execution model of Concurrent Aggregates is based on the Actor model as described in Section 3.2.1. However, Concurrent Aggregates augments the model with aggregates and a number of other features.

In other Actor languages such as ACORE [74], ABCL/1 [105], CANTOR [12, 20] and POOL-T [6], hierarchical abstractions (abstractions built from other abstractions) are built from single objects. Their objects may accept only one message at a time – resulting in serialized abstractions.[3] Multiple levels of abstraction can result in greatly diminished concurrency – even if each level causes only a tiny amount of serialization. Concurrent Aggregates solves this problem by incorporating some features from CST.

The direct predecessor of Concurrent Aggregates is Concurrent Smalltalk. Concurrent Smalltalk [37, 57] is an object-oriented language based on Smalltalk-80. CST is similar to ST-80 in that it supports dynamic typing and single inheritance. CST differs in the addition of asynchronous message sends (analogous to task creation) and distributed objects (allowing an object's state to be distributed). The addition of asynchronous sends permits concurrent operations. CST even allows concurrent operations on a single object – programmers must make appropriate use of locks to assure correct object behavior.

The design of Concurrent Aggregates incorporates some good ideas from Actor languages and CST. On one hand, the object model of CA is based on the actor model – each object processes only one message at a time. Each object has one lock that is implicitly acquired and released. On the other hand, the basic aggregate mechanisms: *one-to-one-of-many* translation and intra-aggregate name computation are clearly derived from similar features in CST. Concurrent Aggregates integrates this package by making aggregates – multi-access data abstractions – the focus of the language. Support is provided for programming with

[3] For example, in the Actor model, the serial message reception order is the actor's lifeline. This assumption is reflected in the notion that an actor's behavior can change at the reception of each message. In fact, in any language that assumes that an object resides on only one node, that processing node becomes a serialization point for its objects.

aggregates that makes it possible to conveniently build programs not easily expressible in either ancestor.

2.3 Other Related Programming Work

There are many concurrent programming models under investigation, but most of these models are not well-matched to fine-grained message-passing concurrent computers. To date, most sequential and parallel languages have required a large amount of state to be associated with a task. This typically implies large tasks and high task overhead. This "process model" has been moderately successful for small scale parallel and distributed systems, but the attendant overheads make it inappropriate for fine-grain machines. Other languages focusing on fine-grained parallel approaches such as dataflow [9], systolic arrays [64] and SIMD [53] have had some success. However, the systems developed to date depend on specific architectural features for their efficiency. These models require data-driven dispatch for each instruction and I-structures, very efficient nearest neighbor FIFO communication, and cheap global synchronization and control respectively. While any of these techniques could be used to generate code for fine-grained message-passing machines, we would not expect fine-grained message-passing machines to support them at the same level of efficiency as the respective custom architectures.

2.3.1 Functional Programming

Functional and near functional approaches to parallel programming have been pursued by many researchers [8, 60, 104]. These approaches often offer the advantage of guaranteed determinacy of execution. The disadvantage of a functional approach is that is difficult to describe a computation that depends on updates to shared state. It is similarly difficult to express non-deterministic computations. However, the main reason we have not pursued a functional approach is that such schemes are typically based on a uniform storage access model. The languages provide little support for expressing locality or data clustering. We feel that expression of such locality is essential to the success of fine-grained message-passing machines.

2.3.2 Systolic Arrays

Systolic arrays have been extensively developed and studied by Kung [63, 64] and others. While systolic computations are an efficient way to describe a number of applications, many others demand dynamic communication structures. In fact, researchers working on systolic arrays have moved to more and more general purpose computing structures [21] to broaden their application area. Systolic programming approaches suffer from the global knowledge problem. In a systolic cell, a programmer must know about all data that passes through his cell. This makes it difficult to build modular programs. In fact as a consequence, getting an entire application to run on a systolic array is difficult. They are often used as back-end processors for acceleration of particular algorithms.

2.3.3 SIMD Programming Languages

A number of different SIMD[4] languages such as C* [92] and Connection Machine Lisp [53] have been developed for the Connection Machine [93]. These languages have a single control stream, but allow parallel operations on arrays of objects. Programming in this style has been termed "data parallel" [56]. Each parallel operation completes before the next operation begins. This model is restrictive because is does not permit the expression of heterogeneous concurrency (two computations doing different things). Not surprisingly, this is a reflection of the fact that SIMD machines [93, 75] are not designed to exploit this type of concurrency. Fine-grained message-passing machines can exploit heterogeneous concurrency efficiently. Concurrent Aggregates is designed to support the expression of heterogeneous concurrency and relatively coarse-grained (10's of instructions) data parallelism.

2.3.4 CANTOR: a Programming Language for the Mosaic C

The CANTOR language [12] was designed for programming the Mosaic C [86], a fine-grained concurrent computer. CANTOR is a statically typed message-passing language based on the primitive actor model. The functionality provided in the initial design of the language was close to primitive actors. In a later version, they have added functions and vectors [20]. Functions provide functionality much like a procedure call – a call and return. Vectors provide a means for constructing indexable objects. The work on CANTOR has focused much more on low level efficiency and the expression of individual algorithms. The Concurrent Aggregates project has focused on facilitating the construction of large programs and entire applications in the language. In particular, CANTOR provides no support for multiple-access data abstractions or first class continuations. In addition, CANTOR provides no support for synchronization within a method, making it difficult to express concurrency within a method. Concurrent Aggregates uses *context futures* (a local future [61]) to support synchronization within a method.

[4] Single instruction, multiple data.

Chapter 3

Concurrent Aggregates

Programming languages for massively parallel concurrent computers need multi-access data abstractions. As defined in Chapter 1, a multiple-access data abstraction is an abstraction with a well-defined message interface that can also process many messages simultaneously. These abstractions have virtually unlimited potential for concurrency.

Concurrent Aggregates allows programmers to build hierarchical abstractions without serialization by providing a multi-access abstraction tool, *aggregates*. An aggregate is a homogeneous collection of representative objects that cooperate to implement an abstraction. Each aggregate has a group name that is used as the abstraction's name. Messages sent to the collection are directed to an arbitrary member of the collection by a *one-to-one-of-many* translation.

CA programmers can use aggregates to build hierarchies of abstractions. Each aggregate is multi-access and therefore can receive many messages simultaneously. By using appropriately sized aggregates in the upper levels of hierarchy, we can increase the message rate for lower levels in the hierarchy – allowing greater concurrency in the hierarchy. Not only does Concurrent Aggregates include multi-access abstraction tools, these tools are integrated into the ordinary data abstraction framework. Externally, aggregate-based and object-based abstractions are used interchangeably.

Concurrent Aggregates exploits concurrency at the level of *message-passing* operations. Each method invoked on an object is a message-passing operation. Every invocation gives rise to a potentially concurrent task. Each message passing operation is analogous to calling a function in a procedure-oriented language, except that it need not require a reply. As in many other object-oriented languages, method invocation in CA involves a type-dependent dispatch using the name of the message, its selector, and the type of the object receiving the message.

We are interested in computers that are capable of exploiting large amounts of fine-grain concurrency. These machines are tuned to support message-passing efficiently, so Concurrent Aggregates takes advantage of this by using message-passing pervasively. All communication and synchronization and hence coordination between objects is performed via message-passing. At first this may seem expensive, but as we gain experience with these fine-grain computing systems, we will be able to optimize many of these message-passing operations out – reducing the communication requirements of the programs.

Concurrent Aggregates also includes specific support for programming with aggregates. This support includes intra-aggregate addressing, delegation, first class messages and first class continuations. The motivation and utility of these features is briefly described here.

Intra-aggregate addressing allows parts (representatives) of an aggregate to compute the names of other parts of the aggregate. This facilitates cooperation within an aggregate. Allowing representatives to pass messages to each other conveniently facilitates implementing a coherent abstraction interface with an aggregate. For efficient communication, internal aggregate names can be used to link representatives directly to other objects in the same or in other aggregates. Externally, the aggregate can be manipulated with a single name – in a manner identical to an ordinary single object. This enables aggregates and single objects to be used interchangeably.

Delegation can be used to piece together (structurally compose) one aggregate's behavior from the behavior of others. In CA, an aggregate can delegate the handling of one or more messages to another aggregate. For example, aggregate *A* might delegate the handling of messages *M1* and *M2* to aggregate *B* and handle the rest of the messages itself. This is done by specifically delegating *M1* and *M2*. This allows a message interface to be composed from several other interfaces without the many levels of indirection necessary in previous schemes (forwarding due to delegation [97, 69]). With another kind of declaration, an aggregate can delegate all messages not handled locally to another aggregate[1]. This feature allows programs to use a more traditional form of delegation.

First class messages allow programmers to write message manipulation abstractions. Such abstractions can be used to implement control structures that perform message reordering or implement data parallel operations on aggregates. Such aggregate operations are an important source of concurrency in our programs.

First class and User-defined continuations in CA allow programmers to manipulate continuations as first class objects. This enables programs to decouple the call and return structures – in some cases simplifying or improving the efficiency of programs. For example, first class continuations can be used to code synchronizing abstractions such as futures.

The Concurrent Aggregates language even allows user-constructed continuations. A programmer can substitute ordinary objects or aggregates as continuations (assuming they handle the reply message) making it possible for programmers to construct complex continuation structures such as a barrier. These structures can be cleanly factored from the

[1] in effect, delegating the rest of the messages.

remainder of the program. These four features – intra-aggregate addressing, delegation, first class messages and first class continuations – aid a programmer in constructing complex, concurrent aggregate behaviors. Examples of usage for these novel features can be found in Appendix A.

This chapter describes the Concurrent Aggregates language. First, we give a brief example program – to give the reader a better perspective to understand the language description. Second, we give a high-level description of the key elements of Concurrent Aggregates programs – objects, aggregates, messages and continuations. Understanding these elements is a prerequisite to understanding CA programs. We also describe the execution model, the Aggregate model and its relation to the Actor Model. Third, we present the CA language itself – describing its syntax and semantics. Finally, we discuss the key design issues in CA and how they were resolved.

3.1 A Brief Example

Figure 3.1 contains a CA program that shows a simple use of aggregates. The `counters` abstraction is used to collect a sum. Some number of `count` operations are performed on the `counters` abstraction. The value from each such operation is accumulated into a local running sum. After all of the `count` operations have completed, it is possible to request the accumulated sum using `read_sum`. This operation returns the sum and clears it.

Each representative is used as a bin in the counting process as shown in Figure 3.2. In the `count` handler we see that the program increments the local instance variable `value` by the `val` argument. `read_sum` computes the overall sum via a linear reduction on the local partial sums. This is done by starting at representative 0 and working our way up through each representative collecting the sum and resetting the partial sum as shown in the `internal_read_sum` handler. The `initial_message` method shown is used to test the program. To give the reader a better idea of how Concurrent Aggregates programs are constructed, we work through several examples in detail in Appendix A.

Using an aggregate to implement the counters abstraction allows a counting operation to be performed more rapidly – more `count` messages can be processed in a given amount of time. The amount of concurrency that the implementation of `counters` can support is parameterizable at the aggregate's creation. Of course to users of the abstraction, the concurrency need not be visible.

3.2 CA Language Elements

Concurrent Aggregates programs contain four key types of elements: objects, aggregates, messages, and continuations. A Concurrent Aggregates program consists of a set of data abstractions (implemented with objects and aggregates). These abstractions send

```
;;
;; An aggregate that accumulates a SUM
;;
(aggregate counters value :no_reader_writer
           (parameters nr_reps)
           (initial nr_reps
                    (forall index from 0 below groupsize
                            (set_value (sibling group index) 0))))

(handler counters count (val)
         (seq (set_value self (+ val (value self)))
              (reply val)))

(handler counters read_sum ()
         (forward (internal_read_sum (sibling group 0) 0)))

(handler counters internal_read_sum (sum)
         (let ((newsum (+ sum (value self)))
               (nextindex (+ myindex 1)))
           (seq (set_value self 0)
                (if (< nextindex (- groupsize 1))
                    (forward (internal_read_sum
                              (sibling group nextindex) newsum))
                  (reply newsum)))))
;;
;; A test program that accumulates a set of numbers
;; and then reads the sum
;;
(method osystem initial_message ()
        (let ((the_counters (new counters 4)))
          (seq (forall index from 0 below 100
                       (count the_counters index))
               (reply (read_sum the_counters)))))
```

Figure 3.1: A Simple Counters Aggregate

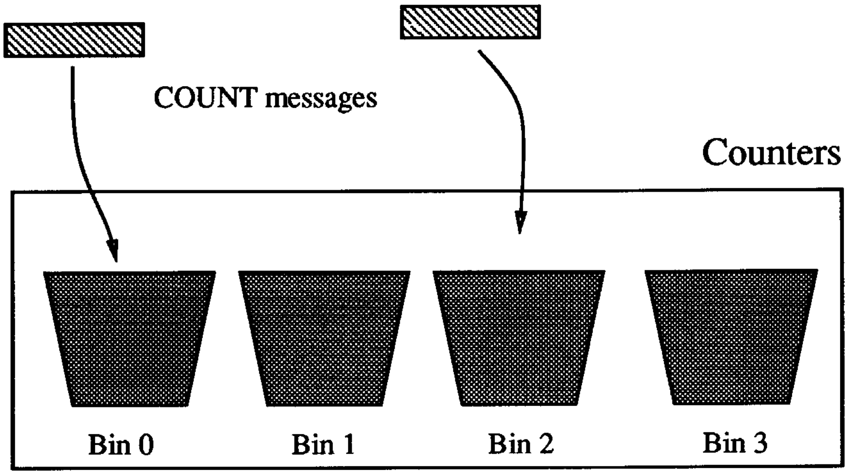


Figure 3.2: The Counters Aggregate: a set of counting bins

messages to each other; each message causes some local computation and perhaps some further message-passing. A computation in Concurrent Aggregates consists of a network of objects sending messages to each other. Figure 3.3 shows a network of objects and a number of messages in transit. As the computation progresses, the connections between objects change, new objects are created and the network of objects is transformed into the result of the computation. The course of a computation is governed by the execution model – the set of rules or state transitions that determine how the computation progresses. We informally describe the execution model for Concurrent Aggregates.

3.2.1 Execution Model

A computation consists of a set of objects, residing in a shared object namespace. These objects are distributed across the machine. They send messages to each other as shown in Figure 3.3. Each message starts a small piece of computation – potentially causing other messages to be sent. The model provides guaranteed delivery of messages: each message will eventually be received by its destination object. Message transmission order between objects is not preserved. The computation proceeds in two kinds of steps – objects executing messages and messages being delivered to objects. The Concurrent Aggregates execution model is based on the Actor Model [33, 2]. However, the rules must be changed slightly to support our notion of aggregates. We describe both models here.

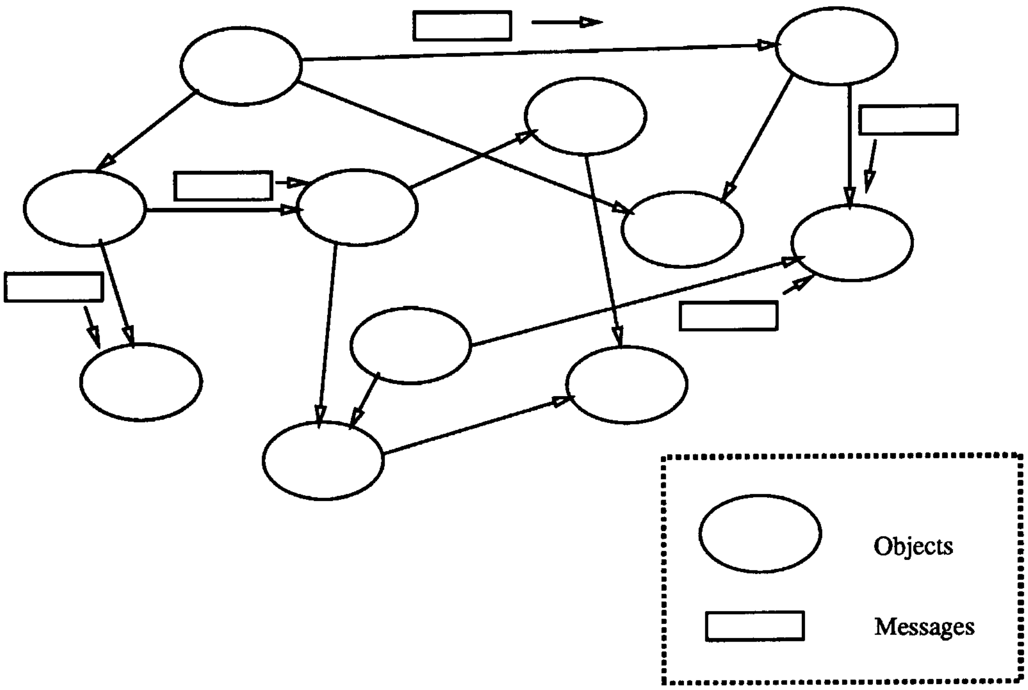


Figure 3.3: A Computation in Concurrent Aggregates

First, we define a few terms then we define the Actor and Aggregate models.

Local State The object storage used to hold a set of names of other objects. This set is a fixed size.

Known Objects A set of names of other objects, often called acquaintances. For a computation, this includes the local state as well as any objects' names in the invoking message.

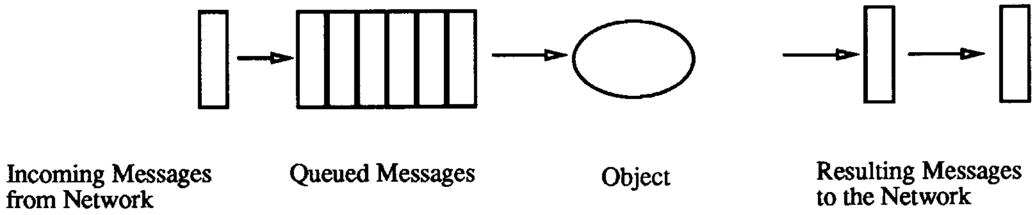


Figure 3.4: An Object Executing Messages

Basic Actor Model

Actor transitions: In response to a message, an actor can perform some number of the following actions.

1. Send messages to its acquaintances (other actors)
2. Specify a new behavior (includes local state modification)
3. Create new Actors

System transitions: Messages sent to actors are delivered eventually and in an indeterminate order. Messages that have arrived at an actor that is occupied are queued in its message queue. Conceptually, the queues are infinite. Each actor repeatedly accepts messages from its message queue, one at time. An implementation can covertly overlap message execution to improve performance. However, this overlap must not be visible to the programmer.

The local state that can be modified is in the receiver object. To modify state in objects, one must send messages. Any shared state in a computation is kept in shared objects and accessed only via message-passing. Execution of each method is exclusive – only one message is being processed by an object at any given time. These two facts simplify the interference that must be considered to assure correct program behavior. A programmer need not consider the interleaving of individual read and write operations. Because execution of messages is non-overlapping, the history of an object can be characterized by its *lifeline*, the sequence in which it executes messages.

Each object repeatedly accepts messages from its message queue and processes them as depicted in Figure 3.4. The system performs the complementary service – accepting messages from the objects and delivering them to their destinations. When there are no outstanding messages in the system the computation has completed.

These rules form the basis of a clean and powerful model. However, as we have described earlier, one can see that the sequential message reception of Actors force them to be serializing. Thus, in general, abstractions built from actors have some serialization[2]. We have proposed the addition of aggregates – collections of actors – as a solution to this serialization problem. The addition of aggregates requires only a few small changes to the Actor transition rules. We call this extended actor model, the Aggregate Model. The transition rules for the Aggregate Model are given below:

Aggregate Model

Aggregates are collections of actors. Each aggregate has a group name. Each actor in an aggregate has its own actor name. Group names and actor names are manipulated uniformly.

Actor transitions: In response to a message, an actor can perform some number of the following operations.

1. Send messages to its acquaintances (these may include individual actors and aggregates)
2. Specify a new behavior (includes local state modification)
3. Create new Actors

System transitions: These are quite similar to the basic actor model – messages sent to actors and aggregates are delivered eventually, and in an indeterminate order. Messages that have arrived at an actor that is occupied are queued in its message queue. Messages sent to aggregates are delivered to an arbitrary actor in the collection. In addition, a distinguished operation SELECT can be used to select actors out of a collection. Within an aggregate, this can be used to communicate with and coordinate members in the implementation of a coherent abstraction.

Our extension of the Actor model is well matched to the philosophy of simplicity and leaving the maximum freedom to the implementer. Our aggregation mechanism can be used to build collections of arbitrary kinds, with virtually unlimited concurrency (little serialization).

One way to see that we need to extend the basic actor model is to realize that our extension affects the basic addressing structure. Changing the addressing structure is not within the scope of the Actor model. Any emulation or interpretation of something so basic

[2] Of course actors that are immutable and a few other cases can be implemented without serialization. One way to implement an immutable actor with little serialization is to replicate it.

is likely to have significant storage and computational overhead. For example, we could use an immutable Actor, containing pointers to all the members, to implement a collection of size N. However, in order to reduce serialization, that actor would have to be replicated. To get serialization as low as with aggregates, it is necessary to have N copies – an $O(N)$ storage overhead!

3.2.2 Elements

Object An object is an entity that shares its behavior and local-state structure with other objects of the same class. Each message received by the object causes a piece of code to be executed. The response of the object to messages characterizes its behavior. The code executed in response to a message may modify the object's local state and send additional messages. An object's local state consists of a set of object names.

An object definition consists of three kinds of top level CA expressions: a `class` declaration, `method` declarations and `delegation` declarations. An object's local state is specified by its class declaration. The local state consists of a finite number of locations (instance variables), which can hold the names of other objects. The behavior of an object is defined by the methods and delegations declared for its class. Each such declaration specifies how to handle messages with a particular name, or selector. Method declarations specify a program fragment (the method body) to execute in response to a message. Delegations specify how to find the object responsible for handling a message. Method and delegation declarations should not both be made for the same selector. If conflicts exist amongst method definitions and delegations, the program is considered to be invalid.

Aggregate An aggregate is a collection of representatives that can be described by a single name. Each representative is an object. Aggregates allow the construction of highly concurrent data abstractions. The basic idea is that externally, an aggregate can be viewed as an ordinary object. Internally, an aggregate consists of many objects and can be used to implement unserializable behavior. Messages sent to the aggregate are directed to an arbitrary representative of the aggregate. The *one-to-one-of-many* translation is shown in Figure 3.5. Since this *one-to-one-of-many* direction is performed by the runtime system (itself multi-access), aggregates are multi-access and do not introduce serialization – each representative can receive messages concurrently.

Aggregate names can be manipulated identically with the names of ordinary objects. Users of an aggregate-based abstraction send messages to it and receive replies just as if it had been constructed with an object. An aggregate is defined by an `aggregate` declaration, `handler` declarations and `delegation` declarations. The `aggregate` declaration defines the structure of the local state for each representative. The behavior of each representative object is described with handler declarations (analogous to methods) and delegations.

Internally, an aggregate is a concurrent and distributed collection of objects working together. Distributing the collection makes it possible to support very high levels of con-

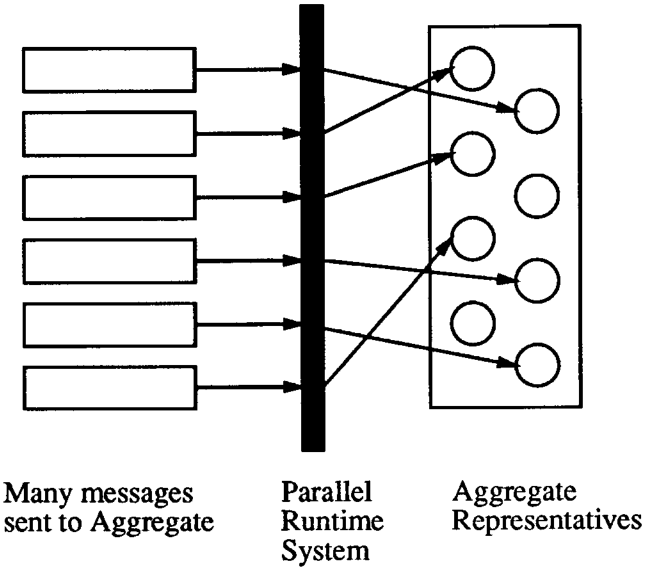


Figure 3.5: Aggregate to Sibling Translation

currency. Each representative can compute the name of the aggregate as well as the names of the other representatives. This ability allows the organization of aggregates to be quite flexible. Consequently, aggregates can be used to implement many different concurrent abstractions. Multiple representatives can be used to implement replication, partitioned state, or more complicated divisions of state and function. As the state distribution is visible to the programmer, he can control the degree of consistency and replication. Allowing the programmer to have such control can result in improved efficiency.

Message Messages form the active part of the computation. In order to perform computation, objects send messages to each other. When a message is received, the receiver object performs some computation in response. That computation may involve sending messages, receiving replies, modifying the local state and finally sending a reply message. Each message send is analogous to calling a function in a procedure-oriented language except that it need not require a reply.

Concurrent Aggregates treats selectors and messages as first class objects. By first class, we mean that CA programs can manipulate selectors and messages explicitly by storing, copying, and sending them. Several language forms in CA allow programmers to send, create, and modify messages. These facilities can be used to implement operations on collections, shared control structure abstractions, and message handling abstractions.

A message is quite similar to an unevaluated function application in a function-oriented language[3]. However, we have chosen to use messages because in many cases, the applications are simply used as holders for immediate parameters. For example, many operations on a collection of objects – data parallelism – can be implemented with first class messages. In cases where no complicated sharing is required, first class messages allow a programmer to factor message arity (number of arguments) out of concurrent program control structures such as fan-out or fan-in trees. By using first class messages, the programmer can control the efficiency of his program. Global compilation may not be required to achieve efficient execution.

Continuation In Concurrent Aggregates, there are two kinds of continuations – system constructed and user constructed. System continuations in Concurrent Aggregates are quite similar to continuations in sequential programming languages, except that they can only be used once. In CA, system generated continuations are references to a return location (a location in an activation record). Replies to such continuations cause the value of the reply to be stored in the return location. If the computation had suspended (i.e. was waiting) on this location, it is resumed. References to system continuations may be passed around just like ordinary objects. Unlike continuations in languages such as Scheme [81], a correct program may only reply to a system continuation once. A reply to a continuation causes it to be consumed – it no longer exists. More than one reply indicates an incorrect program. This design allows activation frames and continuations to be reclaimed cheaply by placing the onus for managing them properly on the programmer. The rationale for this design decision is presented in Section 3.4.6.

Users can construct continuations in Concurrent Aggregates. In a message-passing language, a continuation is nothing more than an object that accepts reply messages. This interface is quite clean because there is no shared state between the replier and receiver of the reply – all communication is done via the message-passing operation. Based on this observation, CA allows the programmer to construct continuations (from objects or aggregates) and use them throughout computations. Due to the clean interface, the construction of such continuations is quite simple. User continuations can be used to implement many different program control structures such as barriers, synchronization abstractions, and broadcast trees.

3.3 Language Syntax

3.3.1 Program Structure: Classes and Aggregates

A Concurrent Aggregates program is a series of top level forms. Each of these forms is described below. The programs are compiled and dynamically linked. The only ordering restrictions are: globals should be defined in the order in which their initial value expressions

[3] The evaluation of arguments is strict at the time of message creation. The computation for the application is actually performed after the message is transmitted and received.

should be computed, object class definitions must precede methods or delegations for that class, and aggregate definitions must precede handlers or delegations for that aggregate.

top-level-form ::= class-def | method-def | delegate-def |
aggregate-def | handler-def | global-def | exp

Class and Aggregate Declarations

class-def ::= (**class** class-name ivar-list parameter-list init-form)
aggregate-def ::= (**aggregate** agg-name ivar-list parameter-list agg-init-form)
class-name ::= ident
agg-name ::= ident
ivar-list ::= ident* | ident* **:no_reader_writer**
parameter-list ::= (**parameters** ident+) | {}
init-form ::= (**initial** exp*) | {}
agg-init-form ::= (**initial** size exp*)

The syntax for class and aggregate declarations is almost identical. The only difference is that the initial form for aggregates requires an additional term to specify the size (number of representatives) of the aggregate. The `ivar-list` specifies the local state of the object (representative). The `:no_reader_writer` switch indicates that reader and writer methods should not be automatically defined. If this switch is not present, `X` and `set_X` methods will be defined to access and modify the object state. The parameter list allows classes and aggregates to be parameterized. The `initial` form is executed when the object (or aggregate) is created. Upon its return, the descriptor is returned to the sender of the new message. The parameters are passed as arguments to the `initial` form and may be accessed as such in the `initial` form.

Method and Handler Declarations

method-def ::= (**method** class-name method-name (arg*) exp+)
handler-def ::= (**handler** agg-name handler-name (arg*) exp+)
class-name ::= ident
method-name ::= ident
arg ::= ident | **:no_exclusion**
agg-name ::= ident
handler-name ::= ident

Methods and handlers define the program to be executed when a message is received in objects and aggregates respectively. Handlers are very similar to methods, but CA uses a different reserved word to remind programmers they are dealing with an aggregate. For each message, the message name and receiver object's class determine which method is invoked. If a message with selector, name, **A** was received by an object of class **B**, the method with class-name **B** and method-name **A** would be invoked on the receiver. An analogous procedure occurs for aggregates and handlers. Object behavior is defined by the methods and delegations defined for that class. In similar fashion, aggregate behavior is defined by the handlers and delegations defined for that aggregate. Method declarations and message delegations should not conflict in object definitions. Likewise, handler declarations should not conflict in aggregate definitions. So long as there are no conflicts, the action taken in response to a message is uniquely defined.

Concurrency Control In many cases, it is necessary to control concurrency in order to assure proper program behavior. Object oriented languages provide a natural granularity for concurrency control – object methods. Concurrent Aggregates allows the programmer to control concurrency on a per object basis with locks. Programmers can specify whether each method is executed exclusively on an object.

Default Concurrency Control Normally methods have exclusive access to the state of the receiver object. Messages reaching a locked object are deferred. When the lock is relinquished (i.e. the message execution completes), the deferred messages will be processed.

No Concurrency Control With some methods, it is desirable to forgo any concurrency control for the receiver object. CA allows a method to be uncontrolled (i.e. no implicit lock - unlock forms around the method body). This means that the method may run arbitrarily interleaved with ANY of the other methods of the object. Uncontrolled methods are specified by using the **:no_exclusion** keyword in the argument list.

Delegation Declarations

delegate-def ::= (**delegate** cname message-name ivar-name)
cname ::= ident
message-name ::= ident | **:rest**
ivar-name ::= ident

Message delegations allow object behaviors to be composed from the behaviors of other objects. A message delegation causes messages of name `message-name` sent to objects of class `cname` to be sent to the object specified by `ivar-name`. In this way, a complicated behavior can be built up from the behavior of its parts. Delegations to the **:rest** message name cause all messages that are not explicitly handled or delegated to be delegated. Message delegations for aggregates work in an analogous manner.

When a message with selector `X` is received, the handlers and delegations defined for the aggregate determine how the message is handled as shown in Figure 3.6. If a method or handler is defined for `X`, that code is invoked. If a delegation is defined for `X`, the message is sent to the delegation target. Method definitions and delegations for a class should not conflict (i.e., there should not be a method definition and a delegation for the same selector). A conflict indicates an incorrect program. If neither method nor delegation is defined and a **:rest** delegation is defined for `X`, then the message is sent to the **:rest** delegation's target.

Concurrent Aggregates incorporates *per messaage* delegation which allows programmers to construct aggregate behavior incrementally. A message interface may be constructed

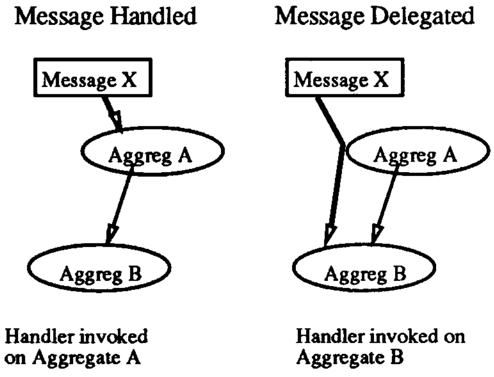


Figure 3.6: Message Handling and Delegation

from several other interfaces without many levels of indirection (forwarding due to delegation). Specific methods may be used to extend or modify the message interface of an aggregate while a **:rest** delegation specifies a default receiver object for all other messages. By enabling programmers to piece behaviors together, CA allows programmers to distribute message handling responsibility over a number of objects – potentially increasing concurrency. If delegations are static (the object handling the delegated message never changes), it is possible to compile out the level of indirection.

3.3.2 Basic Expressions: Method and Handler Bodies

Method and handler bodies can be divided into sequencing, message control, and binding constructs. Handler bodies may also contain sibling expressions – a way of computing the name of another representative within the same aggregate. Within a body, some message and object state can be conveniently accessed as pseudo-variables. Instance variables on the receiver object can be accessed and modified with `(X self)` and `(set_X self new_value)` forms in methods or handlers for that object. For example, the two expressions below would return the value of the instance variable `foo` and set the value of the variable `foo` to 2, respectively.

```
(foo self)
(set_foo self 2)
```

When used with the self pseudo-variable, these expressions do not require that reader and writer methods be defined.

None of the pseudo-variables may be modified. For the object methods and aggregate handlers, the pseudo-variables are given below.

Pseudo-Variables

Object:	self
Aggregate:	self, group, groupsize, myindex
Message:	msg, requester, <argname>

Self refers to the receiver of the message. In an aggregate representative, self refers to the representative receiving the message while group refers to the entire aggregate. Groupsize is the number of representatives in the aggregate, and myindex is the receiver's unique index in that aggregate. These indices are zero-origin. Msg refers to the received message. Requester refers to the current continuation, the default destination of replies. In addition, the explicit arguments of message can be referred to by name (we denoted this with <argname>).

Basic Expressions

exp ::= (message-name exp+) |
        (**concurrent** exp+) | (**conc** exp+) |
        (**sequential** exp+) | (**seq** exp+) |
        (**if** exp0 exp1) | (**if** exp0 exp1 exp2) |
        (**forall** loopvar **from** exp0 **below** exp1 exp*) |
        global-form
loopvar ::= ident
exp ::= (**let** (binding-pair+) exp+)
binding-pair ::= (variable-name exp)
variable-name ::= ident
exp ::= (**sibling** aggname exp0)
aggname ::= ident

(message-name exp+)

This is the basic message send operation. A message with selector message-name is sent to the value of the first expression with the result values of the following expressions as arguments. The expressions are executed as if they were within a concurrent construct, and the message send occurs when all expressions have returned values (i.e. message sending is strict). The value of the reply to the message is used as the value of this form.

The combination of sequencing constructs in a method body allows the programmer to constrain the execution order of message passing operations – control the concurrency. There are two constructs, `concurrent` and `sequential`, that can be used to sequence execution in CA.

(**concurrent** exp+) | (**conc** exp+)

In the `concurrent` construct, execution order is only constrained by local data dependencies. Subject to these constraints, any partial order is acceptable. The compiler and runtime system may choose any order that satisfies the data dependencies. However, a top to bottom and left to right order must be an acceptable order. For example, a total order such as sequential execution is acceptable. Non-local data dependencies, such as a cycle through several method invocations must be satisfied by the programmer. `Concurrent` constructs return a null value after all expressions in the `concurrent` form have returned. Method and handler bodies as well a the initial expressions in class and aggregate declarations have an implicit `concurrent` construct at the top level.

(**sequential** exp+) | (**seq** exp+)

The `sequential` construct enforces a linear order on the execution of expressions. They are executed in the order they occur in the program. This construct allows the programmer to serialize execution as necessary. The value returned by the last expression is returned by the `sequential` construct.

(**if** exp0 exp1)
(**if** exp0 exp1 exp2)

The `if` construct allows for conditional execution. `Exp0` is executed, and its return value determines which arm of the conditional is executed. For each `if` construct executed, `exp0` is guaranteed to be executed. If `exp0` returns a null value, `exp2` is executed. If the result of `exp0` is non-null, `exp1` is executed. The `if` construct returns the value of the arm which is taken. If a null arm is taken, a null value is returned.

(**forall** loopvar **from** exp0 **below** exp1 exp*)

The `forall` construct specifies repeated execution. This is convenient for dealing with a late-bound number of objects or size of an array. For example, this can be very convenient at the leaves of a divide and conquer algorithm. The loopvar is bound in the body of the `forall`. The `forall` is equivalent to a `concurrent` construct with the a variable number of clauses and the appropriate index substituted for loopvar in each clause.

exp ::= (**let** (binding-pair+) exp+)

Let bound variables are bound in the scope of the `let` and may shadow arguments and other `let` bound variables. Pseudo-Variables are reserved words and thus cannot be used as identifiers for let bound variables. Let bound variables cannot be modified.

exp ::= (**sibling** aggname exp0)

For many aggregates, coordination and synchronization amongst the representatives is important. This can be done in several ways: For instance, the control structure in message handlers can be used to provide such coordination. Message handlers can also synchronize via explicit access to shared state. Or, representatives can pass messages to each other to cooperate. We facilitate intra-aggregate message passing with the `sibling` expression.

aggname should be bound to the name of an aggregate. The **sibling** expression requires that exp0 return an integer i such that 0 <= i < groupsize for the aggregate specified by aggname. For appropriate i, the **sibling** expression returns the name of the representative with i as its index (myindex) value. For example, **(sibling group 0)** would return the name of the 0th representative in an aggregate. The representatives in an aggregate have indices from 0 to **groupsize**−1. **groupsize** is a pseudo-variable which contains the number of representatives in an aggregate. Sibling names are ordinary object names – allowing direct connection to aggregate representatives when performance is critical.

3.3.3 First Class Continuations: System and User

exp ::= (**reply** exp) |
        (**forward** exp) |
        (**do** exp) | (**do** exp0 exp1)

Concurrent Aggregates supports several different message control constructs to allow a programmer to control the synchronization between method invocations. The continuations link method invocations together into the overall computation. Each method invocation has a continuation. It can refer to that continuation with the pseudo-variable, **requester**. The **reply** expression sends a reply message with the value of exp to the current continuation – the continuation held in **requester**. By default, the continuation of a method invocation handles the reply message and resumes any computation that has suspended waiting for the reply value. CA allows the programmer to modify the continuation of an invocation with the do and **forward** expressions. Forward uses the current continuation for the continuation of the message being sent.

Do expressions allow the current method to execute an asynchronous or non-blocking send. Execution continues immediately as no reply is expected. Any reply to a message sent with do is discarded. The second do form allows the programmer to specify a continuation for exp0. This continuation is specified by the value of exp1.

Continuations in a message-passing language such as Concurrent Aggregates are simply objects that handle the reply message. Because we have distributed and encapsulated all activation frames, this abstraction is strictly observed. While the system may implement continuations with a context (method invocation frame), continuations might also be implemented with other objects. In fact, any object that handles the reply message can be used as a continuation. CA programmers can construct continuations and substitute them in programs using the do expression. This flexibility allows programs to decouple the forward linkage (calling structure) from the backward linkage (return structure), in some cases yielding simpler or more efficient programs. One good example of a simple user-constructed continuation is a barrier synchronization.

3.3.4 First Class Messages

Message Access and Modification

exp ::= (**message** exp) |
(**msg_at** msg-exp index-exp) |
(**msg_atput** msg-exp value-exp index-exp)
(**send** exp) | (**send** exp0 exp1)

Messages can be viewed as objects in CA. Their interface consists of the **msg_at**, **msg_atput**, and **send** constructs. However, they are treated specially because they are critical to system performance. Messages are by-value parameters [77] – they are copied when references to them are duplicated. This assures that all message operations are local and therefore can be performed efficiently.

(**message** exp)

The message construct creates a message and returns that message. The expression in a message construct must be a message send. A message corresponding to that send is created and returned. Arguments to that message are evaluated before the message value is returned. The continuation of the created message is null. Another way to get ahold of a message is to use the **msg** pseudo-variable to obtain a reference to the current message.

(**msg_at** message-exp index-exp)
(**msg_atput** message-exp value-exp index-exp)

Message state can also be modified as a message is sent by using the do and send constructs. Message state can also be accessed and modified with the msg_at and msg_atput expressions. Message state is mapped as follows: 0 = selector, 1 = continuation, 2 = receiver object. Indices greater than two access the user-visible message arguments in order. The msg_at expression returns the value at the specified index within the message. The msg_atput expression returns the value just inserted into the message.

(**send** exp) | (**send** exp0 exp1)

Send expressions allow the current method to send messages that it may have. Send returns immediately. The continuation in the sent message is not changed by the send expression. The second send form allows the programmer to substitute a receiver object in the message. This receiver object is specified by the value of exp1.

3.3.5 Globals

global-def ::= (**global** global-name initial-value)
global-form ::= (**global** global-name) | (**set_global** global-name value)
global-name ::= ident

Global variables are visible throughout a program. The global name is the name used to reference the variable. The initial-value is a CA expression that returns the initial value for the global. It may not be omitted – all globals must be initialized. Global variables are initialized sequentially in the order of their appearance in the program. This allows the initial values of globals to depend on each other. Globals may be read and written with global form expressions. Access to and modification of globals is atomic.

3.3.6 Primitive Classes

A number of classes are built into the CA language. These primitive classes are the building blocks for constructing higher level abstractions in the language. The user can augment the operations on these primitive types but should not attempt to override existing their method definitions. Such attempts will result in unpredictable program behavior.

Integer:	$+, *, /, -, >, <, =, ! =, >=, <=, \mathit{min}, \mathit{max}, \mathit{mod}, \mathit{and}, \mathit{or}, \mathit{not}$
Float:	$+, *, /, -, >, <, \mathit{min}, \mathit{max}$
All types:	$\mathit{eq}, \mathit{neq}$

Numbers need not be created explicitly with the new selector. They can be used as literal constants or created as the result of an arithmetic expression.

Primitive built in classes include selectors, messages, symbols and arrays.

Array: CA supports 1-D arrays. Multi-dimensional arrays must be constructed explicitly. The message interface for this type is given below:

(**new array** size) An array object with size elements will be created and returned.

(**at** an-array index) Returns the value of the array at the index'th location (indices are zero origin).

(**atput** an-array value index) Stores value in the index'th location of the array.

(**size** an-array) Returns the size of the array.

3.4 Design Issues in Concurrent Aggregates

In the design of Concurrent Aggregates, we have encountered many interesting design issues. Some are peculiar to the design of Concurrent Aggregates, others are of broader concern. We discuss some of the most interesting design issues and motivate our design choices.

3.4.1 Non-serializing Data Abstractions

The central idea in Concurrent Aggregates is to explore programming with non-serializing data abstraction tools. We have argued the need for non-serializing data abstractions in Section 1.2. Simply put, non-serializing data abstraction tools allow programmers to use arbitrary levels of data abstraction without constraining concurrency.

Aggregates provide a framework for programmers to explicitly control replication and consistency in building distributed abstractions. The framework provided by aggregates is significantly different from other such object replication (caching schemes), shared memory, multi-version memories, and state partitioning in distributed memory machines. We describe these other replication and partitioning schemes and consider how the aggregates framework relates to them.

Object replication means creating a number of copies of an object. The object replicas must be kept consistent: other than a performance increase, users of the object should not be able to tell that replication is being done. The number of object copies may be static as in a replicated file system or copies may be created dynamically. When an object is not mutated, object copies can increase the effective bandwidth of the object. However, when an object is mutated, all of the old copies must be invalidated, and only one master copy is mutated. Subsequently, the object can be replicated again. A variation of this scheme is to lock all copies and update all of them simultaneously. When they are unlocked, all of the copies are consistent. In this scheme, the copies of the object are not visible as copies to the user programs. They are all indistinguishable from the original object: they have the same name.

Coherent shared memory systems can be viewed as an object replication system where the copies are created dynamically. The objects in most shared memory systems are the units of transfer – the cache lines. The operations supported on these objects are READ and WRITE. Of course, mutation operations such as the WRITE operation require the destruction or update of all copies of the cache line.

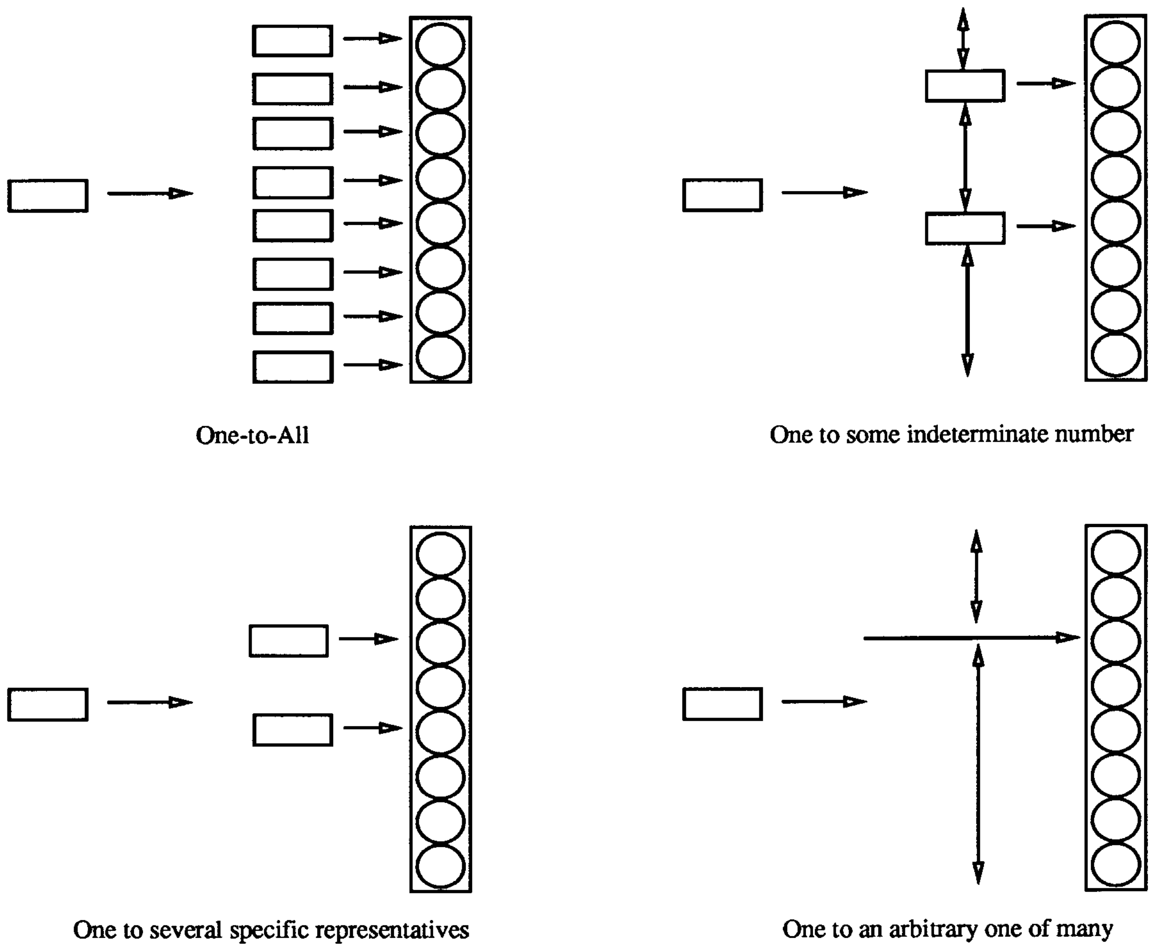
Multi-version memories as proposed by Weihl [101] allow for replication with transient inconsistency. Updates that occur are guaranteed to propagate eventually to all replicas. However, it is possible for different accesses to see different versions of the memory and thus get an inconsistent view of the memory. All versions of a memory have the same name, but upon access, a user of the memory accesses one of the versions. Thus the versions share the same name in the user namespace. Multi-version memories are promising in data structures such as B-tree's [35], where operations can be structured so that correct operation of the structure does not depend on having the most recent information. Using stale information will only result in slightly worse performance, not incorrect results. Extensive studies of the concurrent B-tree implementation using multi-version memories can be found in [100].

State partitioning or domain decomposition has been used in distributed computer systems as well as in programs for distributed memory computers [85, 87, 4, 78, 11, 93] for many years. State partitioning involves taking the state for a data structure or abstraction and spreading it over a number of machines. Each machine is then responsible for handling requests for that data. In such systems, the machine number (or node number) is used as a key for mapping the partitions. As the different machines do not share an address space, the name of such a partitioned collection data set is usually implicit and sharing must be done by convention. For example in single-program multiple data (SPMD) programs, each machine might keep a pointer to its part of the shared data structure in a variable of the same name. There is usually no language support for managing such sharing or partitioning in distributed memory machines.

Aggregates in CA unify and generalize the various techniques described here. Aggregates provide at the language level a means for explicitly controlling replication, partitioning and consistency. Aggregates can be used to implement partitioning – over a set of objects, not tied in any way to the number of physical machines. Aggregates can also be used to implement object replication schemes, though aggregates only support static replication due to the expense of interpreting the dynamic translation required for dynamic replication. To update an aggregate being used for static replication, all parts of the aggregate must be locked, and then the update must be propagated to all copies. Only then can the copies be unlocked. For non-mutating operations, no coordination is required amongst the aggregate representatives. Aggregates can also be used to implement multi-version memories. Of course multi-version memories are simply a less coherent version of replicated shared memory, so they are even easier to implement. We direct all mutation operations to a single representative, `M`, of the aggregate. Non-mutating operations can be performed on any representative. The representative `M` serializes the updates and propagates them to the representatives.

We chose to allow representatives to be mutable (and aggregates thereby inconsistent under programmer control) because it is quite often useful to have inconsistent state amongst the representatives. It can be used to support partitioned state, loose consistency, or other forms of cooperation. These can all reduce the cost of replication from that required for "consistent" storage.

One can view aggregates as an alternative to system-wide coherent shared-memory. Aggregates give more flexibility to the programmer. By allowing the programmer to manage the consistency, his program can be more efficient because he need only keep things as consistent as necessary, not consistent all the time. A programmer can implement the full spectrum from partitioned state (no consistency required) to static replication with full consistency.



One-to-All

One to some indeterminate number

One to several specific representatives

One to an arbitrary one of many

Figure 3.7: Different Aggregate Message Reception Schemes

3.4.2 Messages to only one representative

We chose to have messages sent to an aggregate to be delivered to only one representative object, and one chosen arbitrarily by the runtime system in an aggregate. Other options included delivery to all representatives, as many as convenient, a specific representative or several specific representatives. We eliminated the other choices for the following reasons:

All Representatives: This scheme is not scalable. For large aggregates, the effective message bandwidth would be no larger than for a single object – each representative must at least receive each message sent to the aggregate. Such waste may be acceptable in a single-instruction multiple data style of programming (with the Connection Machine, for example), because the underlying hardware structure prohibits the exploitation of heterogeneous concurrency, but in a multiple-instruction multiple data (MIMD) machine, it is not acceptable. Further, in the fine-grain concurrent computers we have considered, there is no hardware broadcast medium. Thus, such broadcast requires at least N messages for N representatives. These problems with broadcast conventions have also been noted in systems with underlying multi-cast, and many fewer processing nodes [15].

Some indeterminate number: Deliver copies of messages sent to the aggregate to as many representatives as is cheap or convenient. This kind of an imprecise specification, while necessary in some distributed systems built upon unreliable transmission, is inappropriate in computer systems where reliable delivery is assured. In distributed algorithms, having reliable transmission can make a tremendous difference in the efficiency of algorithms [65]. Furthermore, delivery of an indeterminate number of message copies may complicate the construction of distributed algorithms – knowing exactly how many you're going to get is often easier.

One or several specific representatives: A specific number of message copies delivered to particular representatives, all specified by the aggregate creator. This is perhaps the most flexible scheme we have considered thus far. From the point of view of the programmer, it may be even more desirable than the single, random representative scheme we have chosen. One or several specific representatives allows the user of a data abstraction to direct messages to the appropriate parts of the aggregate. However, this scheme puts the onus on the run time system to find the appropriate representatives. Further, in some cases the aggregate creator may not care which representative receives the message – functionally, it may not matter. Besides, if we provide reliable delivery to a single, random representative, the implementer of the abstraction can easily implement the message redirection and fanout required to emulate the several specific representatives behavior. The cost of this emulation is quite small – as low as a single message-passing operation. In fact, in cases where the routing does matter, a clever compiler might perform the optimization automatically – by looking for redirection and fanout code.

One arbitrarily chosen representative: We chose to cause messages sent to an aggregate to be delivered to only one representative object, and one chosen arbitrarily by the runtime system in an aggregate. We chose this scheme because it allowed for data abstractions with scalable bandwidth. This was important because the major reason we're interested in aggregates is to increase the bandwidth of our data abstractions. In addition, the random representative approach can be used to emulate all of the competing schemes we have described. Delivering a message to an arbitrary representative places little restriction on the runtime system. Perhaps this makes the translation the runtime system is performing less expensive.

3.4.3 Aggregates of Identically Structured Objects

Concurrent Aggregates requires that aggregates consist of collections of objects with identical structure. This means that each of the representatives in an aggregate have identical message handlers (behavior) and state structure. We considered allowing programmers to build aggregates of arbitrary collections of objects, but decided against for several reasons.

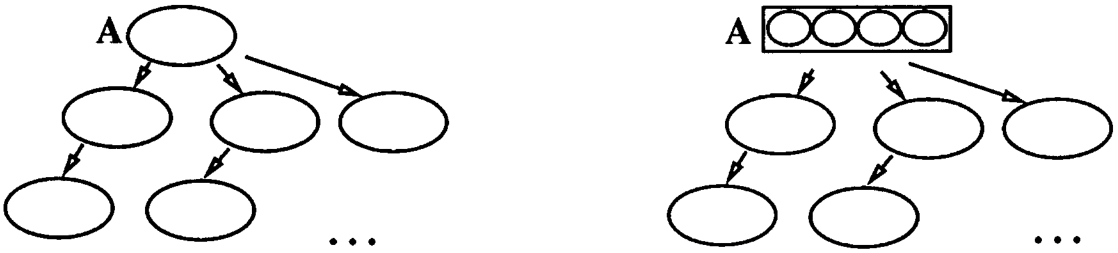


Figure 3.8: Evolution of a Program to Greater Concurrency: replacing object A with an aggregate.

First, any heterogeneous collection of objects can be built into an aggregate by nesting them inside a homogeneous aggregate – the overhead of one level of indirection is required. Second, given that we have chosen to send aggregate messages to an arbitrary single representative, each representative must provide the abstraction interface. If the interface is to be consistent, one easy way to do this is by having all representatives be homogeneous. Of course, one could achieve a uniform interface through delegation, but using homogeneous aggregates is perhaps a simpler fashion. Finally, if one is building aggregates out of objects – not designed to work together with others in an aggregate – it is unlikely that the objects will cooperate to provide a uniform interface. In fact, one might think it likely that messages routed to different representatives in the aggregate might give quite different results. Basically, objects that were not designed to work in an aggregate are unlikely to work together effectively in one. Another interesting fact is that one can think of a heterogeneous aggregate as a homogeneous aggregate with the one level of indirection compiled out. For example, if we had a statically typed language, we might be able to avoid the indirection.

3.4.4 Unified Object and Aggregate Model

Externally, object and aggregate implementations of abstractions should be functionally equivalent to limit programming complexity. A programmer should only have to deal with a limited amount of complexity that is relevant to the correct and efficient execution of his program. A user of a data abstraction should not have to think about the implementation of that abstraction in order to be able to use it. This view of aggregate as an implementation technique to build higher concurrency abstractions or to organize collections of objects in order to build highly concurrent abstractions compels us to unify the object and aggregate model. As much as possible, descriptions of objects and aggregates are similar. The usage interface of objects and aggregates is indistinguishable. This simplifies both the language and the programmers' view of the world.

One could imagine a methodology that involves a user constructing a program first from objects – of modest concurrency – and evolving his program by reimplementing the bottleneck abstractions with aggregates. This process is depicted in Figure 3.8. Such an approach would allow the implementation, debugging, and improvement of massively concurrent programs in a modular fashion. In fact, several of the applications described in Chapter 4 were developed using this methodology.

3.4.5 Method-Oriented Object Description

Concurrent Aggregates incorporates a method-oriented description of an object's program. I made this choice for several reasons: the method-oriented approach clearly specifies the interface of the object – consistent with our view of objects as data abstractions. This object interface is unchanging. A method-oriented approach also makes it possible to succinctly describe delegations of single message names and groups as part of the abstraction interface. This makes it possible to construct an object incrementally, by using a the interfaces of other objects.

Of course, behavior-oriented object descriptions that are used in languages such as ACORE [74] and SAL [2] also have advantages. For example, in a behavior-oriented object description, it is easier to express state transitions in an object. Especially state transitions that involve changes in the message interface. In method-oriented descriptions, such interface changes are often obscured as object state value changes. We chose not to use behavior-oriented object descriptions because we did not want to have objects with changing message interfaces.

3.4.6 First Class Continuations and User Continuations

With continuations, a programming language designer can take many different approaches – each yielding different degrees of safety and expressive power. Allowing the user to manipulate continuations can lead to errors – continuations never replied to and continuations replied to multiple times. It can also dramatically increase the cost of managing activation records. On the other hand, allowing explicit manipulation of continuations can simplify programs and make them more efficient.

The two main alternatives are to add specialized language constructs that allow a programmer to do a few useful things with first class continuations (e.g. supply `forward`) or to allow a programmer to manipulate continuations explicitly. One possible benefit of the former approach is that allowing only a subset of operations might constrain programmers to write only "safe" programs (programs without continuation errors). We reject this approach because even very small sets of continuation operations are not safe (in the context of our disposable system continuations). Furthermore, while `forward` is probably the most common use of first class continuations, there are many other interesting things that can be done with them. In Concurrent Aggregates, we not only allow continuations to be manipulated as first class objects, we go one step further and enable programmers to create continuations from user objects and use them in their programs.

Safety and Expressive Power Permitting programmers to manipulate continuations can make programs more expressive. In the context of message passing programs, this typically means that programs can control more precisely the pattern of message passing they construct. Programs are not bound by the procedure call – call and return – paradigm. We can consider giving programmer access to a safe set of primitives and enforcing their use in a safe manner. In fact, it is quite difficult to have a safe set of primitives without providing quite strong restrictions on how reply messages are derived. For example, `reply` by itself can be used in an unsafe manner as can the simple subset of `reply` and `forward`. Consider the following examples:

```
(method foo biz (a b c)
        (seq (if (predicate_1 a b) (reply c))
             (if (predicate_2 a b) (reply c))))
```

```
(method foo biz (a b c)
        (seq (if (predicate_1 a b) (forward (computation_1 b c))
             (if (predicate_2 a b) (reply c)))))
```

In both cases, the programs may be perfectly okay for solving the problem at hand. However, in neither case is it possible to determine that only one reply message will be made by the programs. Both of these examples illustrate unsafe subsets. These programs would have to be made illegal if we wanted to guarantee no run time errors due to continuations.

One conservative alternative is to do away with the `reply` expression. Each method could, by default return the value of the last expression evaluated. This is the convention in many sequential languages such as Common Lisp, for example. However this is undesirable as it restricts concurrency. The `reply` expression allows programs to reply with the return value as soon as it is available, not only when we reach the bottom of a method execution. Another way of solving this problem is to `reply` and `forward` terminate a method. This works for the examples above, but would inhibit concurrency in the examples below.

First class continuations can allow the program to be more efficient or concurrent. For example, consider the following example in which a subsequent message execution can begin much earlier because we have user visible continuations:

```
(method foo baz (a b c)
        (seq (long_computation_1 a b)
             (forward (long_computation_2 b c))
             (long_computation_3 a c)))
```

Without user visible continuations, a programmer might structure the same program in one of the two following ways:

Alternative 1

```
(method foo baz (a b c)
       (seq (long_computation_1 a b)
            (long_computation_3 a c)
            (reply (long_computation_2 b c))))
```

Alternative 2

```
(method foo baz (a b c)
       (seq (long_computation_1 a b)
            (let ((result (long_computation_2 b c))
               (conc (long_computation_3 a c)
                     (reply result)))))
```

Alternative 1 orders long_computation_2 and long_computation_3, causing unnecessary serialization and thereby reduced concurrency. Alternative 1 also requires an extra message passing operation on the return path. The invocation of **baz** must receive the value from long_computation_2 and send it along to the real user of the value.

Program Alternative 2 allows long_computation_2 and long_computation_3 to go on concurrently, but the return path still requires an additional message passing operation. In both alternatives, the extra message passing operation in the return path could be optimized by a compiler that does *tail-forwarding* [57], the distributed machine analog of tail-recursion, in this simple example. While tail-forwarding is useful in many cases, there are many others that can be improved by explicit manipulation of continuations. Consider the following example in which we know exactly one of the two subtasks will return a value. However, we do not know *a priori* which it will be.

```
(method foo buzz (a b c d)
   (conc (forward (computation_1 c d))
         (forward (computation_2 a b))))
```

This computation is difficult to express without additional message passing operations in a language without explicit continuations. No set of "safe" primitives will allow the construction of this program, because its correctness depends on the behavior of the user programs computation_1 and computation_2. Since there is no reasonable "safe set" or operations dealing with continuations, the programmer must already think about continuations. Given that he is managing continuations already, it is not a large step to allow him to explicitly manipulate them.

User Continuations Making continuations explicit enables programmers to construct many interesting structures such as disjunctive subtasks, memory with presence bits [90], and I-structures [10]. First class continuations allow the programmer to decouple the forward calls (downward) from the upward replies in the dynamic control structure of a program. For example, the **forward** construct allows the reply path of the program to bypass one of the intermediate objects in the calling path. This decoupling can increase the efficiency of the program.

We have found that we can construct even more interesting program structures by allowing a programmer to build his own continuations. In Concurrent Aggregates, we take decoupling one step further. As continuations are simply objects that accept reply messages, we allow a programmer to construct abstractions that handle reply messages and use them as continuations.

Because system continuations are already first class, user continuations can be substituted quite easily by using the do construct. This power can be used to implement complex synchronization structures ranging from a barrier, a transparent future [61], race (speculative concurrency [70]), to much more complicated subset synchronization dependencies. Allowing user continuations allows this synchronization code to be factored out from the remainder of the program – allowing modules from a program to be reused in a variety of different synchronization contexts.

Efficiency We chose to make system continuations in CA a little different from continuations in sequential languages. System continuations in Concurrent Aggregates are *use-once* or *disposable*. They can only be used once. A system continuation in CA corresponds to a single return location for a value. A reply message to a system continuation fills that location and resumes the suspended location (if necessary). Programs may only reply to a system continuation once – subsequent replies would indicate an incorrect program. This differs from continuations in Scheme, which can be called multiple times. The behavior we have chosen for system continuations allows activation records to be collected as soon as all of their continuations have been fulfilled, reducing a global garbage collection problem to a local one.

In several programming languages, researchers have shown that first class continuations can be supported at reasonable cost. Compile time analysis coupled with the appropriate run time support has been shown to significantly reduce this cost in sequential programming languages [47, 32]. The analysis is used to determine when a reference to a continuation may be created and to handle that case specially. If first class continuations are used only rarely, the vast majority of activation record allocations can be done in stack fashion – keeping their cost down. When first class continuations are actually used, they are still quite expensive. Their use requires that activation records be garbage collected, not stack deallocated.

We would like to make widespread use of first class continuations as a means for altering message passing structures. This means that we must make actual use of first class continuations quite cheap. In a system where garbage collection may be quite expensive (due to the distributed memory and limited network bandwidth) we cannot afford to garbage collect the continuations. Thus we settled on the single-use continuations as an acceptable model.

3.4.7 First Class Messages

Many language designers have wrestled with issues of how to support meta-programming [1]. In a function-oriented language such as Scheme, meta-programming is made possible through first class functions and the apply operation [81]. First class functions allow functions to be manipulated by programs. The apply operation allows the dynamic invocation of function values. The fact that the apply operation can accept a variable number of arguments solves the function arity problem (how to write meta-code that works for functions of differing arities). Other languages use first class functions and currying to serve a similar purpose [104, 8, 9, 51]. While these function-oriented approaches are not directly applicable in a message-passing language, we use their example for the basis of our design.

Concurrent Aggregates uses first class messages to support meta-programming. Messages are used as unevaluated applications, containing a selector (message name), receiver object, and some number of arguments. By manipulating messages as applications and evaluating them by sending the messages, CA programmers can write meta-programs that are independent of message arity concerns (in the same sense that other languages allow the writing of programs independent of function arity).

Once we decided that we wanted to allow a programmer to manipulate messages explicitly, it became clear that rather than develop a specialized protocol, the most practical way to manipulate them was to view messages as objects. Treating messages as first class objects is nice, because it allows everything in the system to be treated uniformly, as an object. Of course, because the efficiency of messages is crucial to the efficiency of an implementation, they are managed specially. First class messages in Concurrent Aggregates allow programmers to do a number of things: write code to reorder messages, write abstractions that implement control structures such as `doall` that do not depend on the arity of the body (and thus can be reused), and implement complex synchronization structures depending on explicit manipulation of messages.

Because messages are used for communication in our system, access to messages must be local. If they are, we can avoid an infinite regress problem[4]. To achieve this, we decided to make messages by-value objects. By this we mean that whenever a reference to a message is logically transmitted[5] (in another message), the message is copied so that the reference always points to a local copy[6]. Often, this will result in the creation of additional message

[4] We need to send a message to change a message. But we need to access a message to send that message and so on recursively.

[5] By logically transmitted, we mean even in the case the transmission is to the same node and requires no actual physical transmission.

[6] This assures that the first level of message references is always local.

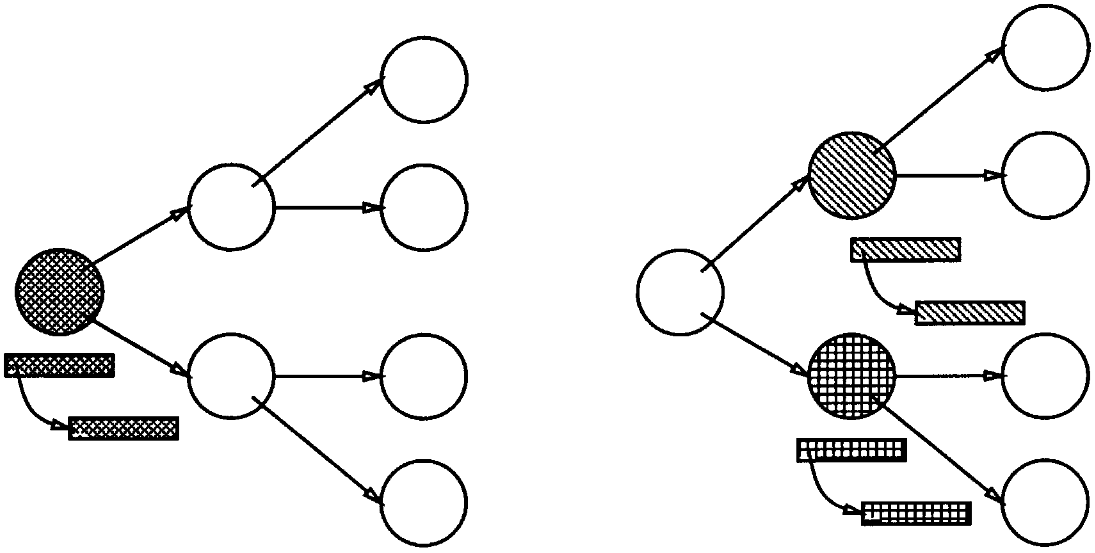


Figure 3.9: Message duplication in a Fan-out Tree

copies. This assures us that a **send** operation can be performed locally. In addition, in meta-programs that use message parameters to support data parallelism, the message parameter is replicated just as desired by default. Making messages by-value parameters causes some additional implementation work, but has the advantage of the desirable sharing and locality properties[7].

One good example of this is a fan-out tree, used to implement a data-parallel operation on a collection. This example is illustrated in Figure 3.9. At each stage in the fan-out tree, the message parameter is duplicated and colocated with the computation holding a reference to it. This results in the right number of copies at the leaves, one for each use of the message. Throughout the fan-out process, the message parameters are always local.

Efficiency If messages were managed exactly like other objects, the overhead would be tremendous. We observe that messages are different from other objects in three important ways. First, messages move a great deal. Second, typically there are only one or zero references to a message[8]. Finally, messages are typically created and destroyed at a much higher rate than objects.

[7] Another designer might have simply chosen to make messages immutable as noted by Dave Gifford. This would allow the compiler to decide if messages need to be copied when fields are replaced or if, as in the case of no sharing, they can be updated in place.

[8] Remember, messages are active objects when they are sent, so they should not be garbage collected simply because there are no references to them. When they become passive objects, they require at least one reference or they are subject to garbage collection.

In order to support efficient execution, messages are treated specially – they receive only local names. This is okay for most cases, as their by-value property causes most messages manipulated by programs to be local. When the single local reference is destroyed, the message can be reclaimed without any garbage collection. Local names for messages means that the operating system need only do a little bit of local work, and no communication to allocate and deallocate them. In rare cases (when a message points to a message), a message name is exported to a non-local location. At that time, a full relocatable object name must be allocated for the message. In our experience this does not happen often. The majority of the time, messages only receive local names – hence their management is inexpensive and requires no communication.

3.5 Summary

In this chapter, I have described a new programming language Concurrent Aggregates (CA). CA is a concurrent object-oriented language that allows programmers to build cooperating collections of objects, aggregates. These collections can be used to construct hierarchies of data abstractions without causing serialization. This allows programmers to use the appropriate levels of abstraction without concern for reducing concurrency.

Concurrent Aggregates integrates programming with aggregates with a more familiar object-oriented model. Message sends give rise to concurrency and message passing is used for all synchronization in the system. In order to support programming with aggregates, the design of CA includes support for intra-aggregate addressing, delegation, manipulating messages as first class objects and manipulating continuations as first class objects. Intra-aggregate addressing makes it convenient to build an abstraction from a collection of objects. Delegation allows programmers to build abstractions incrementally from parts of other objects. First class messages can be used for meta-programming, for example to construct data parallel programs. First class continuations can be used for special synchronization structures that specifically suit the requirements of the application at hand.

We have also discussed a number of issues in the design of Concurrent Aggregates. Non-serializing data abstractions should be allowed, and they need not be fully coherent. Allowing then to be slightly inconsistent can significantly reduce the cost of maintaining distributed state. Messages to aggregates should be received by only one representative because such a scheme is scalable and can be used to construct *any* other scheme. Aggregates are made up of identically structured objects because unless they are designed to work in an aggregate, objects are not likely to be able to cooperate effectively.

Concurrent Aggregates unifies the aggregate model with its object model, allowing objects and aggregates to be used interchangeably. CA uses a method-oriented behavior description as this meshes nicely with the *per message* delegation – allowing a programmer to construct an interface incrementally. Concurrent Aggregates allows programmers to manipulate system continuations as first class objects and to create their own continuation structures. This ability allows synchronization structures to be factored out from programs

– making the computation kernels more reusable. Concurrent Aggregates also supports first class messages. Messages are by-value, so operations on them are nearly always local. First class messages can be used to write meta-programs – extending the CA language.

Chapter 4

Language Evaluation

In order to evaluate our programming language design we have implemented Concurrent Aggregates, written and executed a number of application programs. Our experimentation and evaluation has focused on the primary innovation in Concurrent Aggregates – non-serializing data abstraction tools. Programmers may use aggregates to explicitly control distribution and consistency of state across the representatives in an aggregate. This freedom can be used to implement a wide variety of cooperative structures – replicated state, partitioned state, and complex cooperation. In this chapter, we describe a number of application programs and use them to evaluate the Concurrent Aggregates language. We also evaluate the efficiency and concurrency of our CA programs.

Design and study of programming languages for machines much more powerful than those currently available is difficult. Due to the performance difference between existing and target machines (three to four orders of magnitude), it is hard to write and run real application programs[1]. Emulation of a different machine architecture implies an additional execution overhead. To run programs that are as realistic as possible (and therefore quite large), an efficient implementation is essential.

In implementing CA, we have taken a hybrid approach that yields reasonable performance, while maintaining system portability. We can simulate reasonably large programs (millions of message-passing operations), yet can move to faster computers as they become available. In order to be portable, we chose to compile CA into another high-level programming language. We selected C++ as our target language because it is becoming widely available and its object-oriented style is a good match to Concurrent Aggregates. Each method or handler in CA is compiled to a function in C++. Thus, each method can be executed as straight-line code – all interpretation at this level is compiled out. Between methods, the linkage (message sending and method invocation) is provided by a runtime system written in C++. This runtime system can simulate a variety of different message passing machines ranging from an idealized message passing machine to a bounded resource machine.

[1] It can take many hours to simulate just one second of machine time!

We have used Concurrent Aggregates to write a number of application programs:

- Matrix Multiplication
- Multigrid Relaxation Solver
- N-body Interaction Simulation
- Printed Circuit Board Router
- Parallel FIFO Queue
- Concurrent B-tree
- Logic Simulator

These application programs have showed us both strengths and weaknesses of the language. This chapter presents our evaluation of Concurrent Aggregates. First, we present the application problems we studied and describe the algorithms used to solve them. For each application, we briefly describe its program structure. Second, we present an overall analysis of Concurrent Aggregates. In this evaluation, we consider the importance of multi-access data abstraction tools and then discuss issues of program efficiency, modularity and concurrency.

4.1 Application Studies

In this section, we present the seven applications we used to evaluate the Concurrent Aggregates language. For each application program, we first describe the problem it solves. Then, we describe the algorithm we are implementing and the resulting program structure in Concurrent Aggregates. Finally, we present a brief summary of where aggregates were useful in the program. A more detailed description of several CA programs can be found in Appendix A. Documentation, source code, and detailed simulation statistics for all of our application studies can all be found in [29].

4.1.1 Matrix Multiplication

Matrix multiplication is an operation that occurs in many algorithms. We consider a simple algorithm for multiplying dense matrices. More efficient algorithms based on decomposition are known [91, 102]. If many elements of the matrix are zero (the matrix is sparse), special algorithms that are much more efficient can be used. Matrix multiplication of two n by n matrices A and B to define matrix C is computed according to the following rule:

$$C[k,l] = \sum_{i=1}^{n} A[k,i] * B[i,l]$$

From this definition, we see that there is opportunity for massive concurrency in matrix multiplication. Each element in the result matrix can be computed concurrently – none depends on any other. This means that a naive matrix multiplication scheme would have up to n^3 concurrent operations![2] For a 100 by 100 matrix, this would be one-million-fold concurrency. The average concurrency is $\frac{n^3}{logn}$, or 150,514 for the 100 by 100 case.

Attempting to exploit the peak concurrency may be neither desirable nor achievable. We chose a formulation of matrix multiply that yields n^2 concurrency. Each matrix multiplication is divided into n^2 sequential inner products. Each dot product defines one element of the result matrix. By constraining each dot product to be sequential, we limit the concurrency to be n^2. As we have n^2 dot products and each can only have one outstanding request at a time for each matrix, each matrix has only n^2 readers. This is well matched to the number of readers we can actually support without using replication.

The matrix multiply computation in CA is depicted in Figure 4.1. It shows a number of dot product computations computing against the two operand matrices. As the dot product proceeds, it reads across a row of the first matrix and down a column of the second matrix. At each step, a partial sum is added to the dot product. When the dot product reaches the end of both matrices, it stores the value of the dot product in the result matrix (not shown). This decomposition of matrix multiply into a number of computations, each computing a part of the result, has been called *result concurrency* [23]. Each dot product is performed at the result matrix (i.e. values are read from two term matrices and processed at the result matrix). This approach results in lower contention at the term matrices, at the expense of doubling the message traffic.

We implemented each matrix as an aggregate. This allows computation to be performed on different parts of the matrix simultaneously. As each matrix is an aggregate, it can be described with an ordinary name and passed, stored and manipulated as any other object. Of course, users of the matrix abstraction cannot tell how it is implemented, save through performance differences.

Each representative in the aggregate holds one element of the matrix. While this very fine grain partition implies some storage overhead, it also allows maximal concurrency – all n^2 elements of the matrix can be accessed simultaneously. The basic matrix operations, **at** and **atput**, each require three message passing operations: the initial request, one message to forward the request to the right representative, and one message to send the reply. A basic matrix operation and the message handlers required to implement them are shown in Figure 4.2. External requests **at** and **atput** are converted to internal requests and forwarded to the correct representative. This forwarding passes the continuation of the original request to the internal request, causing the reply to go the source of the external request. In Section 5.4.1, we show how this forwarding message can be eliminated at compile time.

[2] Each element in the result matrix involves $2n - 1$ operations. Of these operations, at least initially, n of them can be done concurrently, assuming we perform a fully parallel vector element-wise multiply then a tree reduction for each result element.

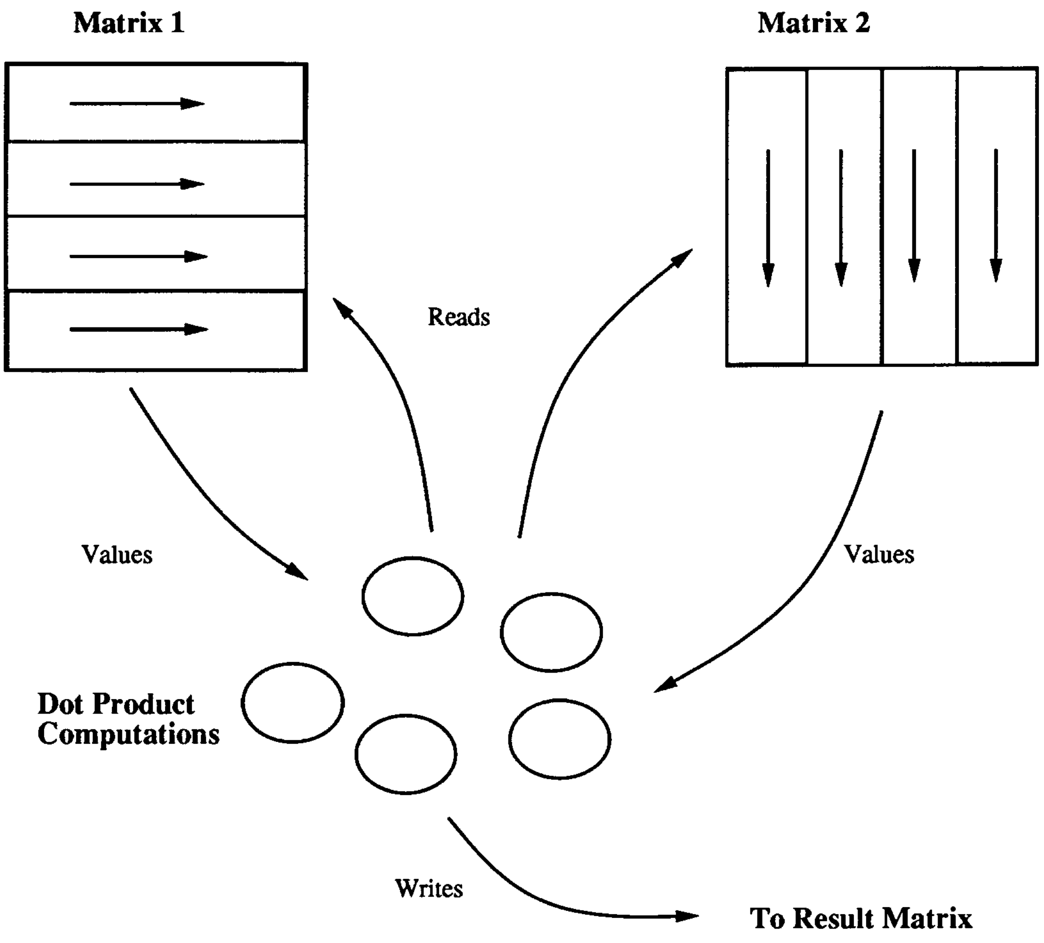
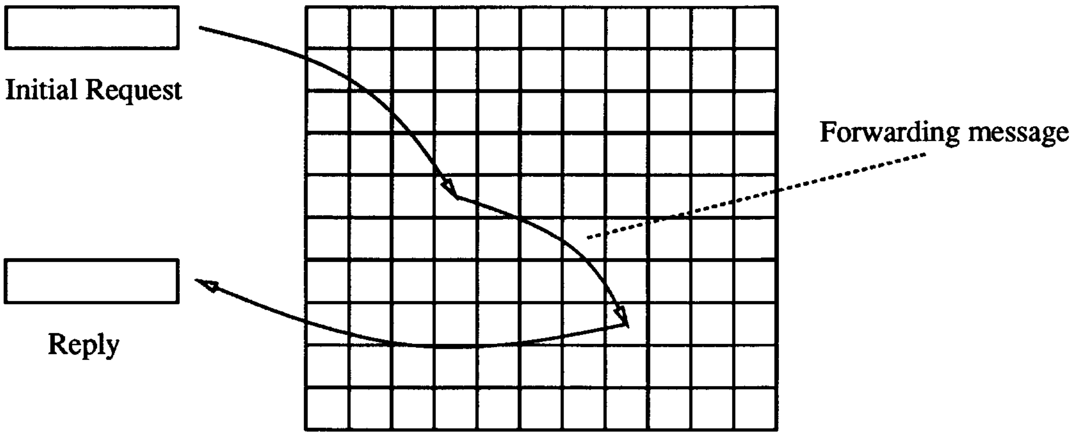


Figure 4.1: Matrix Multiply in Concurrent Aggregates



```
;; A 2-D matrix aggregate
;;   forward external requests according to our mapping function
(aggregate matrix_2d state xsize)

(handler matrix_2d at (xindex yindex)
   (forward (internal_at (sibling group (+ (* xindex (xsize self))
                                           yindex)))))

(handler matrix_2d internal_at ()
   (reply (state self)))

(handler matrix_2d atput (value xindex yindex)
   (forward (internal_atput (sibling group (+ (* xindex (xsize self))
                                              yindex)) value)))

(handler matrix_2d internal_atput (value)
   (seq (set_state self value)
        (reply (state self))))
```

Figure 4.2: Basic Matrix Operations

```
let G[i] be grids for i = 0,1,2,3,4,...m
Where the grid spacing in grid G[m] is $2^m * h$ (coarsest grid) and h is the grid
spacing of the bottom grid, G[0] (finest grid). We assume that the bottom grid
is $n$ by $n$ with $n = 2^i$ for some integer $i$. Thus G[i] is $(n/2^i)$ by $(n/2^i)$.

 1. Start with $i = 0$ (finest grid)
 2. Relax several iterations on G[i] to obtain an approximate solution to the
    equation.
 3. Inject (use a restriction operator) the residual error from G[i] to G[i+1].
 4. $i = i + 1$; If $i < m$, go to step 2
 5. Iteratively relax on G[i] = G[m] until found solution
 6. Interpolate solution from G[i] to G[i-1] (use a prolongation operator)
 7. $i = i - 1$
 8. Relax several iterations on G[i]
 9. If $i > 0$ go to step 6
10. Return G[0], the solution.
```

Figure 4.3: The Multigrid Algorithm

4.1.2 Multigrid Solver

Multigrid is an efficient algorithm for numerically solving partial differential equations [76, 73]. It is an indirect method based on finite differences. The variable of interest is represented over a continuous space by a set of values at a set of discrete points called grid points. By performing successive relaxation operations (averaging local grid values), we solve for the variable over the space.

Multigrid is an improved relaxation-based solution technique. The crucial observation in multigrid is that relaxation techniques efficiently reduce the high frequency components of the error (the difference between the computed solution and true solution). By viewing the error on successively coarser grids, low frequencies in the error become high frequencies and hence can be reduced effectively by relaxation techniques. Thus, multigrid makes use of a hierarchy of grids. As we go up in the hierarchy, the grids become smaller. Typically, the grid sizes get exponentially smaller, so only a few levels of grid are needed even for large problems. The multigrid algorithm is described informally using pseudo-code in Figure 4.3.

Multigrid is an efficient algorithm on both sequential and concurrent machines. Define n to be the number of grid points in the finest grid. Multigrid algorithms are attractive because they require only $O(n)$ operations. Simple relaxation schemes require n iterations of $O(n)$ operations each, $O(n^2)$ operations overall. At any grid point, the successive iterations

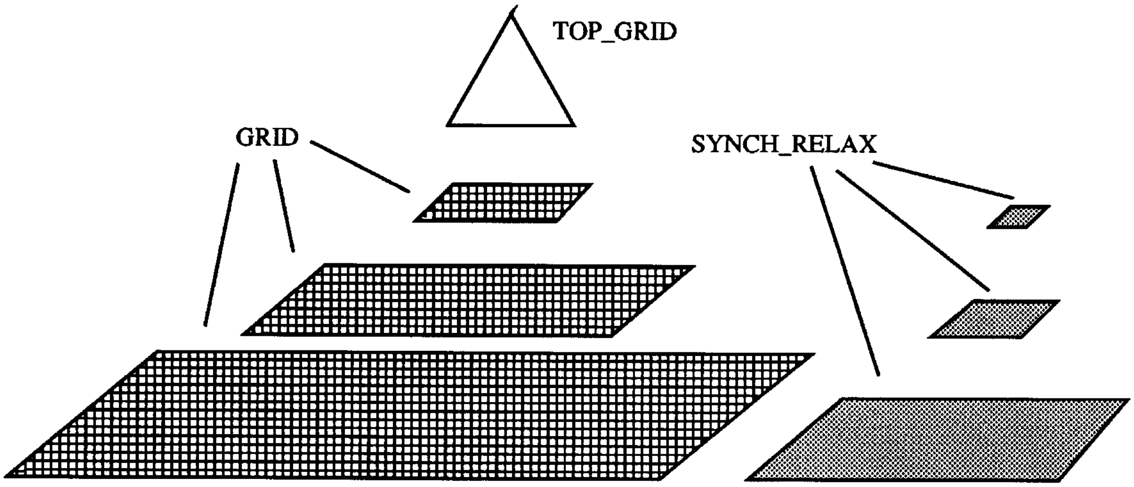


Figure 4.4: The Multigrid Algorithm in Concurrent Aggregates

are sequential, resulting in a critical path length of $O(n)$ operations. In multigrid, the depth of the hierarchy is $log\sqrt{n}$ and the number of operations at each level, i, is $O(\frac{n}{4^i})$. Thus, the total work is dominated by the lowest level and the overall work is $O(n)$. The critical path is determined by the time to go up through the hierarchy and back down (we lump the number of iterations at each level into the constant factor) and therefore is $O(log\ n)$.

The multigrid algorithm in Concurrent Aggregates consists of a number of abstractions which are composed to form the multigrid solver. The basic structure of the solver is a hierarchy of grids – each implemented by an instance of the **grid** abstraction.

Each **grid** abstraction is a simple relaxation solver that performs a fixed number of iterations. Each **grid** is paired with a **synch_relax** abstraction which enforce the necessary synchronization for successive iterations of the relaxation. This is necessary to prevent a grid point from getting too far ahead and causing some grid values to be computed using values from the wrong iteration. The per grid point synchronization provided by **synch_relax** allows different parts of the grid to get out of synchrony but never too far – allowing the exploitation of inter-iteration and inter-grid concurrency.

After a **grid** has finished its relaxation steps, it injects (interpolates) the grid values to its upper (lower) grid neighbor. This operation is performed by a restriction (prolongation) operator. Each grid does this in turn. When the computation reaches the top of the hierarchy, the **top_grid** abstraction performs a barrier synchronization and starts the algorithm back downward. On the way down the hierarchy, the multigrid also consists of a fixed number of iterations for each relaxation solver. When we reach the bottom of the hierarchy, we have solved the partial differential equation. The upward then downward traversal of the grid is often called a V-cycle. The structure of multigrid in CA is depicted in Figure 4.4.

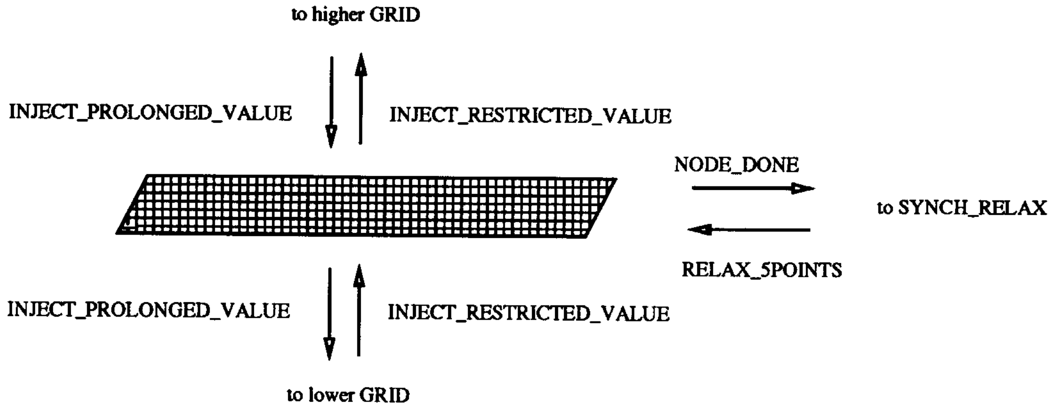


Figure 4.5: Interfaces for the Grid Abstraction

Novel CA language features are used in a number of places in the multigrid program. Aggregates are used for the **grid**, **synch_relax**, **top_grid**, and **barrier_tree**. **barrier_tree** is a dynamic combining tree used in the concurrent barrier synchronization abstraction. The aggregates, multi-access abstractions, allow the grids to be connected to each other by single aggregate pointers.

The multiple access property of aggregates allows the synchronization code to be factored out of the **grid** abstraction. Without multi-access abstractions, the serialization at an abstraction boundary would make the factoring infeasible. The **synch_relax** abstraction is a highly concurrent abstraction based on an aggregate that supports fine-grain synchronization for the grid. One consequence of this improved program modularity is that **synch_relax** could be replaced with a simpler structure that performs a coarser-grain synchronization with no change to the multigrid relaxation code.

In order to illustrate how Concurrent Aggregates allows us to construct clean interfaces for abstractions, we depict the **grid** aggregate with interfaces, in Figure 4.5. A **grid** interface has three parts: to the upper grid, to the lower grid and to its cooperating **synch_relax** aggregate. These interfaces are defined by handlers and message sends used in the code shown in Figure 4.6. The interface to the **synch_relax** abstraction is very clean. Each completing local relaxation operator ends with a message **node_done** to the **synch_relax** abstraction, indicating its completion. When the **synch_relax** abstraction determines that the next local relaxation operator can be computed, it sends a **relax_5points** message to the **grid**, beginning the next local grid point computation.

```
(aggregate grid value
           count synch_node
           scratch_value
           xsize
           restrict_prolong ;; 0 for up, 1 for down
           upgrid downgrid
           :no_reader_writer
           (parameters totalsize ixsize ival nr_iters)
           (initial totalsize ... ))

;;
;; Jacobi method indirection solution
;;    (also interface to synch_relax grid)
(handler grid relax_5points (local_x local_y)
... send relax_5points_step messages to neighbors ...)

(handler grid relax_5points_step (xval yval val)
        ...
        (if (= (count self) 4)      ;; finished local operator
            (do (node_done (synch_node self)))
          ...))                     ;; tell the synch_grid we're done
;;
;; upward and downward interfaces
;;
(handler grid inject_restricted_value (val x y)
        ... put it in the right place and start relaxation ... )
(handler grid inject_prolonged_value (val x y)
        ... put it in the right place and start relaxation ... )
```

Figure 4.6: The Code for a Grid abstraction

```
(aggregate top_grid downgrid msg_list barrier_tree
        ...)

;; Catches the messages, Modifies the selector,
;;    receiver and two arguments and Saves the message
;; Count for barrier_synch message indicates how many nodes
;;    are done
;;
(handler top_grid inject_restricted_value (val x y)
  (seq (msg_atput msg inject_prolonged_value 0)
       (msg_atput msg (downgrid self) 2)
       (msg_atput msg (* 2 x) 4)
       (msg_atput msg (* 2 y) 5)
       (set_msg_list self
           (new msg_pair msg (msg_list self)))
             (do (barrier_synch (barrier_tree self) 1))))
```

Figure 4.7: The code for a Top_Grid abstraction

In order to make all grids identical, we constrained the top_grid abstraction to have an interface compatible with the upward grid interface. Upward messages are caught by top_grid and used to synchronize the computation globally. First class messages are used by the top_grid to halt the upward phase of the computation, synchronize and proceed to the downward phase. The top_grid catches, modifies and resends the messages as shown in Figure 4.7.

4.1.3 N-body Interaction Simulation

The N-body interaction problem involves action at a distance (the force of gravity) and the motion of bodies over a period of time. We use a simple algorithm that explicitly calculates $\frac{n(n-1)}{2}$ force interactions and uses them to update the velocities of the bodies[3]. Our N-body simulation proceeds as a number of time steps, each overlapped with the one before. At time k, to proceed, each body must compute the following:

$$F_j(k) = \sum_{i=1}^{n} F_{ij}(k)$$

When a body has computed the force acting on it in the current time step, that force is used to update its velocity. The new velocity is used to move the body and start the process again. Because the force $F_{ij}(k+1)$ depends on the position of both bodies i and j

[3] More efficient algorithms have been invented [52, 107], but we choose this simple algorithm as the most convenient means of studying the language.

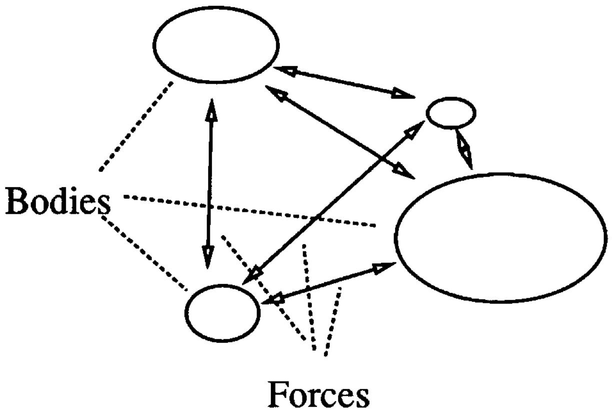


Figure 4.8: Bodies interacting via gravity.

for time step k, we see that no body can get more than one step ahead of any other. This is because the new position for each body depends on the current position of all the other bodies.

Force computations and position updates are not synchronized by any global control. They are triggered by the implicit dependences for data. For this algorithm, the CA program implements the minimal constraint – the computation proceeds as fast as the data can be computed and communicated.

Our N-body program uses an aggregate to implement a bodies abstraction which contains the state of all N bodies. All messages for a particular body are sent to the aggregate and forwarded to the appropriate representative based on an argument specifying the number of the destination body. This scheme allows us to avoid linking a body to all of its interactions – saving $n - 1$ elements of state for each body. In addition, due to the fact that each body acceleration requires the accumulation of $n - 1$ forces, each body also has a dynamic combining tree. Force messages are first forwarded to the appropriate body and then on to the appropriate dynamic combining tree[4]. This tree can accumulate the forces concurrently, producing the total force more quickly than a linear reduction. We chose a dynamic combining tree instead of a static combining tree because we do not know the likely arrival order of forces for a particular body. Its data dependent. Also, dynamic combining trees are generally more space efficient as they require space proportional to the maximum rate at which requests arrive. In comparison, static trees usually require space proportional to the total number of requests.

[4] The serialization of force messages through the body before reaching the combining tree does not limit combining in this case. It takes much longer to combine two force vectors than to forward the force message to the tree.

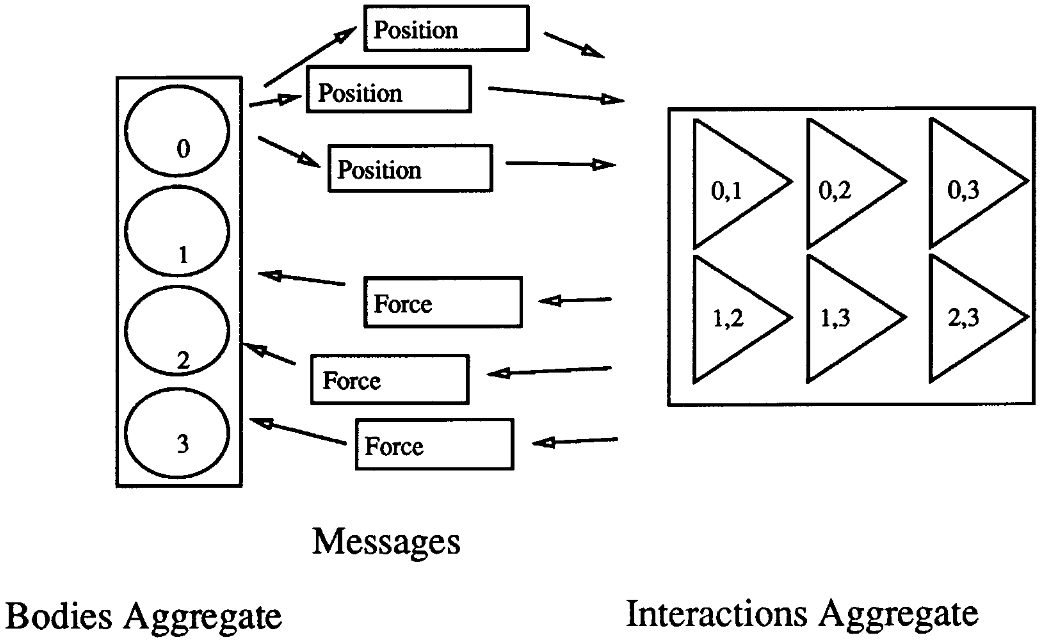


Figure 4.9: The Bodies and Interactions Abstractions

An aggregate is also used for the interactions abstraction which computes all of the forces due to body interactions. Each interaction receives position messages from two bodies, computes their interaction, and sends the resulting force messages to each one. We need not worry about linking the bodies directly to the interactions either. As in the bodies aggregate, the linkage is achieved via index computation and message-passing within the interactions aggregate. The division of the N-body simulation into two abstractions is depicted in Figure 4.9. The members of the body abstraction are labeled with their body numbers. The interactions are labeled with the numbers of the bodies whose interaction they calculate.

Each body reports its position to the interactions aggregate, and in return receives messages telling it how to accelerate itself. This clean separation of the two abstractions allows the implementation of each abstraction to change independently. For instance, we originally used a linear reduction to accumulate the forces on a body. However, the contention caused by our spin-locking concurrency control scheme resulted in an excessive number of messages. To solve this problem, we changed the bodies aggregate to include a combining tree for each body – reducing contention due to spin-locking. This change to the bodies aggregate required no change in the rest of the program. The changes to the bodies aggregate are shown in Figures 4.10 and 4.11. In the original implementation, each body processed a series of accelerate_body messages, one for each interaction. This series of computations amount to linearly reducing the forces on each body. The modified program delegates accelerate_body messages to a dynamic combining tree which consolidates requests. The

```
(aggregate bodies location velocity mass acc_count
           interactions iters
        ...)

(handler bodies accelerate_body (force_vector)
         (seq (set_velocity self (add (scale force_vector
                                             (/ 1.0 (mass self)))
                                      (velocity self)))
                ... increment mycount by 1 ...
                ... if we have all forces ...
                       (forward (update_position self 1.0))))
```

Figure 4.10: Accumulating Forces on a Body: Linear Reduction

```
(load "dcombtree.ca")  ;; include combining tree abstraction

(aggregate bodies location velocity mass acc_count
           interactions iters dcombtree
        ...)

;; delegate the accelerations to dcombtree
(delegate bodies accelerate_body dcombtree)

;; combined request
;;
(handler bodies really_accelerate_body (force_vector count)
         (seq (set_velocity self (add (scale force_vector
                                             (/ 1.0 (mass self)))
                                      (velocity self)))
                ... increment mycount by count ...
                ... if we have all forces ...
                       (forward (update_position self 1.0))))
```

Figure 4.11: Accumulating Forces on a Body: Using a Combining Tree

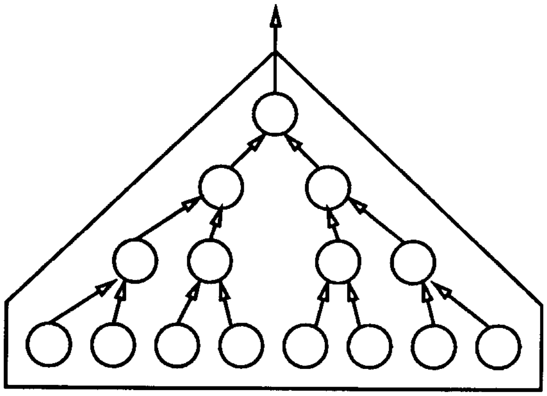


Figure 4.12: A Dynamic Combining Tree of size 16

consolidated request is presented to the body as a really_accelerate_body message. It needs one more argument – count – which indicates the number of forces that were consolidated into this request.

Dynamic Combining Trees Dynamic combining trees are a simple abstraction with widespread use. We used combining trees in the N-body simulation and many of the other application programs. A dynamic combining tree aggregate is shown in Figure 4.12. The Concurrent Aggregates code for a dynamic combining tree is shown in Figure 4.13.

A dynamic combining tree accepts a number of values – one from each of its leaves – and combines them with some associative binary operation. The resulting value is produced at the top of the tree. Our combining tree aggregate accepts a number of messages of the form: `(accelerate_body <combiningtree> <value> <count>)`. The selector stored in the combining tree specifies the binary operation that should be used to combine values. Value is the value to be combined and count specifies how many values have been combined together.

This tree can be adapted for other messages by changing the name of the `accelerate_body` handler to another name. It's also possible to build a general combining tree (one that works for all kinds of messages) by using first class messages[5]. The dynamic combining tree aggregate uses each representative as a node in a combining tree. The initial code connects the representatives into a tree structure and initializes the state of each tree node. The combining function is performed by the code for an `accelerate_body` message. If there are no waiting requests – this is the first of a set of `accelerate_body` messages to be combined – we send the `send_requests` message to the current tree node. When `send_requests` eventually gets processed, the tree node will send its consolidated request to its parent in the tree. In

[5] This is a minor problem with the language. The reusability of these abstractions would be enhanced if CA allowed the definition of a default or `:rest` handler.

```
;;
;;  A dynamic combining tree
;;
(aggregate dcombtree waiting_reqs
           myparent consolidated_value selector :no_reader_writer
           (parameters size top comb_selector)
           (initial size          ;; initial method received by sibling 0
                (seq (forall index from 0 below groupsize
                          (init_help (sibling group index) comb_selector))
                     (set_myparent self top))))

(handler dcombtree init_help (comb_selector)
         (seq (set_waiting_reqs self 0)
              (set_consolidated_value self 0)
              (set_myparent self (sibling group (/ myindex 2)))
              (set_selector self comb_selector)
              (reply done)))

;; if waiting requests, combine.  Else push and send send_requests
;;
(handler dcombtree accelerate_body (val count)
     (seq (if (eq 0 (waiting_reqs self)) (do (send_requests self)))
          (if (eq (consolidated_value self) 0)
              (set_consolidated_value self val)
            (let ((sel (selector self)))
              (set_consolidated_value self (sel val (consolidated_value self)))))
          (set_waiting_reqs self (+ (waiting_reqs self) count))))

(handler dcombtree send_requests ()
         (seq (if (= myindex 0)
                  (do (really_accelerate_body
                           (myparent self)
                           (consolidated_value self) (waiting_reqs self)))
                (do (accelerate_body (myparent self) (consolidated_value self)
                                (waiting_reqs self))))
             (set_waiting_reqs self 0)
              (set_consolidated_value self 0)))
```

Figure 4.13: An Implementation of a Dynamic Combining Tree

the interim, all **accelerate_body** messages will be combined together. The root tree node is connected to an object outside the aggregate. At the root, a **really_accelerate_body** message is sent to the body. Exactly how much combining occurs depends on the rate of requests and the system message queueing policy. If messages queues are FIFO, then all requests that are in the message queue already when a tree node gets the first request will be combined. If none are in the queue, then the request will be propagate up the tree almost immediately.

4.1.4 Parallel FIFO Queue

Our next application is a parallel first-in-first-out (FIFO) queue. Its behavior is linearizable in the sense defined by Herlihy and Wing [54]. This means that queue operations whose durations (time from request to reply) overlap may appear to occur in either order. If requests are applied serially to the queue, its behavior is exactly FIFO. Our queue is similar to one described by Schwartz in [83]. Our queue is implemented entirely in software. Schwartz's queue requires a hardware combining network used to compute queue indices. Our queue can be scaled up to almost arbitrary size and throughput by increasing the number of representatives in the synchronizing array, queue interface, and combining trees. A more detailed description of the queue with select code fragments is shown in Appendix A.

An interesting feature of our parallel queue is that it is not only an element buffer. It's also a dequeue request buffer. We normally think of a queue as a buffer for a non-negative number of elements. However, our queue is essentially just synchronizing between readers and writers (dequeuers and enqueuers) it can serve as a buffer for dequeue requests also. For a queue of size l, the producer can get l elements ahead or behind his consumer. However, one restriction of our implementation is that only l requests can be active simultaneously[6].

Our parallel queue program consists of five parts: a parallel interface, two index-request combining trees, a synchronizing array, and a counter that holds the enqueue and dequeue pointers for the queue. All of these abstractions, save the counter, are implemented using aggregates. The structure of a parallel queue is shown in Figure 4.14.

The enqueue and dequeue counters are used to manage the synchronizing array in a "circular" fashion, reusing the array when the indices exceed the array size. When a queue request is made, it is received by the interface. The interface acquires the appropriate type of index (enqueue or dequeue) by sending a request to one of the dynamic combining trees. These trees are very similar to the tree described in Section 4.1.3. The counter allocates a range of indices for each request. This range is mapped to the set of combined requests. Details of the mapping are in Appendix A. With an index, the interface object makes a request against the synchronizing array. When the interface has successfully read or written the array, it returns the appropriate value to the originator of the queue request.

[6] This restriction allows our synchronizing array locations to be fixed size. They require no dynamic allocation.

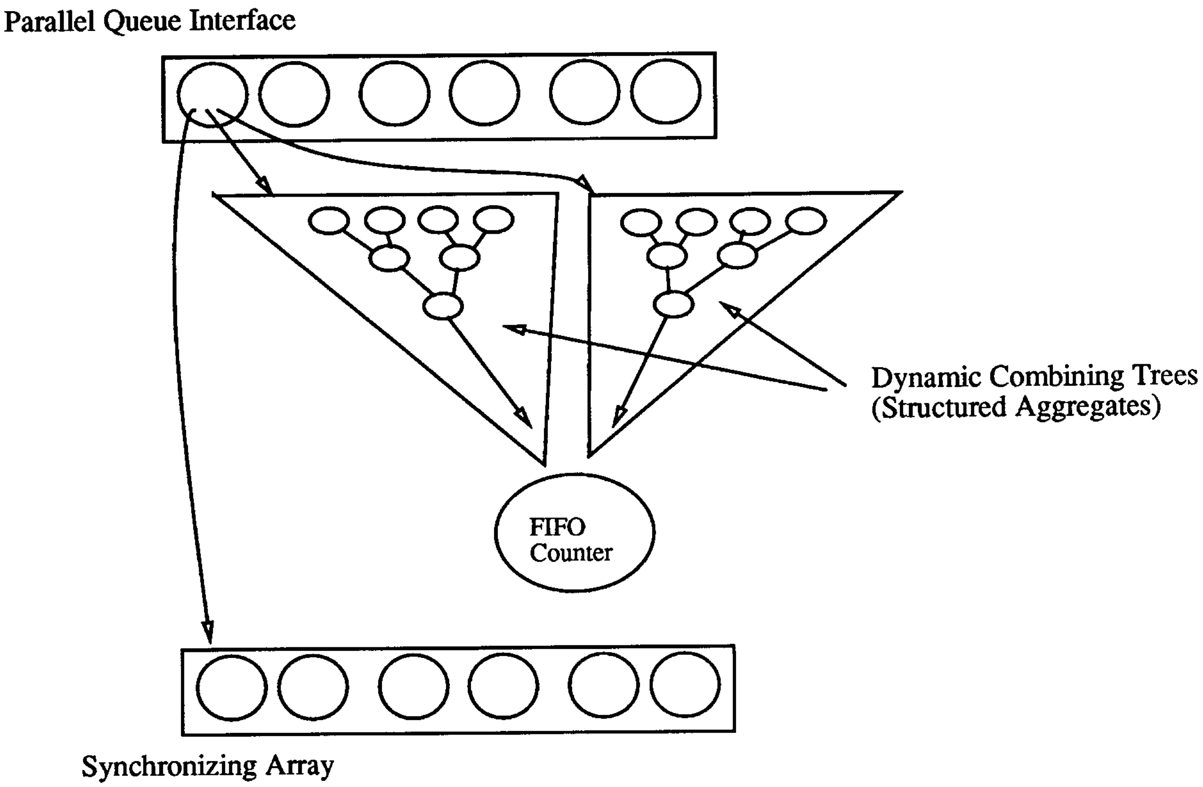


Figure 4.14: The Parallel Queue Abstraction

The array is "synchronizing" in the sense that it enforces the read-once-write-once protocol required for the array. Each location is read and written exactly once. The array must be synchronizing because the index allocation and data storage are two separate operations – allocation of a write index does not imply that the data is already stored there.

However, the array must be even more complicated because we want to reuse the locations. One consequence of this fact is that this synchronizing array requires four states for its "presence bits." The extra states serve to tell us when we can reuse locations. Some additional complexity is required to deal with the rounding (as the increment beyond the end of the array) the in and out pointers and reusing array locations. In order to implement the queue, it must be possible to distinguish the following states.

Synchronizing Array States

Full The location has been written but not read.

Waiting The location has been read, but not written. Location data bits contain the reader's continuation.

Empty-clean Neither a read nor a write has occurred since last reset.

Empty-dirty Both a read and a write have occurred since last reset.

The full and waiting states are similar to presence bits for memory in the HEP [90]. However, the empty-clean and empty-dirty states are unusual. They are required because in managing the array as a circular FIFO, we are actually using the array as a synchronization namespace between readers and writers. When the in and out pointers come back around, we reuse the synchronization names. Since messages may take an arbitrary amount of time in the system[7], we cannot be sure that the synchronization for a particular location has occurred – both the read and write may have not arrived. To safely reuse the storage, we must be able to distinguish the two states – empty-clean and empty-dirty and thus tell that a synchronization has occurred. When we reuse locations, we reset them to empty-clean state.

The parallel queue demonstrates a number of interesting uses of aggregates. The parallel interface, which simply holds pointers to the parts of the queue abstraction, uses the aggregate mechanisms to implement a number of replicas of the same state. This replication is used to avoid a bottleneck in accessing the queue. The queue interface provides the glue to put the abstraction together, and provide a coherent interface. The dynamic combining trees demonstrate another use of aggregates – a structured collection. Each representative is linked to other representatives in a pattern that forms a tree. The behavior of each representative node assures that the appropriate combining function is implemented by the overall aggregate. A third use of aggregates is demonstrated by the synchronizing array. The array partitions its state over the representatives. Each representative handles requests

[7] Eventual delivery is assured, but no particular time period is promised.

for a subset of the array indices. The synchronizing function can be performed locally by each representative. Requests arriving at the wrong representative are forwarded to the appropriate representative.

The queue interface aggregate allows us to access a queue uniformly, without regard for replication, size, and other implementation details. However, the efficiency of Concurrent Aggregates could be improved by sending messages directly to a particular subset of the aggregate. This would allow us to direct messages to the leaves of the combining tree only, improving its behavior to be more FIFO-like. It would also allow us to direct messages to particular representatives in the synchronizing array abstraction – avoiding any forwarding of requests. In Chapter 5, we investigate the performance impact of sending such messages directly – compiling the level of indirection out.

4.1.5 Printed Circuit Board Router

A printed circuit board router is used to route wires from one pin to another. In general, we would like to route wires (nets) along the shortest path. Routes must avoid obstacles on the board. A large board may include tens of thousands of nets. The PC board we are building for our prototype J-machine [79] has over 5,000 nets. A complex VLSI board might easily have 20,000 nets. One common approach is to route nets independently, and when congestion occurs (too many wires in a region may demand a board with too many layers), routes passing through a congested area are "ripped up" and rerouted [46, 88, 34, 99]. However, the cost function (originally simple distance) is changed to discourage routes from passing through the congested area. Our algorithm operates within this paradigm. For simplicity, we simulate only a single phase of routing. The routing process is depicted in Figure 4.15.

Our Concurrent Aggregates program uses the A* search technique [67] to find a path for each net. The obstacles are recorded in a single shared PC board "grid" which represents each point a net can pass through on the board. The grid can support massive concurrency – each point on the board can be polled independently to determine if is blocked or passable. Each net is routed independently and each net uses a priority queue to implement its A* search. Partial routes are prioritized by the distance traveled thus far plus an estimate (guaranteed to be a lower bound) on the remaining distance to be traveled. To route a net, we simply choose the partial route in the queue with the lowest priority and extend it. Priorities are assigned as the sum of the distance travelled thus far and an estimate of the remaining distance. Thus, choosing the lowest priority causes us to pursue a path that may be the shortest one. The new partial paths are inserted into the priority queue. Eventually, we will find the shortest path. Furthermore, we will typically have done much less work than an exhaustive search.

There are some complications – typically there are many paths to an intermediate point between start and destination as shown in Figure 4.16. Rather than extend each of the paths that reaches a point, we would like to merge the routes. If we pursue any path from a point of convergence, we only extend the merged path. Since we are only interested in

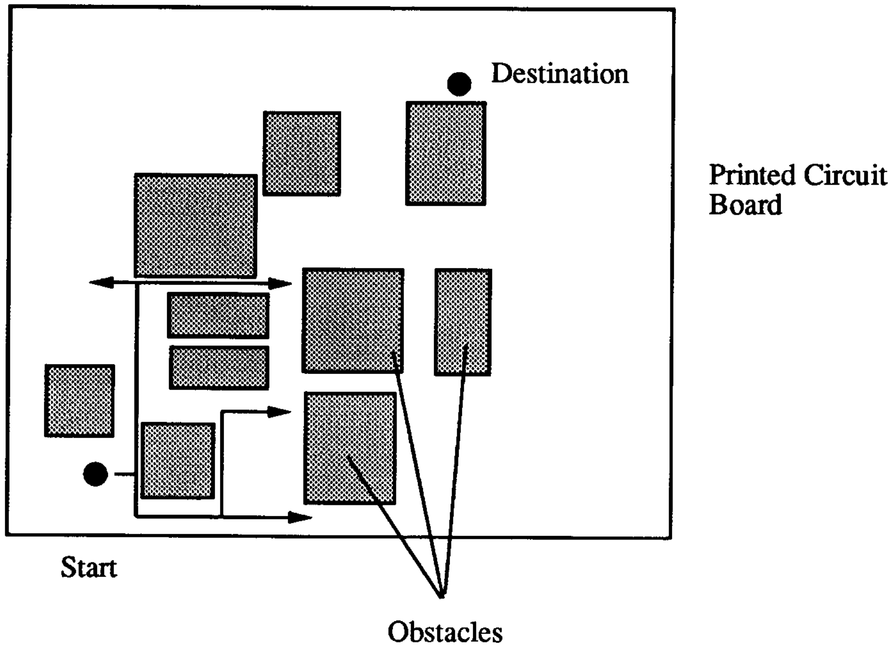


Figure 4.15: Using A* search to find a path

the shortest route, we merge by eliminating the longer routes at intermediate points. If the partial paths are the same length, we choose one arbitrarily. In order to merge paths, each net maintains an occupancy table which includes all of the points that have been reached, as well as the distance to reach that point. Of course, with sequential A* search, the first path to reach an intermediate point will always be the shortest (equal in length or shorter than all of the others).

We may not want to route 5,000 nets simultaneously, or we may want to make use of a massively concurrent machine to route a smaller board. We can do so by making the basic net routing operation, A* search, concurrent. Our approach to doing this is to pursue the *k* best queue entries simultaneously, instead of just one. This change has two implications. First, we will perform some extra computation by extending paths that would not normally have been extended. Second, this means that we may reach intermediate points (and the ultimate target) with suboptimal paths. To deal with this, we need to keep distances in the occupancy table. Shorter paths to intermediate points are allowed to supersede longer paths. This guarantees that we will ultimately find the shortest path, but we may do a significant amount of extra work. This approach allows us to expose approximately *k*-fold concurrency in the search for a single net.

Aggregates are used in a number of abstractions in our printed circuit board router. The PC board grid that contains obstacles that must be routed around is a partitioned abstraction – each representative holds the state for a small part of the board. As queries for

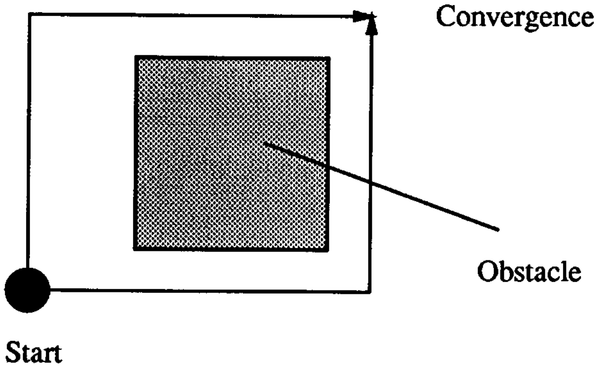


Figure 4.16: Two Convergent Paths

different parts of the board do not depend on each other, the representatives can collaborate to implement this abstraction without further message passing.

The net abstraction tops the hierarchy of abstractions as shown in Figure 4.17. In the PC board router, both the net and the occupancy table abstractions are implemented with aggregates. The net abstraction provides multiple access to the various parts of the net: the starting point, the end point, the occupancy table, the grid for the entire board and the priority queue for active paths. A multi-access net abstraction allows the concurrent pursuit of several paths for a single net. Using an aggregate for the occupancy table is also require to support simultaneous pursuit of several paths. The occupancy table is used to keep track of the shortest path for this net to a given point in the grid. This information allows us to eliminate the majority of redundant paths before they are inserted into the priority queue. The occupancy table (implemented by a hash table) is a good example of an abstraction that partitions its state and responsibility for handling requests over the representatives. The code for a hash table is shown in Figure 4.18.

For each hash table request, insert and member, the key is hashed to determine which representative is responsible for the request. Probe returns the list of <key, element> pairs where the key should be found. `Hash_pair` is used to implement the appropriate query and replacement operations on the for the `hash_table`.

If we wanted to pursue more than a few paths for each net simultaneously, serialization at the priority queue would limit performance. The priority queue is currently implemented with an ordinary object, and thus can only support limited concurrency. If the number of partial paths being pursued were increased, the priority queue would become the bottleneck, preventing any further increase in performance. The priority queue is pipelined, but can only accept messages at the rate of a single object, far less than what is possible with an aggregate. In order to significantly increase performance on a single net, it would be necessary to implement the priority queue with multiple access abstraction tools.

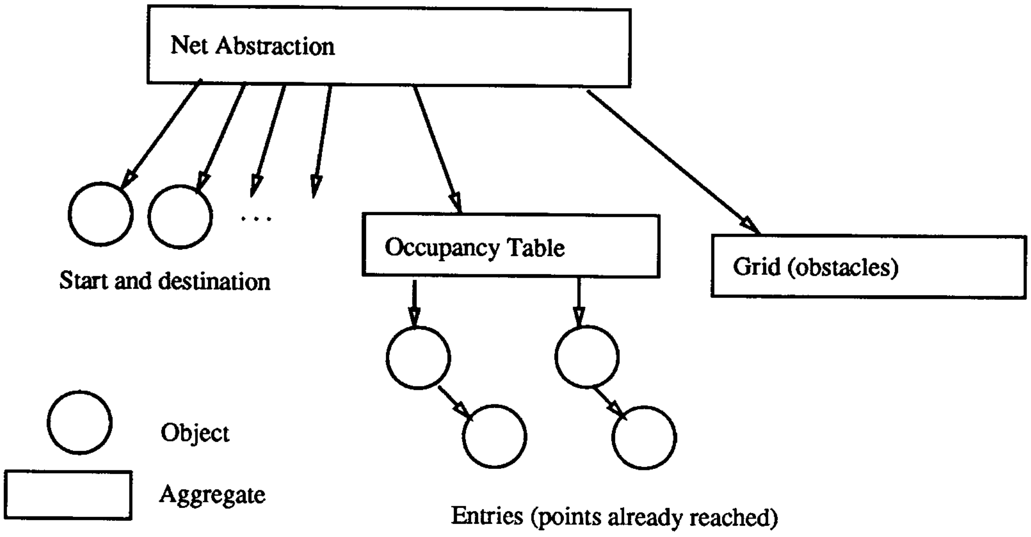


Figure 4.17: The Hierarchy of Abstractions used to implement the Net Abstraction

4.1.6 Concurrent B-tree

B-trees are used in many important programs [35] that require set abstractions with efficient insert, query and delete operations. Perhaps the most notable applications of B-trees involve their use in databases where the node sizes can be matched with page sizes in the virtual memory system. The granularity control available at nodes makes it possible to optimize the paging performance of a B-tree for very large trees. Our implementation of a concurrent B-tree was based on the algorithms of Lehman and Yao [68] and Lanin and Shasha [66]. These algorithms allow reads and inserts to lock only a constant number of nodes at a time. Deletes can require the locking of $O(log\ n)$ nodes at once. The basic structure of a B-tree is shown in Figure 4.19.

Generally, operations on the B-tree start at the top and work their way down to the bottom of the tree. QUERYs make no changes to the tree while UPDATEs may modify it. INSERTs may cause splitting of nodes at the bottom of the tree (expanding the tree downwards) and DELETEs may cause merging of nodes in the tree (an upwards traversal). Left-to-right links are maintained in the B-tree in order to assure correct behavior in the case of concurrent updates and queries are going on at the same time. When this happens, the query may find itself a few nodes too far to the left. The left-to-right links allow the query to move to the right until it finds the appropriate node.

Our concurrent B-tree makes interesting use of aggregates to avoid bottlenecks in concurrent access to data[8]. Aggregates are used for the `btree`, multi-version memory (a loosely

[8] The concurrent B-tree program was written by Paul Wang as part of his study of concurrent B-trees. A

```
;; Hash Table Abstraction
;; hash keys over the size of the aggregate and
;; then do linear search on the local list
;; (hash elt) must yield an integer for comparison with a key
;;
(aggregate hash_table local :no_reader_writer
        (parameters size)
        (initial size ...))

(handler hash_table insert (key elt cost)
        ... )

(handler hash_table member (key elt) :no_exclusion
         (let ((eltlist (probe group key)))
           (forward (find_key eltlist elt equal))))

;; probe returns a list from the appropriate bucket
;;
(handler hash_table probe (key)
         (let ((hash_index (mod key groupsize)))
           (forward (internal_probe (sibling group hash_index)))))

(handler hash_table internal_probe ()
         (reply (local self)))
;;
;; Hash pair abstraction
;;    Elements must have define "key" and "equal" operations
;;
(class hash_pair key elt next cost :no_reader_writer
        ...)

(method hash_pair find_key (t_key test_sel)
        (if (test_sel (key self) t_key) (reply (elt self))
          (if (neq 0 (next self))
              (forward (find_key (next self) t_key test_sel))
            (reply 0))))

;; for the end of list
(method integer find_elt (x y)
        (reply 0))

(method hash_pair replace_key (t_key t_elt test_sel newcost)
        ... this is used to insert new elements
            or new values for an element ...)
```

Figure 4.18: Hash Table Program

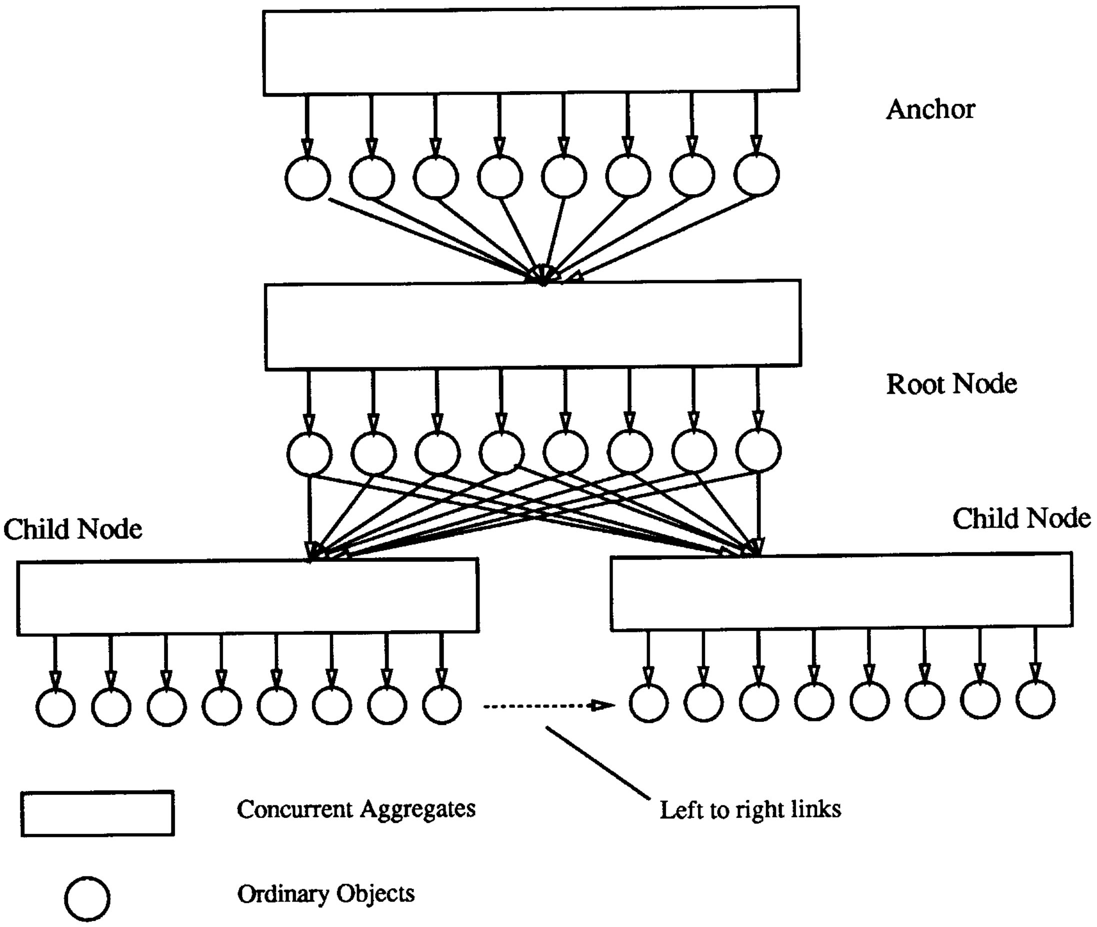


Figure 4.19: A Concurrent B-tree

consistent replicated memory), and regular memory (a consistent replicated memory) abstractions. btree is a multi-access abstraction that allows requests to access the b-tree concurrently. All requests to the B-tree come to the btree abstraction. Having a multi-access abstraction allows us to process them without serialization. We need this level of indirection in case a split propagates all the way back up to the root.

A multi-version memory [101] is a loosely-consistent, replicated memory. Loose consistency means that some read requests may receive stale data. Multi-version memories are used for the internal nodes of the B-tree. Multi-version memories can process multiple concurrent requests, due to replication. Replicas can be updated lazily – avoiding the overhead of locking all replicas simultaneously. Thus, the multi-version memory can continue to handle read requests while it is being updated. Read requests are handled by an arbitrary copy of the memory, and therefore may read old data. For the B-tree algorithm we used, reading old data from an internal node will not affect the correctness of the operation. However, it may make processing a request slightly less efficient (i.e. require it to traverse more tree nodes). The use of old data may cause the request to compensate by moving a few nodes to the right. This is no problem, as we keep all nodes linked left-to-right.

The third aggregate abstraction used in the concurrent B-tree is regular memory – a consistent, replicated memory abstraction. This abstraction is used for the leaf nodes which contain actual data elements. Read requests are processed by an arbitrary copy, allowing many reads to proceed concurrently. Writes to the regular memory must first lock all copies of the memory, update them, and finally unlock them. Of course, this locking prevents other requests from using the memory for a much longer period than with the multi-version memory. However in contrast to the multi-version memory, the regular memory always presents a consistent view of the memory. The locking overhead is only significant on writes, and since most uses of the B-tree are read only (i.e. simple queries), a larger overhead for writes is acceptable. Accelerating reads at some cost for writes improves the overall performance of the B-tree. For leaf nodes, we cannot use multi-version memory because a consistent view of the replicated memory is required.

In the concurrent B-tree, aggregates are used to support a two different kinds of replication. For the multi-version memory is a loosely-consistent replication scheme. The regular memory is a consistent replication scheme. For replication, aggregates structure is convenient because it allows the programmer to explicitly manage the level of consistency and tailor it to his program. In addition, the *one-to-one-of-many* direction of messages to representatives mapping is a good match for the uses of aggregates in the B-tree program as we do not care which representative handles requests.

more complete discussion of this program can be found in [100].

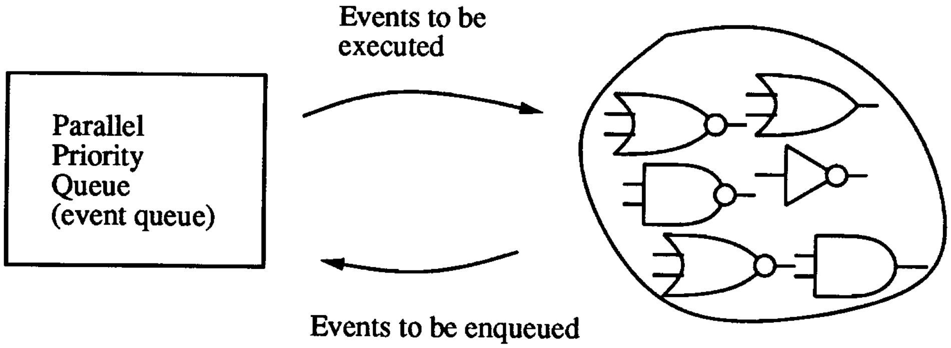


Figure 4.20: A Logic Simulator

4.1.7 Logic Simulator

Our logic simulator program simulates simple digital logic circuits. All timings in the circuits are discretized. The logic is simulated in an event-driven fashion [96, 95]. When a circuit node makes a transition, an event is queued. Events are evaluated for each time step, causing more gates to switch, causing more events to be scheduled in the future. In a large chip, it is not uncommon to have tens of thousands or hundreds of thousands of gates. It is important to do event-driven simulation because a typical percentage of active gates can ranges from 2 – 10% per discrete time step [22].

We implemented a multiple delay digital logic simulator in which transitions to 1 or 0 can take different amounts of time. Our logic simulator implements event cancellation [3], and could be easily extended to support oscillation detection. Our simulator also detects multiple events on a gate's inputs in one time step and fires the gate only once (eliminates redundant gate firings).

Our logic simulator consists of a parallel priority queue and a network of nodes and gates. The basic structure is depicted in Figure 4.20. Messages from the queue to the network involve the evaluation of events. Messages going in the opposite direction are used to schedule events. Events in our circuits have a finite minimum delay of several time steps, so it is possible to evaluate several time steps simultaneously. This allows us to increase the available concurrency.

Aggregates were used to implement a restricted parallel priority queue. We were able to avoid the cost of using a fully general priority queue by making the following observations about event priorities. First, our priorities, really time stamps, are monotonically increasing. Second, because events have a maximum delay, at any given time, all of the priorities in the queue are within a fixed size range. Based on these restrictions, we make the observation that the queue must manage events in a sliding window of priorities. This means that we can use statically allocated, indexable storage (e.g. arrays) to implement the priority queue. Because we are going to evaluate events from a number of different time steps

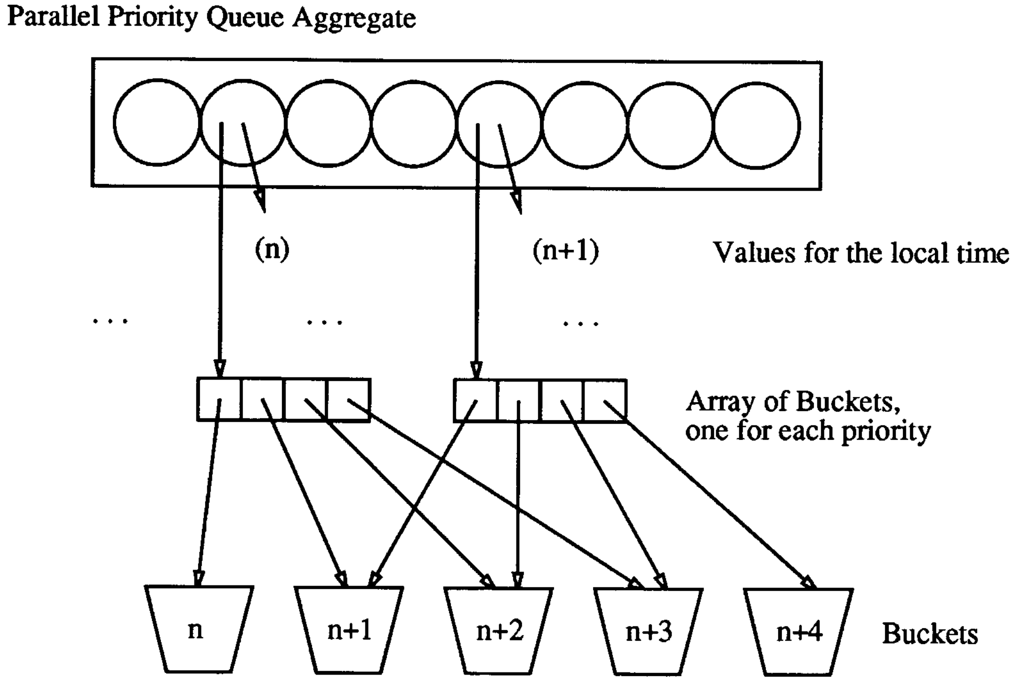


Figure 4.21: Local Consistency in a Parallel Priority Queue

simultaneously, we also require the users of the queue to have some notion of time. Enqueue operations take an event and a time. Dequeue operations must specify the time for which we want the events. The priority queue also incorporates the driver for our logic simulation. The structure of our priority queue implementation is depicted in Figure 4.21.

The parallel priority queue makes an interesting use of aggregates – replicated, local consistency. Each representative in the queue aggregate has an array which contains pointers to time step buckets. The correspondence between time steps and the pointers in the array is kept locally – each representative has a time variable. This allows each representative to adjust its array pointers and advance time in a decoupled fashion. In addition, representatives in the queue aggregate can process requests independently, allowing many requests to go to each bucket simultaneously. Time updates can be propagated gradually because enough information is maintained to keep each representative locally consistent. Figure 4.21 shows the local consistency in a parallel priority queue. The local times for the two representatives shown are n and $n+1$ respectively. This difference is reflected in the differing arrangements of pointers in their arrays of buckets. The pointers in the right representative's array of buckets have been advanced one step further than those of the left representative. Each bucket is labeled with its time.

The local consistency scheme allows us to overlap enqueue operations with advances of queue time. In order to advance the time in the priority queue, we tell each representative to update its local time variable and move the buckets up in its array *atomically*. No requests should be processed by a representative while it is advancing its time. We can continue to send requests to the queue while the update is going on because the other representatives can process them. At each representative, the advance of time is performed atomically and independently. Because the change is atomic and kept consistent with the local time information, no inconsistency will be visible to users of the abstraction.

Our queue has a "bucket" of events for each time step. We carefully designed the queue aggregate so that many requests could arrive at each bucket concurrently. In order to exploit that concurrency, the buckets must support concurrent access. Each bucket is implemented with a combining tree aggregate. As enqueue requests are received, they are linked into a dynamic broadcast tree as shown in Figure 4.22. Thus, an event becomes part of one such tree when it is entered into the event queue. When the simulation is ready to compute all events for a time step, it sends a message through the appropriate broadcast tree – firing all of the events attached to the tree. Each node in a dynamic combining tree combines requests and links the events into small broadcast trees. In turn, the consolidated request and the broadcast are forwarded up the tree. The requests are further consolidated and the small broadcast trees linked into larger ones. The code which performs this function is quite similar to the dynamic combining tree code shown in Figure 4.13.

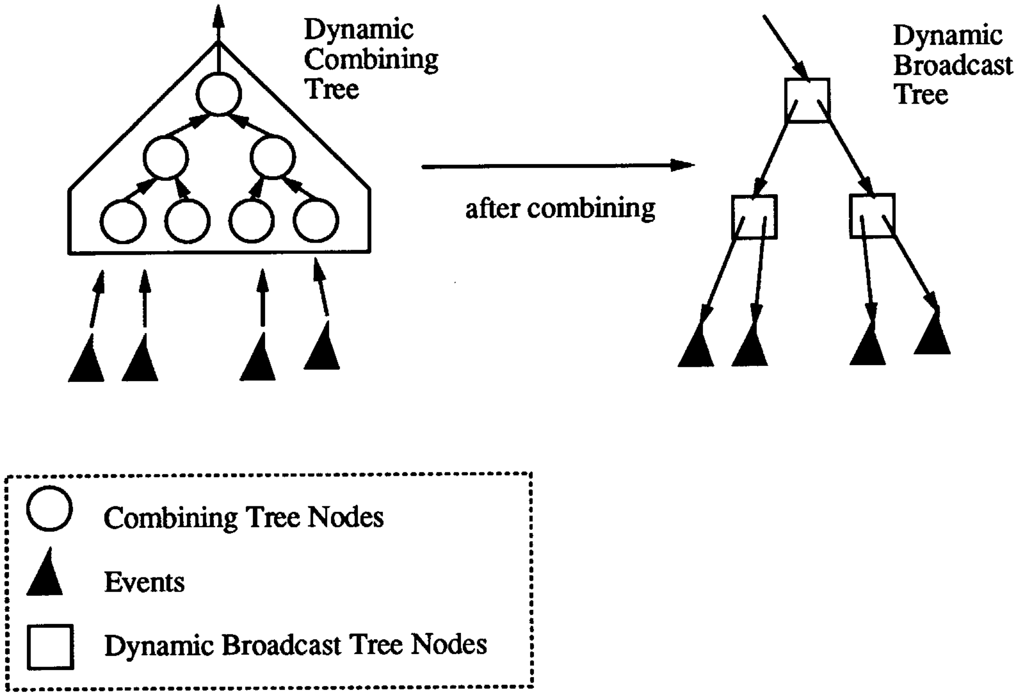


Figure 4.22: Construction of a Dynamic Broadcast Tree

4.2 Evaluation of the Language

Based on the application programs we have written in Concurrent Aggregates, we evaluate the language and its features. First, we evaluate the use of multi-access data abstraction tools in CA programs and find them to be very useful. Second, we present measurements of program concurrency under a number of different assumptions. Finally, we consider the efficiency of CA. Compared to a shared memory program with similar concurrency, the CA multigrid program is of comparable efficiency. In Chapter 5, we show that this efficiency can be significantly improved with simple compiler optimizations.

4.2.1 Non-serializing data abstractions

The multi-access abstraction tools provided in Concurrent Aggregates were used to construct a variety of non-serializing abstractions. These abstractions allowed hierarchical structuring and complexity hiding in the application programs without causing serialization. The power of these tools stems from giving the programmer explicit control over distribution, consistency and update of state. Such control allowed programmers to build efficient implementations of traditional abstractions as well as some novel abstractions. In this section, we describe a number of these different paradigms for using multi-access abstractions.

Replicated State Each representative in an aggregate can be used to hold a replica of the abstraction state. Read requests (non-mutating) can be handled locally by any representative. Write requests must lock all representatives, perform the write, and then propagate the new state to each representative before unlocking it[9]. If writes do not occur frequently, the replication will increase the effective bandwidth of the abstracation. For read-only (immutable) objects, aggregates can effectively increase their bandwidth. No mutations are ever performed, so there is no locking overhead. The queue interface aggregate (parallel FIFO queue application) and net (PC board router application) aggregate are two examples of read-only replication. If there are many more reads than writes, replication can be used to increase the effective bandwidth of mutable state. One example of replicated mutable state is the regular memory abstraction (concurrent B-tree application).

Loosely-Consistent Replicated State As in the previous case, each representative in an aggregate is used to hold a replica of the abstraction state. However, these copies are not kept completely consistent. Updates are propagated gradually. Use of loosely consistent replication can provide many of the benefits of replication – higher availability and throughput – with lower cost updates. But, some read requests may get stale data, so

[9] This use of aggregates is very similar to caching of data in a shared memory machine [89, 25]. Object caching schemes with similar functionality have been used in distributed systems [17, 15].

it is only useful when a consistent view of the state is not required. One scheme for loosely-consistent replicated state is multi-version memories [101]. In a multi-version memory, a request may be a read, read_newest, or a write. Reads may get stale data, read_newest and write requests are serialized by a single copy of the data. In the B-tree example, we used multi-version memories for internal tree nodes because 1) using the most recent information was not essential for most operations and 2) high bandwidth is essential for internal nodes to allow many B-tree requests to proceed concurrently. Multi-version memories may work better than replicated memories because the overhead for updating internal nodes may be less.

We found another interesting example of loosely-consistent replication in our logic simulation program. The parallel priority queue is an aggregate in which each representative keeps an array of buckets. The queue is illustrated in Figure 4.21. There is one bucket for each time step, and a representative's array points to the buckets for time n, $n+1$, $n+2$, etc. Buckets can be accessed concurrently, so the queue can process a large number of requests simultaneously. However, moving forward one time step requires that the arrays in each representatives be updated. Doing this consistently requires synchronizing the entire aggregate. This is undesirable because synchronizing the entire aggregate causes the queue throughput to temporarily go to zero. To avoid this, we maintain a local time count in each representative. When we want to perform an update, we send an update message to each representative and proceed. The representatives each process the update eventually. In the interim, the "relative" time of each representative is captured by the local time, and queue request winds up in the correct bucket.

Partitioned State Aggregates can be used to implement abstractions with partitioned, non-interacting state. Many program abstractions consist of collections of non-interacting state – the elements in an array for instance. We refer to such abstractions as partitioned state abstractions. While requests to the abstraction need only access the data at a single representative, the abstraction still forms a useful program structuring. This structuring improves the modularity of programs. Consistency amongst the parts of the abstraction is not an issue as the state of each representative corresponds to a different part of the abstraction's state. Typical implementations of partitioned state abstractions divide the abstraction state evenly over the collection of representatives in an aggregate. Each representative is responsible for operations on its part of the state. If a representative receives a request to operate on part of the state, it handles the request. If the request requires operation on another representative's state, it forwards the message to the appropriate representative. Examples of partitioned state abstractions include: hash table (various applications), synchronizing array (parallel queue), grid (PC board routing and multigrid), concurrent bucket (top_grid in multigrid), bodies abstraction and interactions abstraction (N-body simulation).

Structured Cooperation A more complex use of aggregates involves a form of structured cooperation. An aggregate is organized into a network with a particular interconnection pattern. When requests are received, the interconnection pattern shapes the resulting computation. One example of this is a dynamic combining tree – variations of which were used for barrier synchronizations, buckets in priority queues, and index allocation in the parallel FIFO queue. In each of these cases, the representatives are initially structured into a tree. Using this interconnection structure, requests are combined and propagated up the tree. Another example of structured cooperation is a grid. Representatives of an aggregate can be linked into a 2-dimensional grid and thereafter refer to their neighbors by their north, south, east or west direction. This can result in simpler code – singularities and boundary conditions can be handled uniformly.

Using aggregates for structured cooperation is interesting because different representatives are not consistent. Representatives are linked together, each forming a different part of the overall abstraction. They are specialized by the interconnection. The behavior of the abstraction emerges from the interconnection of representatives.

4.2.2 Program Modularity

Aggregates allowed us to implement an abstraction barrier, at little cost in concurrency. As a non-serializing abstraction tool, using aggregates allowed programmers to structure their programs without concern for reducing concurrency. This improved the modularity of Concurrent Aggregates programs.

None of the abstractions described would have been practical to implement in an Actor language [2] because they would reduce concurrency dramatically[10]. One example of this is the grid abstraction in our multigrid solver which processed thousands of message simultaneously. Implementing it as a serializing abstraction would be unacceptable[11]. In fact, data parallelism over the grid points was the primary source of concurrency in multigrid. If all of our abstraction tools were serializing, even if the serialization was a single instruction, in many cases we would be forced to avoid using them. In a grid abstraction, a few instructions of serialization per message would cause many thousands of cycles of serialization for each iteration on the grid, drastically reducing concurrency.

The grid abstraction in the multigrid application not only has well defined interfaces to its upward grid and downward grid, it has a well defined interface to a synchronization abstraction, synch_relax. We have even modularized the synchronization structure of our program. As we explained in Section 4.1.2, the synch_relax abstraction could be replaced by any other appropriate synchronization abstraction, such as a barrier, that had a compatible interface. The grid abstraction would not need to be modified.

[10] In fairness to the Actor model, it was developed to model programming in distributed systems, not tightly coupled, fine-grain message passing machines.

[11] One way of reducing serialization due to an abstraction is to use caching or replication schemes. However, they are unlikely to help in this case as the grid points are written quite often (making coherence expensive) and the number of copies required to supports the thousands of simultaneous accesses would be quite large.

4.2.3 Program Concurrency

We simulated the application programs described on small data sets and measured the concurrency using our message-passing machine simulator. These results are indicative of how our application programs will behave on larger machines, using commensurately larger data sets. It is difficult to accurately extrapolate these results to larger machines, so we leave any extrapolation to larger machines to the reader.

Simulation Models Our vehicle for study is a message-driven simulator. This simulator can model a number of different machines, varying the cost of various machine and runtime system operations. In order to separate the constraints on program execution due to message passing dependences and bounded resources, we use two basic simulation models in our experiments: the *Idealized Model* and the *Bounded Resource Model.* **The time units for the two models are not the same, and therefore measurements made using the two models should not be compared directly.**

Idealized Message Passing Model The *Idealized Model* models an implementation of Concurrent Aggregates in which all communication occurs in unit time. Program execution is constrained only by message dependences (causality) and mutual exclusion on objects. Of course, such an implementation is unrealistic. However, the statistics from *Idealized Model* represent the basic constraints on concurrency in the program structure. From another perspective, this simulation model corresponds to a machine in which local computation is quite fast, and communication is performed synchronously[12].

In simulations using the *Idealized Model*, we present a number of different statistics. Each of these is defined below:

Critical Path The number of message passing operations on the path from beginning to end of the computation. This may include some overhead due to spin locking on objects. The time unit used here is "message sweeps." In each sweep, all messages are delivered and executed.

Total Messages Total Messages executed in performing the computation.

Peak Message Concurrency The maximum number of messages executed in any message sweep. This can be loosely interpreted as the largest number of processors it would be useful to have in executing the program. This measure is quite sensitive to simulation details.

[12] All nodes communicate at once and then compute until they have no more work to do. Then all of the nodes communicate again. The communication occurs in unit time. No time is charged for the computation. This model is quite similar to the model presented by the Connection Machine, except our nodes are truly MIMD and computation is not infinitely fast on the CM [93].

Application	Data Size	% size	Crit. Path	# msgs	Peak Conc	Avg Conc
Matrix Mult	4096 elts	≈ 25%	541	1,880,041	8070	3,475
Multigrid	4096 pts	≈ 25%	1,642	2,894,816	7029	1,763
N-body	64 bodies	<1%	20,576	2,020,304	2748	98
PCB Router	8n 4096gp	< 1%	40,351	1,271,812	75	32
B-tree	10K ops	10%	34,995	1,054,941	106	30
Logic Sim.	3,584 gates	≈ 2%	35,267	1,512,461	306	43

Figure 4.23: Program Statistics using the Idealized Message Passing Model

Average Message Concurrency The average number of messages executed per sweep, the total messages divided by the critical path length.

Our simulation results for the Idealized Message Passing Model are presented in Figure 4.23. Most of the statistics presented are as defined above. We present results for each of the application programs described in Section 4.1.

Due to limitations of our simulation approach, we only ran CA programs on modest-sized data sets. The size of the data sets and the approximate relation of these data sets to "real size" problems are both presented in the table. The matrix multiply computation was for multiplication of two 64 by 64 matrices. This is a modest size matrix multiplication and only 25% of the work required to do a more realistic 128 by 128 matrix multiplication. The Multigrid application involved a 64 by 64 grid at the finest level, again, roughly 25% of the work required for a 128 by 128 grid typical in the Particle-in-Cell code [73]. The N-body simulation was only for a tiny number of bodies, 64. Many simulations for molecular dynamics often include 10,000 to 100,000 bodies and other researchers have run simulations with millions of bodies [108, 52]. The printed circuit board router example connected 8 nets across a board grid of 64 by 64 tracks. In practice, boards are much larger and have 5,000 to 20,000 nets. The distance nets must be routed also effects the amount of work, and varies greatly.

The B-tree simulation involved a tree with 1,000 keys. With a maximum fan-out of 10 per tree node, the depth was approximately 3-5. For the benchmark, we built a tree of 1,000 elements and then used 30 "workers" to perform 10,000 operations against the tree. The effective concurrency we measured is quite good as it approaches the number of workers. The operations were 80% QUERYs, 15% INSERTs, and 5% deletes. They were generated in that mix using system random number generators. The logic simulation example involved a synthetic circuit with 16 32-bit counters. All told the counters included 3,584 gates. In the Concurrent VLSI Architecture Group at MIT, we are currently building a moderately complex chip (the Message-Driven Processor [42, 41]) which contains approximately 50,000 gates, not counting the on chip memory arrays. In other systems the gate count may run into the hundreds of thousands. This collection of gates had an average of 14 events per simulated time step. The average activity level ($< 0.5\%$ gates active per time step) is low compared to that in many simulations. The counters show little activity as the high order bits rarely change. Activity levels typically range from 2 – 10% [22]. A circuit with 50,000 gates and a 5% activity level would have plenty of concurrency to keep a machine of several thousand processors busy.

Our results show that Concurrent Aggregates programs can exhibit massive concurrency. The concurrency figures in Figure 4.23 are probably not realizable – communication latency and resource contention are sure to reduce concurrency. However, these figures give us an upper bound on the achievable performance on these programs. To provide a more accurate estimate of the performance we would expect, we also performed simulations under a bounded resource model.

Bounded Resource Message Passing Model In the *Bounded Resource Model*, we model the finite processing resources of an actual machine. Resource limitations affect the evolution of the computation. For example, if several objects are resident on a single processing node, only one of those objects may be active at a time. This in turn will delay the responses to messages arriving at that node as they are queued until the processor becomes free to serve them. This, in turn, delays dependent parts of the computation.

It is quite difficult to model the detailed cost of operations in a real machine. Not only is such detailed cost accounting expensive, it is not yet clear exactly which operations a fine grain message-passing machine should be optimized to support. However, it seems important to model contention for resources, as such contention can significantly change the behavior of a computation. Our solution to this problem is to use a very simple approximate machine model. Every local operation takes unit time. Local operations include primitive operations (add, sub, mult, etc.), context switches, function calls, method invocations, local object allocation, object location resolution, and aggregate to representative resolution. Communicating a message from one part of the machine to another takes a fixed latency of one time unit. This overcharges for the simpler local operations (like addition and subtraction), but is quite close to the costs in our J-machine for the other local operations. This computation to communication cost ratio corresponds roughly to the realities of our J-machine prototype[13].

The simple cost model reduces simulation complexity, making it possible to simulate larger problems. The *Bounded Resource Model* simulations reflect a machine of 4096 nodes. While we expect much larger machines to be constructed (16K-64K nodes), 4096 processors is a good match for the problem sizes we could simulate. Typically, our application programs had from 10k to 100K objects. The processor resource limit may reduce the average concurrency by truncating the peaks and smearing them out over time. All objects are placed randomly on nodes and do not migrate.

[13] This reflects the assumption that the network is relatively lightly loaded. Network contention is not modeled in the simulation.

Application	Data Size	% size	Crit. Path	# msgs	Peak Conc	Avg Conc
Matrix Mult	4096 elts	≈ 25%	5,922	1,880,055	3,595	2,098
Multigrid	4096 pts	≈ 25%	20,377	2,616,544	1,959	811
N-body	64 bodies	<1%	71,134	2,130,878	747	117
PCB Router	8n 4096gp	<1%	136,796	1,283,174	63	30
B-tree	10K ops	10%	209,805	984,610	64	29
Logic Sim.	3,584 gates	≈ 2%	265,416	1,056,782	122	14

Figure 4.24: Program Statistics using the Bounded Resource Message Passing Model

In our simulation results using the bounded resource machine, we present a number of different statistics. Each of these is defined below.

Critical Path The number of time steps from the beginning to the end of the compuation. The time unit here is defined above.

Total Messages Total Messages executed in performing the computation. This number differs from the count in the Idealized Model due to different rates of spinning and waiting periods on spin locks.

Peak Concurrency The maximum number of processors busy in any time step of the computation. It is influenced (and bounded) by the number of processors in the simulation, 4096. This statistic can be quite sensitive to simulation details.

Average Concurrency The average number of processors busy per time step.

Our simulation results for the Bounded Resource Message Passing Model are presented in Figure 4.24. We used the same application codes and data sets as with the Idealized Model simulations. As expected, the program concurrency shown in this simulation model is reduced from that of the idealized model. However even with resource restriction, the amount of concurrency can be very large.

We found that the reduction in concurrency is attributable to several causes – contention for objects, contention for processing resources on a node, and communication latency. Contention for objects, not modeled in the idealized model, reflects the structure of the program. The number of messages that must be processed in a particular phase of the computation is determined by the algorithm. Contention for node processing resources causes performance loss attributable to the runtime system. In choosing where to place objects, a runtime system would like to avoid placing simultaneously active objects on the same node. Practically, some of this is unavoidable, and if we have abundant concurrency, such node contention should only be significant if it happens in regimes of low concurrency. Communication latency reduces concurrency due to the distribution of a program throughout the machine. When a runtime system chooses placement for data in the system, there is a direct tradeoff between avoiding node contention and attempting to minimize communication latency.

For realistic data sets, the programs we have considered are likely to exhibit massive amounts of concurrency. In fact, the amount of concurrency should be enough to utilize machines with thousands of processors. From our simulation results and structural analysis of the programs, we expect that in the absence of machine constraints, the concurrency in all of the applications we have considered should scale in proportion to the data set size[14]. This is a good sign for builders of massively concurrent message passing machines.

4.2.4 Program Efficiency

Program efficiency is important because speeding up unnecessary work is not productive. It is difficult to evaluate the efficiency of a programming language in a manner independent of compiler details, architectural quirks, and circuit technology. The situation is even more difficult in our case because efficiency usually involves a comparison to some baseline language or architecture. We are constructing programs for a machine with massive processor concurrency, and most well established baselines involve moderately concurrent machines[15].

We considered a number of metrics for the cost of a program including total run time, total instructions executed, and total messages. We eliminated total run time as it is the most closely tied to architectural details and even specific implementations of the architecture. We also eliminated total instructions executed as a metric because it does not accurately reflect the cost of a computation on a concurrent machine. In a sequential von Neumann machine, the number of instructions executed is directly related to the total run time. However, in a concurrent machine with thousands of instruction execution engines (processors), the number of instructions and run time are not so simply related.

As communication is the one resource that becomes more critical as we build finer and finer grained machines, we have chosen total message count as our metric of program efficiency[16]. As we scale fine-grain machines to larger and larger numbers of nodes, they will ultimately be communication limited[17]. The cross section can only grow as $n^{\frac{2}{3}}$, while the number of nodes is n. Thus, the message traffic is a good measure of the efficiency of a programming system and algorithm[18]. We consider the efficiency of Concurrent Aggregates

[14] For the B-tree example, it may be more complicated. The concurrency grows with the tree size and replication factor for each node. The effective concurrency depends heavily on the mix of operations on the tree.

[15] The machines with comparable amounts of concurrency are the CM-2 [93] and the NCUBE/2 [78] machines. While the CM-2 has massive concurrency (64,000), it is difficult to compare our results to it, as the CM-2 presents a SIMD programming model. The NCUBE/2 can be configured as large as 8K processors, but programming experience with such large systems is minimal.

[16] Of course this metric is subject to optimization by the compiler. Better compilation, or simply compilation that favors larger grains will reduce this number. Favoring larger grains will typically reduce concurrency. To avoid this pitfall, we consider message counts in program formulations with approximately the same program granularity.

[17] This is because we must build machines in 3-space. We need to pack things closely to minimize propagation delay, so the volume of our machine is proportional to n, the number of processors. The bisection of a 3D densely packed machine is $n^{\frac{2}{3}}$ and limits the bandwidth for random traffic.

[18] We do not consider any exploitation of locality through caching or clever mapping of data onto the ma-

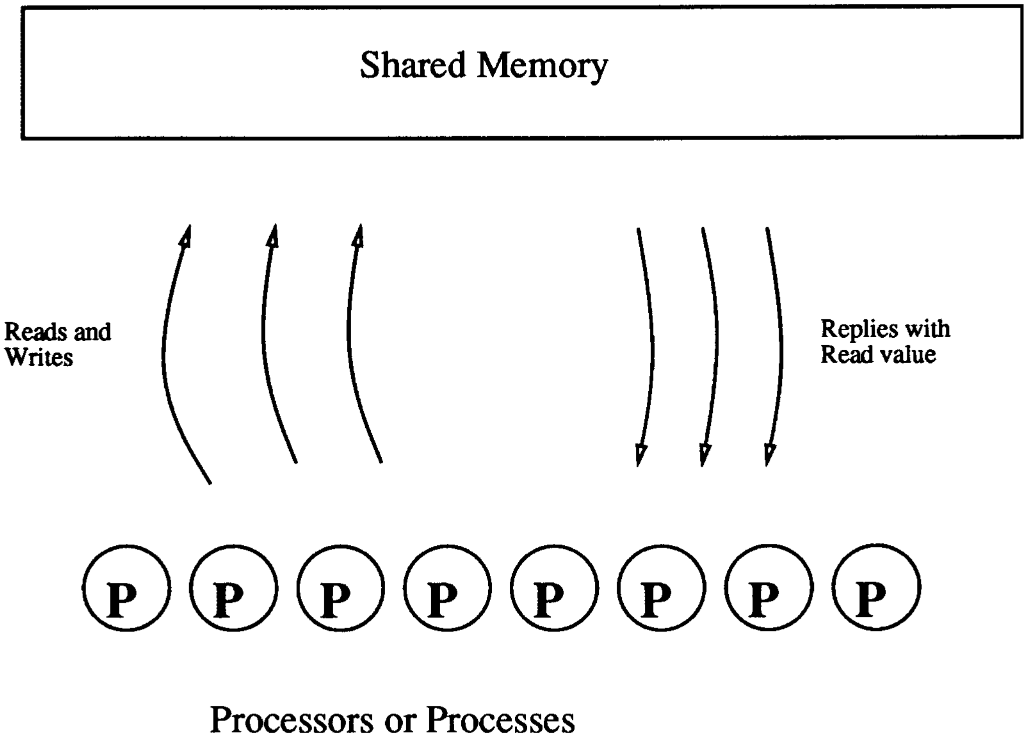


Figure 4.25: A Simple Shared-memory Model

by comparing the message traffic required to implement the multigrid algorithm in CA to that required to implement a multigrid algorithm with comparable concurrency on a shared memory machine. We find that the message traffic required for the CA multigrid program is close to that required for bare memory accesses in a shared memory model. Increasing the bare memory accesses to the full shared memory execution cost and allowing for optimization of the CA program would bring the numbers even closer.

Message Count for the Shared Memory Model We consider the multigrid algorithm described in Section 4.1.2 on a simple shared memory model, depicted in Figure 4.25. In our shared memory model, each read takes two messages (request, then returning data), and each write takes one message. Our accounting of messages is conservative because we do not account for the message traffic for program control – synchronization and process management. Also, we only charge one message for a write, while many systems may require a second message for a write acknowledgement. In this algorithm, we use a three-level grid structure. For each level of the grid, we perform eight relaxation steps on both the way up

chine. In some cases, these techniques can dramatically reduce the communication required. However, they depend on many language, compiler, runtime and machine issues. In order to reach a basic understanding, we consider message count as the metric.

and down the hierarchy. At each relaxation step, we use a five point stencil for the relaxation step. Upward restriction is done with a single point, while downward prolongation is done by spreading the one value over four points.

Ignoring boundaries and assuming a 64 by 64 grid and a three level grid structure (4096 grid points in the largest grid, 256 in the smallest), we get the following message counts. We first compute the number of messages for the iterations within each grid level, then add in the number of messages required to link the grids to each other. For each iteration, the five point stencil requires 5 read and one write for each grid point. For the 16 total iterations on the upward and downward pass, the number of messages required is shown below:

```
messages / 5 point stencil = (5 reads * 2 messages/read)
                                   + (1 write * 1 message/write)

messages per grid point = nr-iters * messages/stencil

                        = 16 * 11 = 176
```

The total number of grid points is the sum of the three levels. For example, the total number of messages for the relaxation operations on a 64 by 64 grid is:

```
relaxation messages = 176 * (4096 + 1024 + 256)

                    =   946,176 messages
```

Of course, to form the overall multigrid algorithm, we must communicate values between the grids. This is done with restriction and prolongation operators. Our restriction (upward linkage) operator takes a single value for every four grid points and injects it into the upper grid. Our prolongation operator takes one data value and divides it over four grid points in the lower grid.

Grid Points	Shared Memory	CA I	CA II
1024	239,424	686,964	345,303
4096	957,696	2,581,233	1,353,579

Figure 4.26: Comparison of Message Counts for Multigrid

The accounting for the messages required here is shown below.

```
upward messages = nr_target_gridpoints *
(4096 to 1024)          (1 read  + 1 write)
                = 1024 * (2+1) = 3072
upward messages = nr_target_gridpoints *
(1024 to 256)           (1 read  + 1 write)
                = 256 * (2+1) = 768

downward messages = nr_source_gridpoints *
(256 to 1024)           (1 read + 4 writes)
                  = 256 * (2 + 4) = 1536

downward messages = nr_source_gridpoints *
(1024 to 4096)          (1 read + 4 writes)
                  = 1024 * (2 + 4) = 6144

total linkage messages  = 3072 + 768 + 1536 + 6144
                        = 11,520
```

This means that over 11,520 messages are required for upward and downward coupling in the multigrid algorithm. By summing the relaxation messages and the linkage messages, we get our approximation for the message traffic required for multigrid.

```
total messages  = relaxation messages + linkage messages
                = 946,176 + 11,520
                = 957,696
```

We compare this number to our actual measured numbers for the Concurrent Aggregates language in Figure 4.26. We note that the measured traffic for CA program includes whatever message traffic is required for initialization and synchronization as well as some process management overhead. The numbers for the shared memory model do not include a significant amount of control information that must be transmitted. In addition, the statistics presented are taken from our first implementation of Concurrent Aggregates. The measured numbers from our working CA implementation are shown in the column labeled

CA I. As we will see in Chapter 5, we are already aware of a number of crucial efficiency issues in the implementation of CA. We expect that some minor language changes and better compilation will significantly improve the message efficiency of programs. The numbers of messages which would be produced by a program that made use of some simple optimizations presented in Chapter 5 are shown in the column CA II. The traffic for the shared memory machine is probably somewhat understated as we have only considered data reference traffic. To support a comparable level of concurrency, the control traffic would have to be quite significant. With the optimizations used to calculate the performance for CA II, we are optimistic that the CA program will be ultimately be of comparable efficiency.

4.2.5 Evaluation of other interesting language features

Intra-Aggregate Addressing An aggregate is a cooperating collection of objects. In order to cooperate, it is often convenient to be able to access the names of the other parts of the aggregate. The intra-aggregate addressing facility was used extensively. The representative indices were used to compute state partitionings and object interconnection. For example, the representative indices were used to determine which representative handled which part of the state in the synchronizing array abstraction. In combining tree abstractions, the indices determine the intra-aggregate interconnection to form the tree structure. Because it is used so pervasively, cheap implementation of aggregate name operations is essential to implementing Concurrent Aggregates efficiently.

First Class Continuations and User Continuations Continuations were used in many of our application programs. The ability to explicitly manipulate continuations made it possible to construct synchronizing structures such as futures, a synchronizing array and a barrer synchronization within the CA language. Without this ability, more restrictive control synchronization would probably have to be applied in these programs. User constructed continuations found use in fewer places – a barrier synchronization, fanning out replies in a combining tree, and a race construct – to support speculative concurrency. While allowing the user to manage continuations explicitly was convenient at times, the use-once characteristic of system continuations caused quite a number of subtle bugs in programs. Double reply cases are difficult to detect and if activation frame names are being reused, may cause a very strange behavior. The extra reply may have come from any activation that handled the continuation, not only the one we called directly. This often makes finding its source quite difficult.

First Class Messages Allowing programmers to manipulate messages as first class objects turned out to be a useful feature. First class messages were used as partial applications – factoring the details of a partial application from the code that manipulates the application. For example, we constructed several varieties of fan out trees that implemented do-all operations on each representative of an aggregate. We also used first class messages to construct message reordering abstractions. For instance, we built a message queue abstraction

– used to defer the processing of messages to a later time. A variant of this message queue was used in the top_grid abstraction from the multigrid application. First class message manipulation in Concurrent Aggregates can also be used to explicitly change the order in which messages are processed. It could even be used for a message-ordering system – to implement point-to-point order-preserving message transmission.

By-Message Delegation Our CA programs did not make much use of delegation. We had hoped that delegation would allow us to compose behaviors incrementally – piecing together the desired behavior and message interface for an abstraction. However, we did not use it often because for most of our abstractions, the subparts needed to cooperate quite closely. Abstractions not designed as a subpart in general did not include the appropriate code for cooperation. This experience may be due to the type of program we examined, our limited experience, or perhaps the way we chose to integrate delegation into the Concurrent Aggregates language. It may be the case that delegation will become more important as we construct larger and larger programs.

4.3 Summary

In this chapter, we have presented an evaluation of Concurrent Aggregates. We evaluated the language by writing a number of application programs and executing them. The application programs studied are a matrix multiplier, a multigrid solver, an N-body interaction simulation, a PC board router, a scalable parallel queue, concurrent B-tree, and a logic simulator. The application programs involve a wide variety of different algorithms and program structures. For each application, we first described the algorithm and then the basic program structure in Concurrent Aggregates.

We evaluated the primary innovation in the language – non-serializing data abstraction tools – based on our experience writing the application programs and numerous other small programs. Our experience with these tools was quite positive. The non-serializing data abstraction tools make it easier (or possible) to build modular programs. Programmers are free to use abstractions wherever they improve program structure. In CA, adding a level of abstraction need not reduce program concurrency. In addition, allowing programmers to explicitly manage distribution and consistency enabled them to tailor the level of consistency to the task at hand. Programmers came up with a number of interesting uses of our aggregates. Some of these are familiar – read-only replication and consistent replication. Others were quite novel – partitioned state, loosely consistent replication, and structured cooperation. We described each of these schemes and gave several examples.

We also evaluated Concurrent Aggregates programs with respect to their concurrency and efficiency. Though we could only simulate modest-sized data sets, our simulations showed that for a number of applications, CA programs can exhibit massive concurrency. For some programs, that concurrency should be sufficient to utilize a machine with thousands of processors. To evaluate the efficiency of CA programs, we considered a case study that compared the message traffic of a CA multigrid program to a shared memory implementation. Our comparison showed that our unoptimized CA program required 2-3 times the messages required for the shared memory version. However, the optimized CA program was quite close to the shared memory program.

In light of our programming experience with Concurrent Aggregates, we also evaluated other novel language features: intra-aggregate addressing, first class and user continuations, first class messages, and by-message delegation. Intra-aggregate addressing has emerged as a key feature for building aggregates. Without it, constructing a coherent abstraction interface would be quite difficult. First class and user continuations were used in many places in our programs. The use-once nature of system continuations was problematic at times, but we still feel that the efficiency gain of avoiding full garbage collection of activation frames justifies this sacrifice. First class messages turned out to be a very useful way of implementing partial applications. These partial applications were most often used to implement data parallel operations. The last feature we evaluated, by-message delegation, did not fare as well as the others. We found little use for it, as abstractions not designed for cooperation in the current context were often not useful – abstractions did not compose well.

In Chapter 5, we take up implementation issues in Concurrent Aggregates. Specifically, we will consider the cost of sending messages to aggregates, *one-to-one-of-many* translation and intra-aggregate addressing. Our programs used these features heavily, so their implementation is a key issue in the overall language implementation. We will also discuss some compiler optimizations which would allow us to implement CA more efficiently.

Chapter 5

Implementation Issues

Efficient implementation is an important issue in the design of programming languages. In this chapter, we examine the support – hardware and software – required for an efficient implementation of Concurrent Aggregates. We begin by considering the support required for concurrent object-oriented languages with fine-grained mobility (examples include Concurrent Aggregates, Emerald [19, 18], CST [58, 57, 38] and Amber [27]). Subsequently, we discuss and evaluate the cost of implementing the novel features of CA. These include first class continuations and first class messages. In addition, aggregates require two additional services of the operating system – *one-to-one-of-many* translation and intra-aggregate addressing. We consider how often these features are used and a number of different schemes for supporting them. As efficiency is an important concern, and our implementation is simple (unoptimized), we also examine compiler optimizations. Using dynamic run statistics and manual analysis of the source code, we estimate the performance improvement possible with these optimizations.

5.1 Basic Run time Support

A run time system for Concurrent Aggregates must contain a number of basic services. These services are listed and described below. We refer to these services as basic because they are required by many "reactive" message-passing languages. In this respect they are not unique to Concurrent Aggregates.

Message Transmission In order to compute, objects must send messages to each other. The system must assure reliable transmission of messages from one object to another. For efficient support of fine-grain computations, the network must be low-latency and high bandwidth.

Object Location The system must support the association of an object name with its storage. This may be a two-step process involving determining the appropriate node as well as the location within that node. In systems with no relocation, this association may be completely static.

Storage Allocation and Reclamation A run time system must manage storage for messages and contexts (activation records) as well as user objects. In practice, the allocation and deallocation of storage for messages and contexts must be very efficient. Our design of first class continuations takes this into account, and hence context allocation and reclamation can be done efficiently. Reclaiming storage for user objects involves a garbage collection problem.

Reactive Invocation In response to a message, the system must invoke a piece of user code – a method. To support fine-grain concurrency, this invocation process should be as rapid as possible.

Efficient support for the basic services has been studied extensively, so it is not considered in detail here. A variety of hardware and software approaches have been taken to support basic runtime services. We refer the interested reader to the literature on these various topics: message transmission [40, 39, 43, 45, 103], Object Location [94, 58, 62, 71], and reactive invocation [41, 84]. Efficient storage allocation and reclamation in fine-grained message-passing is a very active research area. Some work in this area is described in [7, 58, 50]. Many systems handle storage for messages and activation frames specially, allowing them to be handled and reclaimed efficiently. However, efficient reclamation of object storage is a general concurrent garbage collection problem. A detailed discussion of these topics is beyond the scope of this thesis.

5.2 Supporting First Class Continuations and Messages

Concurrent Aggregates allows both continuations and messages to be treated as first class objects. Continuations may be copied, stored, and used (replied to). Messages may be copied, modified, and resent. References to messages can be stored. While these capabilities facilitate programming with aggregates, they render invalid traditional assumptions in storage and name management. In this section, we discuss implementation issues for these two features in detail.

5.2.1 First Class Continuations

First class continuations can be used in many ways, but our primary intention was to allow more efficient composition of abstractions. In both sequential and concurrent programs, first class continuations are often used to separate the call and return structure of a program. In sequential programs, the motivation for such separation is usually program clarity. For example, exception handling facilities or co-routines can be implemented with first class continuations. In Concurrent Aggregates, the separation of control structure possible with first class continuations can be used to improve program efficiency. Allowing programmers to manipulate continuations allows them to express some concurrent control idioms more efficiently. Our motivation for first class continuations in CA is discussed in greater detail in Section 3.4.6.

First Class Continuations in Sequential Languages In sequential languages, activation records[1] are usually stack allocated. This allows inexpensive allocation and deallocation of activation records. First class continuations in a language such as Scheme [81] allow a program to call a particular execution point of the program after that place has returned control. This is useful for implementing advanced control structures such as co-routines. However, the existence of references to activation records requires some activation records to persist after they have returned control. Such activation records cannot be deallocated according to stack discipline. With first class continuations, references to the activation record may persist and thus, it may not be safe to reclaim the record yet. To incorrectly deallocate the frame might cause a "dangling reference" error. Persistence of activation records can dramatically increase the cost of allocating and deallocating stack frames.

Implementations of first class continuations on sequential machines need not reduce the basic efficiency of the programming language. Empirically, first class continuations are rarely used. Thus, optimistic schemes (try for best case and trap the others) have been quite effective for Scheme programs [32]. In this situation, the best case is when no references to the activation record are created. Then, it can be stack allocated and deallocated. Some run time overhead may be required in a few cases to determine whether a reference is created. The expensive case, of course, is when a reference escapes. These situations can be detected via a mix of compile and run time techniques. When an escaping reference is detected, the system heapifies the activation records as necessary (copies into the heap). Once activation records have been moved to the heap, they are collected by the more expensive garbage collection mechanism. Since this does not happen often, the cost of activation record allocation and deallocation is quite small on average.

First Class Continuations in a Concurrent Language In a programming language with concurrent invocations, activation frames in a computation form a tree. Invocation, return, and computation may proceed in all parts of the tree simultaneously. The activation frames can no longer be managed as a stack, so a more general means must be employed. Typically, the storage for activation frames is explicitly managed using a free list. The free list may be partitioned into one list for each node. The activation tree can be extended and contracted locally, without any global actions. If references to records do not escape, the allocation and deallocation can be done in a manner analogous to stack discipline. This scenario is depicted in Figure 5.1.

[1] We use the terms activation record and activation frame interchangeably.

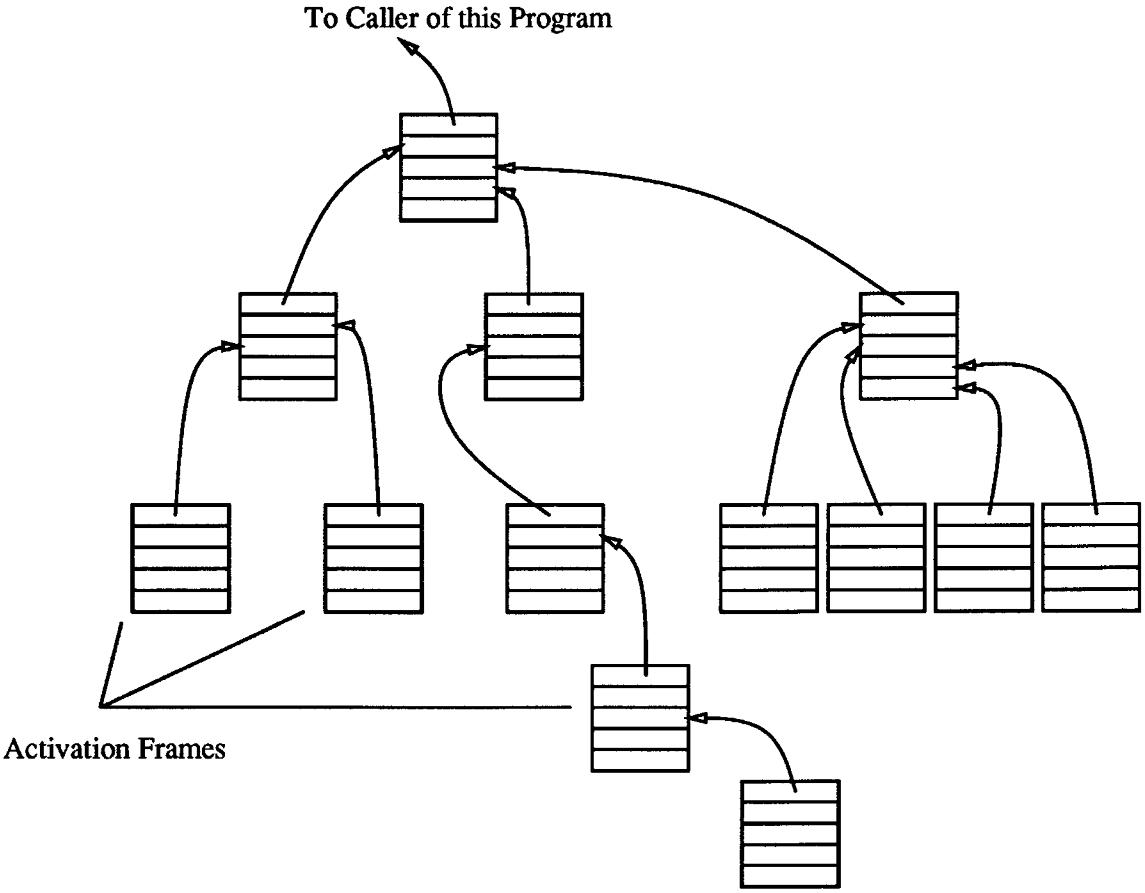


Figure 5.1: A Tree of Concurrent Activation Frames. All of the frames shown may be active concurrently. All arrows shown are return links.

Introducing first class continuations complicates the picture. It becomes unclear when an activation frame can be reclaimed. If a reference to a frame can persist in a stored continuation, then it is unsafe to reclaim the storage for an activation frame, and transitively, all of its ancestors in the call tree.

Traditional first class continuations in a concurrent environment complicate reasoning about program execution. If a continuation can be invoked more than once, then the two uses may occur concurrently. This means that two threads may be active within a single activation record at once. One of the reasons for pursuing an object-oriented approach is that the programmer can be freed from worrying about arbitrary interleavings of low level operations such as reads and writes against a shared memory. The object-oriented model allows the programmer to deal with concurrency at a higher level of abstraction, simplifying his task. However, if we allow multiple uses of a continuation, there is the potential for multiple threads of computation in a single activation record, forcing the programmer back to dealing with arbitrary instruction interleavings. This problem arises because our concurrency control is done on the basis of activation frames (i.e. a frame holds the lock to an object) instead of activations themselves. We point out that two replies to the same continuation is quite different from having two outstanding continuations for a context. The latter case may happen whenever, the programmer uses a `concurrent` construct. However, this concurrency is explicit and confined, as the end of a `concurrent` construct is an implicit join.

In Concurrent Aggregates, we chose a cheaper kind of first class continuation – disposable continuations. Programmers are free to replicate references to the continuations, but they must assure that only one reference is ever used. After use, the continuation ceases to exist. Multiple uses of a system continuation give unpredictable behavior. Programs that make multiple use of a system continuation are considered to be invalid. The uses we have found for first class continuations did not involve the reuse of continuations (calling a continuation multiple times). And, while somewhat unconventional, disposable continuations are easy to implement efficiently. Deciding when to reclaim an activation frame is straightforward. Each activation frame is scanned before it is collected. If all continuations created for the frame have been used, we can reclaim the activation record. Otherwise, the activation record must remain until all of its continuations have been used. If some of the continuations are never used, the activation frame must be reclaimed by the garbage collector.

User continuations in Concurrent Aggregates imply no special execution overhead, just normal garbage collection. User continuations allow `reply` messages to be handled by user abstractions. The type-dependent dispatch mechanism needed for languages such as CA can be used to decide which code should handle any `reply` message. If the object is a system continuation, then the system reply handler is invoked. If it is a user object, then the appropriate user code is invoked.

5.2.2 First Class Messages

First class messages allow programmers to build meta-programs, programs that control the evolution of a program execution. We have used first class messages to build message queues, fan out trees and other useful message handling abstractions as user programs. In this section, we discuss our design motivation for first class messages. We also examine implementation issues in supporting them.

Concurrent Aggregates incorporates messages as first class objects with a few restrictions (for a more detailed description, see Chapter 3). Messages can be stored, modified, and sent[2]. Messages are by-value parameters [77]. This means that they are copied when used as an argument to an invocation. A by-value convention for messages conveniently assures that messages are local for most message operations. In cases where a partial application is being replicated for application to a parallel data collection, the implicit copying does just the right thing to the message by default.

First class messages are useful as *partial applications.* For example, messages can be manipulated as applications whose arguments can be modified or whose time of execution is to be controlled. Examples of the former type include fan out trees and message routing abstractions. Examples of the latter type include message reordering queues and various synchronization structures, such as a barrier.

We chose messages over some variety of closures for partial applications. This is because messages are a simpler type of application that allows a programmer to control locality. Messages are by-value, so they are typically local. In addition with messages as applications, operations to manipulate and invoke applications need not require communication. Further, replication of messages does not require any special analysis of programs. Use of closures would require analysis to determine when copying of closures is allowed. On top of that, the run time system would still have to make good decisions about where to place closures and when to move them in order to produce good performance. With messages none of this is necessary, the programmer can control the locality.

Our parameter passing convention for message references (by-value) was designed to support use of messages as partial applications efficiently. We designed it to support fan out for data-parallel programs and storage of partial applications for meta-programs. For fan out operations, as shown in Figure 5.2, at each stage of the tree, the message should be replicated. This produces the appropriate number of message copies at the leaves of the fan out tree. In fact, it is necessary to replicate the message (partial application) at each level in order to preserve the logarithmic time property of the fan out operation. The copying of messages allows the partial application to have enough bandwidth at the leaves to support a massively concurrent data parallel operation.

Copying at message invocation boundaries is a correct implementation of these by-value semantics. Such an implementation allows locality to be improved – we can copy

[2] This is the analogue to call in a procedure-oriented language.

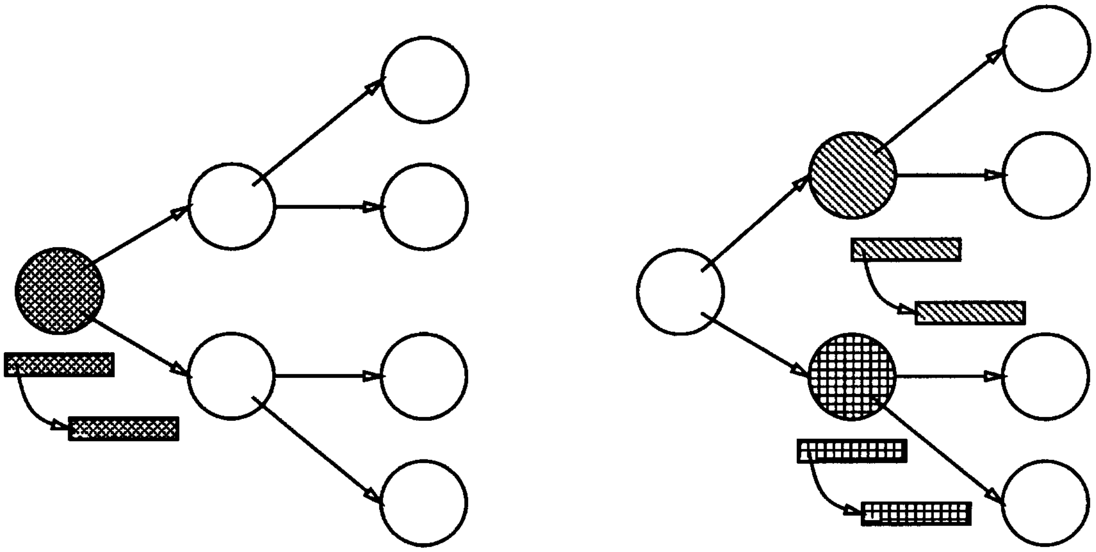


Figure 5.2: Fanning out a Partial application for a Data Parallel Operation: Each tree node is on a separate processor. Objects with the same shading are on the same node.

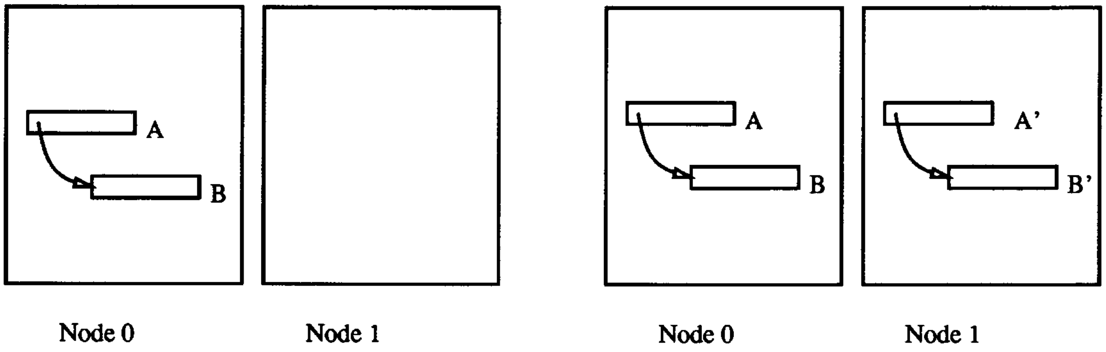


Figure 5.3: Copying maintains Message References as local.

the messages to the node that contains references to them. This optimization of locality through copying is shown in Figure 5.3. When a message, A, containing a reference to another message, B, is transmitted from Node 0 to Node 1, the result is a new message, A', on Node 1 with a local copy of B, B'. Messages A and B on Node 0 may continue to exist or be reclaimed. By placing the new copies of messages in the right places, we can assure that most operations on messages are local. Such operations include modification, copying, and calling (sending the message). This is a pleasant outcome as messages are the elements of communication, so to avoid recursive requirements for communication, operations on messages must be local.

Another use of first class messages is in the construction of meta-programming abstractions. For example, consider a message-queue abstraction that receives messages and upon demand resends them. This could be used to control the unfolding of concurrency in a program in response to some measures of machine loading. Figure 5.4 shows a message queue abstraction that is built from a singly-linked list. The list is built out of **mpair** objects and the **mqueue** object implements the message queue interface. **push** takes a message and enqueues it. **resend** transmits all the messages in the queue.

One drawback of a by-value convention for messages is that unnecessary copying of messages may result. The operation of the code shown in Figure 5.4 causes a potentially avoidable message copy operation. In the second line of the **push** method, the message reference transmitted in the invocation results in a message copy operation. This copying operation assures that the new message copy is colocated with the message pair. Later, when the message is resent, this colocation allows the message to be resent without any extraneous message passing. If the message being stored is quite large, the copy operation may be much larger than the savings due to the colocation of the pair and message copy. We felt that most messages would be relatively small and deemed these extra copying operations an acceptable cost. These intuition is born out by our measurements in Table 5.1.

Another implication of by-value messages is that it becomes difficult for computations to share state through messages. When a message reference is sent from one object to another, an implicit copy operation occurs, preventing message state sharing. The by-value semantics effectively preclude state sharing between caller and callee through a message reference parameter. This is not a serious restriction on programmers as any other objects can be used to implement the desired state sharing.

Implementation While programming with first class messages can be very convenient, their implementation presents a number of interesting challenges. Messages are used pervasively, so their implementation efficiency has a crucial impact on overall language efficiency. Potential expenses of first class messages involve the allocation and deallocation of names, allocation and deallocation of storage and message locality (avoiding the necessity of recursive implementations of message operations[3]). In order to understand the impact of our

[3] Sending a message to perform a message operation may require a message operation. This in turn may require sending a message which may in turn require a message operation and so on recursively.

```
;;  A cons object
(global last_mpair (new mpair 0 0))  ;; list terminator

(class mpair left right
       (parameters ileft iright)
       (initial (set_left self ileft)
                (set_right self iright)))

(method mpair resend ()
        (do (send (left self) (global console)))
        (if (neq (global last_mpair) (right self))
            (do (resend (right self)))))
;;
;; A deferral queue
;;
(class mqueue head mpref
       (initial (set_head self (global last_mpair))
                (set_mpref self (global last_mpair))))

(method mqueue push (mess)
        (seq (set_head self (new mpair mess (head self)))
             (reply (head self))))

(method mqueue resend ()
        (if (eq (head self) (mpref self))
            (reply nothing_to_resend)
          (seq (do (resend (head self)))
               (set_head self (mpref self))
               (reply resending))))
```

Figure 5.4: Code for A Message Queue

Application	FC Msgs	Avg Msg Size	Total Msgs	FC Msg %
Matrix Mult	0	N.A.	1,880,055	0.00%
Multigrid	131	6.93	2,616,544	< 0.01%
N-body	8,572	5.96	2,130,878	0.40%
PCB Router	2	5.00	1,283,174	< 0.01%
Logic Sim.	16,824	4.00	1,056,782	1.59%
Total	25,529		8,967,433	0.28%

Table 5.1: Usage of First Class Messages

changes to message semantics, we first consider how message handling is optimized in existing systems. Subsequently we discuss each of the implementation problems and present our compromise solution.

First class messages jeopardize many efficiency enhancements traditionally used to accelerate message processing. These enhancements include using local names, allocating storage from a FIFO buffer (temporary names), and lazy copying into the heap. These techniques have been used in systems such as the J-machine [42] and the Reactive Kernel [84]. If messages are first class, they need names and persistent storage. If they may be shared or accessed remotely, they need global names. Allocation and reclamation of such names for each message might well be a very expensive operation. Persistent storage would seem to require a more expensive scheme for storage allocation.

First class message parameters exist only in a small percentage of the messages in Concurrent Aggregates programs. In Table 5.1, we present the counts of first class message uses and their sizes. For some perspective, the total number of messages sent in these applications is also presented. First class messages are not used that often. The usage rate ranged from 0% to 1.6%. This is encouraging because if the first class message feature is used rarely then it may be possible to construct an "optimistic" implementation. We term it optimistic because it only incurs the cost of the full generality when it's really required, not all the time. Typically , optimistic schemes work by assuming the best case and trapping to handle the worst. In addition, by-value messages allows the system to avoid allocating global names for most first class messages – local names suffice.

Most messages have no explicit references to them. They are implicitly composed at the sending side and destructured at the receiving end. If the data shown in Table 5.1 is indicative of CA programs in general, programs create explicit references to relatively few messages. On this basis, our implementation optimizes for the common case – messages that are not explicitly referenced.

When a message arrives, storage is allocated for it in a region of memory managed as a circular FIFO buffer. Messages are processed in the order they are received, so many of the messages are discarded at this stage and never make it out of the FIFO buffer. If a message need persist beyond the first invocation, it must be copied into the heap. Such messages are only allocated local names. The by-value semantics for messages assures that references to these messages are local.

Our implementation attempts to keep message references local. When a message reference is transmitted, the message must be copied to assure that when the message reference arrives, it will point to a local message. To assure this, a trap occurs when a program attempts to transmit a message reference. The trap handler copies the message parameter into the end of the transmitted message. On the receiving end, message parameters are unpacked and referenced as local objects.

In a more efficient implementation, it would be possible to reduce the overhead due to these message copying traps. For example, sometimes it is possible to detect message references at compile time, allowing copy code to be compiled inline and avoiding the run time cost of an exception. We can easily determine when references to messages are created or "escape." A programmer can obtain a reference to a message by using the `MSG` pseudo-variable to refer the current message or the **message** construct to create a new message. With local data flow information, we can annotate the sends to do the appropriate copying and creation of a message reference. However, unless we can infer types for the entire program or require some type declarations, we will not be able to eliminate all traps to copy by-value messages. If message copy traps have a significant performance impact on programs, then we would consider a statically-typed dialect of Concurrent Aggregates that would allow us to reduce the cost of message copying. But, given the levels of use shown in Table 5.1, it seems unlikely they will have significant performance impact.

5.3 Supporting Aggregates

If aggregates are used pervasively in programs, they must be implemented efficiently. The additional requirements of aggregates are the *one-to-one-of-many* translation and intra-aggregate addressing. In this section, we discuss several different implementations and discuss their advantages and disadvantages. We also study the cost of a number of these different approaches through simulation on traces derived from our application programs.

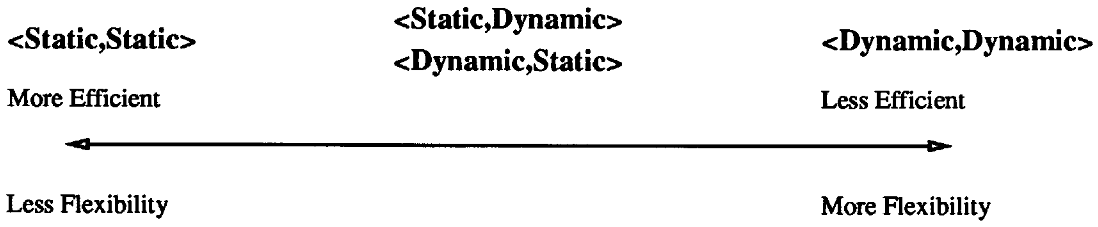


Figure 5.5: Spectrum of Aggregate Naming Implementations

5.3.1 A Spectrum of Implementations

We consider a series of implementations of the aggregate naming operations: interface translation and intra-aggregate addressing. Both of these functions require the mapping from an aggregate name to some storage in the machine. We break down the translation process into two steps[4].

- Aggregate name to representative name translation. This is required for both *one-to-one-of-many* translation used to implement aggregate interfaces. It is also required for intra-aggregate addressing.
- Representative name to object storage.

We focus on the first translation step which is unique to aggregates. The second translation can be viewed as a simple object location problem. This type of service must be provided by the run time system to support our basic object oriented programming model (see Section 5.1.).

We parameterize our implementations of aggregate name translation in the flexibility of the mappings for each translation step. These implementations can be seen as part of a spectrum as shown in Figure 5.5. At one end of the spectrum, the **Static,Static** scheme provides the least flexibility in placement and association. The first **Static** indicates that the mapping from an aggregate name to its representatives is fixed. The second **Static** indicates that representatives map to fixed machine nodes. The **Static,Static** scheme can be implemented very efficiently as computation of relationships between aggregate and representative names does not require any communication.

At the other end of the spectrum, other schemes may allow the runtime system and a Concurrent Aggregates programmer more freedom. For example, the **Dynamic,Dynamic**

[4] In an actual machine implementation, for efficiency one might want to merge them into a single translation step. An example where this merging is done in conventional computer systems is a virtually addressed cache. The virtual to physical address and physical address to data translations are merged into a single translation step in the cache.

scheme could support aggregates of arbitrarily named or placed objects. Objects could be collected in aggregates long after their creation, rather than being created as a part of them.[5] These less restrictive schemes appear to be much more expensive to implement. We describe these schemes in more detail below. In particular, we evaluate the flexibility gained and the cost of implementation.

Static,Static A fixed mapping from aggregate name to representative names. Representatives are fixed to machine storage according to their names. This scheme allows for static collections and would be well matched to a system lacking both dynamic allocation and object relocation.

Static,Dynamic A fixed mapping from aggregate name to representative names. However, each representative is a relocatable object, and can be moved around the system to allow for load balancing.

Dynamic,Dynamic A dynamic mapping scheme between aggregate name to representatives allows the runtime system to avoid fragmentation problems in the namespace. It also admits the possibility of an object participating in multiple aggregates[6].

Dynamic,Static A weaker version of the above Dynamic,Dynamic scheme that might be appropriate for a system without object relocation.

A static aggregate interface mapping can be implemented efficiently with no communication and fixed cost. The static mapping from aggregate name to representative names allows *one-to-one-of-many* translation to be done locally. No information other than the aggregate name is needed, so no communication is required. The static mapping also allows indexing, intra-aggregate addressing, to be done without any communication. A dynamic mapping from aggregate name to representatives allows both the runtime system and the programmer more freedom. The runtime system need not allocate representative names in a manner that allows them to be algorithmically derived from the aggregate name. This avoids fragmentation problems in the object namespace.

The mapping of representatives to machine nodes is an orthogonal issue to the aggregate interface mapping issue. The static or dynamic nature of the mapping between representative names and the machine nodes is primarily a reflection of the underlying system that is supporting aggregates. If the system does not support efficient object migration, then a static scheme may be the only workable one. If aggregates are being used to support data parallelism, then a static scheme may be a good one. A placement with uniform spreading assures that data parallel operations can go with optimal concurrency. However, typically

[5] This creation of objects as part of an aggregate is the model we currently support in Concurrent Aggregates.

[6] Another way of achieving this, as noted by Bill Weihl, is to simply allow multiple names for an object. In such a scheme, an object would have a name for each aggregate in which it was participating. This "aliasing" of the object buries the first-level resolution in the second-level translation. One disadvantage of this alternative is that things like an eq test become more difficult to implement.

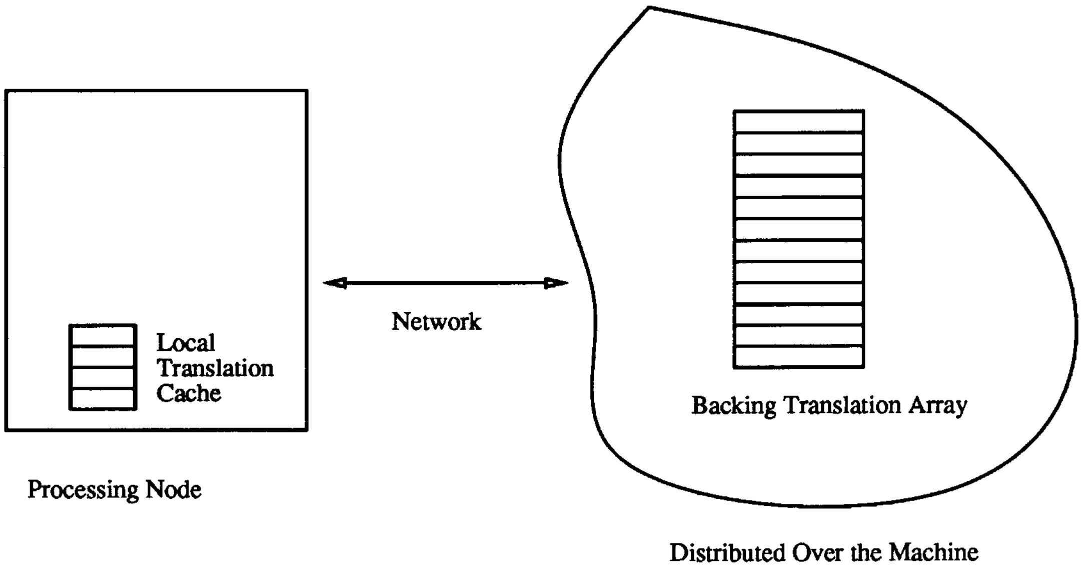


Figure 5.6: A Generic Scheme for Supporting Dynamic Translation

a static mapping means that only one type or a few types of mapping are supported. As we have seen, aggregates can be used in many different ways. In different cases, a different placement may be considered "good." A dynamic scheme for mapping representatives to nodes allows placement to be tuned to the particular aggregate or to the current machine situation: loading, node failures, network partitions and resource sharing. Object location is not a problem novel to aggregates, so we do not focus on it here. The interested reader is referred to [94, 58, 62].

5.3.2 Examining Aggregate to Representative Translation

Our simulator implements a static mapping between the aggregate group name and the names of the representatives. It is assumed that the names of all representatives in an aggregate are encoded in its name. Thus translation from the aggregate group name to the name of one of the representatives, *one-to-one-of-many* translation, can be done locally. These assumptions are reflected in the simulation results shown in all other parts of the thesis. For each *one-to-one-of-many* translation, we charge a constant amount to decode the representative name from the aggregate name. The heuristic used to select a representative is to choose any one of the representatives randomly.

Experimental Design In order to investigate the cost of supporting a dynamic translation from aggregate to representative, we collected traces of translation requests from our application runs. These traces contained all translation requests for aggregates, including both one-to-one-of-many or intra-aggregate addressing translations. In order to support

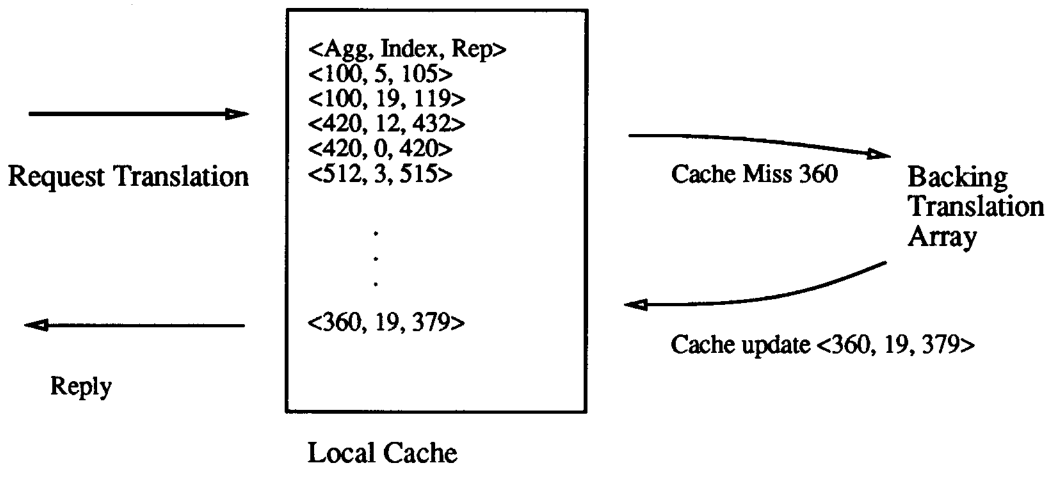


Figure 5.7: Updating a Cache for an Aggregate Interface Miss

dynamic translation efficiently, we assumed that the implementation would take the form shown in Figure 5.6. There is a cache for each node, so if a translation hits in the cache, no communication is required. Changes to members of the aggregate may require invalidation of some cache entries.

The aggregate to representatives translation is held in a backing array which may or may not allow concurrent access. For large aggregates, we would probably want to support concurrent access. Each node that accesses the aggregate may cache some information about the aggregate, allowing some translation requests to be handled without accessing the backing array.

To examine the potential for accelerating aggregate to representative translations, we simulated an ideal cache for the translation at each node. This ideal cache keeps all translations it has ever seen. Matches in the cache can be used to avoid message traffic for later translations. Matching in the cache is fully associative. We use the information in these local caches in the following manner. The information in the cache consists of 3-tuples. Each tuple contains the name of the aggregate, the name of a representative in the aggregate, and the index number of the representative in that aggregate.

The main limitation of this approach is the lack of feedback into evolution of the program execution. By running our cache simulator on traces, we can estimate the amount of work required to do translation (cache hits, misses, etc.), but we cannot determine how this would affect the actual run time of the computation. The best that we can do is to estimate the additional amount of work.

Application	Total Msgs	Interface	% Misses
Matrix Mult	1,880,055	528,448	0.7%
Multigrid	2,616,544	790,408	0.5%
N-body	2,130,878	97,150	6.4%
PCB Router	1,283,174	69,212	13.9%
Logic Sim.	1,056,782	24,013	3.3%
Total	8,967,433	1,509,231	1.6%

Table 5.2: Aggregate Interface Cache Simulations

For *one-to-one-of-many* translations, the node extracts all of the tuples with matching aggregate name and chooses one of them arbitrarily. If there are no matching tuples, a request is made to the backing array and that value is used for the translation. The node also places this new translation in its local cache.

In order to support indexing (intra-aggregate addressing), tuples are selected on the basis of aggregate name and index. If the tuple is not present in the cache, an access is made against the backing array and response is used to service the request and to update the cache. A translation miss and cache update is depicted in Figure 5.7.

Aggregate Interface Translation Caching We ran our translation traces against this ideal cache design to determine whether caches were likely to be helpful in implementing dynamic aggregate to representative translation. If the hit rates were high enough, caches might be able to accelerate the translation, decrease its latency, and decrease its cost (fewer messages required). The results for aggregate interface or *one-to-one-of-many* translation are shown in Table 5.2. For each application kernel, the interface column indicates the number of *one-to-one-of-many* translation requests made for the entire program execution. The % Misses column shows the miss rate for caches. As we might have anticipated, it is quite low. Intuitively, once a single tuple exists in the cache for a given aggregate, all interface translations for that aggregate will hit in the cache. Our results indicate that caching may be effective in implementing dynamic translation from aggregate names to representative names.

Caching of the aggregate interface translation may decrease its randomness. Rather than repeatedly selecting random representatives from the aggregate, all subsequent requests from a particular node will be sent to the same representative. For programs in which many of the requests for an aggregate originate from one or a few nodes, this may cause reduced throughput for the aggregate. For programs where the requests come from many nodes, this should not be a problem.

Intra-aggregate Addressing Translation Caching We also studied the performance of the ideal cache on indexing accesses (intra-aggregate addressing). These results are presented in Table 5.3. In the intra-aggregate column, we indicate the total number of

Application	Total Msgs	Intra-aggregate	% Misses
Matrix Mult	1,880,055	544,768	98.6%
Multigrid	2,616,544	811,873	96.8%
N-body	2,130,878	69,504	53.0%
PCB Router	1,283,174	30,721	97.1%
Logic Sim.	1,056,782	9,790	17.5%
Total	8,967,433	1,466,656	94.8%

Table 5.3: Intra-Aggregate Addressing (indexing) Cache Simulations

Application	Total Msgs	Aggregates	Tuples/Aggregate
Matrix Mult	1,880,055	3.0	44.0
Multigrid	2,616,544	3.4	57.4
N-body	2,130,878	2.1	5.0
PCB Router	1,283,174	3.6	2.6
Logic Sim.	1,056,782	1.6	2.1

Table 5.4: Cache Sizes for Aggregate Naming Translations

indexed translation requests for the application program. % Misses contains the percentage of translation requests that missed in the cache. It seems clear that the caches were not effective for indexed references. In some cases, the hit rates were low enough that the caches were totally ineffective.

The low cache hit rates for indexing translations may be due to the randomization of the interface translation. We expected that caching might be effective for indexing translations if there were some higher frequency communication patterns in the computation. If these patterns involved intra-aggregate addressing, the repetition of these patterns would cause hits in the translation caches. However as all messages to aggregates are sent to a random representative, we suspect that this randomization is destroying the translation locality. Any other aggregate interface translation scheme that results in more repeated identical translations would produce more cache hits. We also suspect that the aggregate interface optimizations described below in Section 5.4.1 will improve cache performance for indexing translations. Compiling interface code into the users of aggregates avoids the interface randomization step. In the context of this thesis and CA implementation, we must say that our caching scheme was ineffective for indexing translations.

Cache Sizes We are interested in using finite-sized, even relatively small, caches, so we also measured the maximum sizes that our ideal caches reached. These numbers are presented in Table 5.4. We collected two statistics for the translation cache sizes, the number of different aggregates for which there are tuples and the number of tuples per aggregate in the cache. The product of these two numbers is the number of tuples in the node cache. For three of our applications, the N-body simulation, PC board router, and Logic simulator, numbers were quite small. However for the other applications, both grid-oriented applications in which much of the intra-aggregate addressing has to do with element selection on partitioned state, the caches grew to much larger size. This is probably related to the randomized aggregate interface effects described above. Most of the indexing requests are made by code running at a random representative of the aggregate, which means an arbitrary one of many nodes. This randomness may be destroying whatever locality there is in the reference streams for a particular processing node. For this reason, non-random schemes may be attractive for the *one-to-one-of-many* translation. At any rate, our simulations show that the prognosis is not good for caching indexing translation requests with partitioned state aggregates.

5.4 Improving the Implementation of Concurrent Aggregates

By studying static and dynamic properties of our programs, we have identified a number of optimizations that may have significant impact on the efficiency of CA programs. In this section, we describe these optimizations and estimate the improvement achievable in the context of our application programs. As in Chapter 4, we use the number of messages as the cost metric for computations. The rationale for this choice is given in Section 4.2.4. For efficiency comparisons to other programming approaches, we also refer the reader to Section 4.2.4 where we compared the message traffic of an optimized implementation of a Concurrent Aggregates program with the memory reference traffic required for a shared memory machine. The optimizations we considered fall into three categories: reducing the aggregate interface overhead, reducing locking overhead, and general optimizations for concurrent object-oriented languages. For each optimization, we describe an example of the code that would be optimized. Where appropriate, we indicate additional program information that must be available in order to perform the optimization. This information must come from the programmer or from compiler analysis. For each optimization we present statistics that indicate how much these optimizations will improve the performance of our programs.

5.4.1 Aggregate Interface Optimizations

In the design of many aggregates, two messages are required before a request even gets to the appropriate representative. It is desirable and plausible to reduce the linkage overhead to only one message passing operation. In many cases, where responsibility for handling a request only depends on a limited amount of aggregate specific information, we can achieve this performance improvement. In general, optimization of linkage also depends on having some information about the type (which class or aggregate) of an abstraction at the point of use. We will also call this type of optimization "request direction."

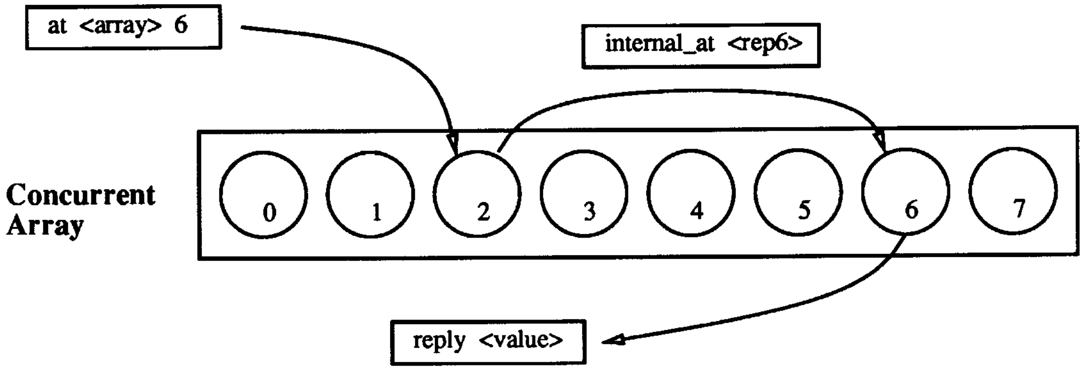
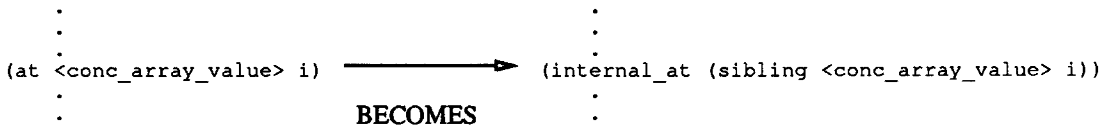


Figure 5.8: An Operation on a Concurrent Array

```
(aggregate conc_array state :no_reader_writer
           (parameters size)
           (initial size))

(handler conc_array at (index)
         (forward (internal_at (sibling group index))))

(handler conc_array atput (value index)
         (forward (internal_atput (sibling group index) value)))

(handler conc_array internal_at ()
         (reply (state self)))

(handler conc_array internal_atput (value)
         (seq (set_state self value)
              (reply (state self))))
```

Figure 5.9: Code for a Concurrent Array



```
(at <conc_array_value> i)
```

```
(internal_at (sibling <conc_array_value> i))
```

Figure 5.10: Compiling Indexing Code into the Caller

Take as an example the implementation of a concurrent array. Its operation is illustrated in Figure 5.8. Each access to the array requires three messages. The code for the concurrent array is shown in Figure 5.9. It seems clear that one third of the messages could be eliminated by using the index argument of the **at** and **atput** messages to direct the message to the correct representative. The computation for the direction operation in the **at** and **atput** messages could be compiled into the user of the abstraction, thereby avoiding the extra message. This improvement is shown in Figure 5.10 where the indexing code in the **at** operation is moved into the caller. This optimization is really a special case of *inlining* or *open-coding* methods.

Aggregate interface optimization requires some additional information about the program. Whether or not interface optimization can be performed automatically depends on what type information is available at the call site and precisely what information is used to perform the request direction. The difficulty in obtaining type information depends on the degree to which selectors are overloaded (reused) and how much information about types we can derive. For the purposes of considering this optimization, we assume that we can derive type information at the call site and focus on the information used to do the request direction. Another possibility is to design an aggregates language with more static type information.

We classify the information required to do the indirection into three levels. At each level, some number of messages can be eliminated with the appropriate information. For each level, we describe the type of information required and give an example of the kind of code being optimized.

None No information about the aggregate is required. The representative index to handle the request can be derived from the arguments to the call only. A good example of this is the Bodies aggregate in the N-body simulation. In that case, an aggregate is used only to collect the representative objects together and allow them to be manipulated as a single entity.

```
(aggregate bodies location velocity mass acc_count
           interactions iters dcombtree
           (parameters nr_bodies nr_iters)
           (initial nr_bodies
                    (fanout group (message (default_init place nr_iters))
                            0 groupsize)))
        .
        .
        .
(handler bodies accelerate (force_vector index)   ;; 1 is request
         (forward (internal_accelerate            ;; count for leaves
                      (sibling group index) force_vector 1)))

(handler bodies accelerate_body (force_vector count)
        .
        ... actually do the update on a body ...
        .
        )
```

The `accelerate` handler simply forwards request to the representative specified by `index`. If this code could be compiled into its callers, we could avoid one forwarding message transmission.

Size Information about the size (number of representatives) of the aggregate in addition to the arguments is required to compute the index of the representative to handle the request. For example, in a concurrent array or a concurrent hash table, the interface handlers compute some mapping function (or hash function) based only on the input arguments that determines which representative should really handle the request. The mapping function is combined with the size to determine the actual representative to handle the request, as using the size allows "chunking." Chunking is the distribution of work over a virtual set of representatives may be collapsed together to improve efficiency in an actual implementation. In the example below, we implement a hash table by chunking responsibility for the keys over the representatives in the aggregate. Size information may not require compiler analysis or user declaration. It may be encoded into aggregate names, as it is needed to do *one-to-one-of-many* translation.

```
(aggregate hash_integer local :no_reader_writer
        (parameters size)
        (initial size
                 (init (sibling group 0) 0)))

(handler hash_integer insert (key elt)
   (let ((hash_index (mod key groupsize)))
     (forward (internal_insert (sibling group hash_index)
                               key elt))))

(handler hash_integer internal_insert (key elt)
        .
        ... actually do the local insert ...
        .
        )
```

In the example above, the insert handler uses the key and the modulo function to determine the appropriate representative to forward the request to. The only data needed to compute the representative number is key and the size of the aggregate.

Static Some static information from the state in the aggregate may also be required to do the direction of requests. Typically, this involves some constants for the aggregate bound at creation time. This kind of information might be available in a language with a static allocation model, or a strongly typed language. For example, static information might include the bounds of a two dimensional array. The most important instance of this involves grid-oriented applications: matrix multiply and multigrid[7]. In both cases, the static information required was one of the dimensions of the array (the other could be deduced with size information). The static information would allow for dramatically better performance on these two applications. Below, we show part of the code for our two-dimensional matrix abstraction. This is an example of code where static information could be used to compile out linkage messages. In this case, the value of `xsize` is static and bound at creation time for the matrix.

```
(aggregate matrix_2d state xsize
        (parameters init_val isize ixsize)
        (initial isize
                 (init (sibling group 0) init_val ixsize)))

(handler matrix_2d at (xindex yindex)
    (forward (internal_at
               (sibling group
                  (+ (* xindex (xsize self)) yindex)))))

(handler matrix_2d internal_at ()
         (reply (state self)))
```

The `at` handler uses its arguments, the size of the `matrix_2d` aggregate and the value of `xsize` to determine which representative should handle the request. If the value of `xsize` was known at compile time, or could be cached, we could avoid a forwarding message step in the interface to `matrix_2d`.

To avoid counting efficiency gains due to optimization more than once, we only count the additional number of messages that could be eliminated by having this level of information compared to the previous level. Thus, if we had size and static information, we would sum the numbers under **None**, **Size** and **Static** to find the number of messages we would be able to optimize. The effectiveness of these optimizations is estimated by hand analysis of programs and dynamic message counts. We present the results for our application programs in Table 5.5.

These results reflect not only the different levels of aggregate usage in the different application programs, but also the way in which they are used. For example, where aggregates are used to hold replicated state, it does not matter which representative handles a request.

[7] It is also important for the N-body application. In that case, the interactions aggregate is structured like a grid.

Application	Total Msgs	None	Size	Static	Total Red.	% Red.
Matrix Mult	1,880,055	0	0	524,288	524,288	27.9%
Multigrid	2,616,544	7	0	788,878	788,885	30.1%
N-body	2,130,878	36,436	1	28,244	64,661	3.0%
PCB Router	1,283,174	13,513	14,646	0	18,159	1.4%
Logic Sim.	1,056,782	0	0	0	0	0.0%
Total	8,967,433	49,956	14,647	1,341,410	1,395,993	15.6%

Table 5.5: Messages Eliminated by Aggregate Interface Optimization

Requests are handled by whatever representative receives them. Consequently, there is no interface overhead to optimize. When aggregates are used as partitioned state (as with the concurrent array example), typically there is a level of forwarding at the interface, so significant improvement is possible. In the matrix multiply, multigrid, and N-body simulation applications the aggregates used are predominantly partitioned state. This may explain the larger improvements shown in Table 5.5.

5.4.2 Locking Optimization

In order to implement mutual exclusion for object operations, we used a simple spin-locking scheme. Upon invocation, a method typically locks its receiver object. When an invocation terminates, it unlocks the object. A message arriving at a locked object "spins" until it can acquire the lock. A message spins by repeatedly resending itself. Each time its receiver object is locked, it resends itself.

The liabilities and advantages of spin-locking schemes are well known. The "busy-waiting" consumes resources and can result in contention that requires quadratic time to process n messages. For some cases spin-locking can provide very low-latency access, often an important performance issue. Our rationale for using spin-locking was to avoid allocating heap storage for incoming messages. Spinning messages reside in the message queue where storage allocation and deallocation is much cheaper than in the heap. For most of our application programs, spin-locking did not cause significant overhead. However in a few cases, the spin-locking overhead is significant. The spin-locking overhead (number of messages induced by spinning) is shown in Table 5.6.

These results show that spin-locking caused overheads ranging from 0 - 35%. Spin-locking overhead can be eliminated by storing these messages locally (in the context of our first class message scheme), and thereby avoiding the allocation of global names for them. The messages are passive while waiting and hence do not consume any processing resources. When the desired object is unlocked, the waiting messages are resent. Thus, the necessary exclusion for Concurrent Aggregates can be implemented without the overhead caused by "busy-waiting."

Application	Total Msgs	Spin Msgs	% Reduction
Matrix Mult	1,880,055	0	0.00%
Multigrid	2,616,544	292,576	11.18%
N-body	2,130,878	744,023	34.92%
PCB Router	1,283,174	5,154	0.40%
Logic Sim.	1,056,782	348,537	32.98%
Total	8,967,433	1,390,290	15.5%

Table 5.6: Spin-Locking Overhead

5.4.3 General Optimizations

In our experience with Concurrent Aggregates, we have found a number of other situations that occur often enough to warrant optimization. These scenarios include messages to self (functions), messages to numbers or other immediate objects, and inlining objects. We describe each of these below:

Messages to Self These are messages sent to the same object on which we're currently executing. Concurrent Aggregates has only methods and handlers but no functions. This means that in order to share a piece of code, it is necessary to do two message passing operations. This can be avoided by adding functions to the language, or inlining the code for self calls.

Messages to Immediate Objects Many of the built-in objects in Concurrent Aggregates are immediates such as integers, floating point numbers, and selectors. While we would like to provide the programmer with a consistent message passing interface, we need not implement operations on immediates in that fashion. We can operate on immediates directly with much greater efficiency.

Inline Objects By inlining objects, we can make them subject to the same optimization described for immediates. In order to make an object inline, it must be immutable or have only one reference to it. We must be careful about doing this because inlining an object may transform transmissions of references to it (one word) into transmissions of the entire object (many words).

For each of our application programs, we estimated the potential for performance improvement. For **Messages to Self** and **Inline Objects**, these estimates are based on hand analysis of the code and are therefore probably conservative estimates of the performance achievable[8]. For **Inline Objects**, we only estimate the number of messages eliminated by the inlining optimization. We do not consider the increased communication traffic caused

[8] It is more likely that we overlooked an optimizable method, handler, or class than included one that could not be optimized.

Application	Total Msgs	Self	Immediates	Inline	Total Red.	% Red.
Matrix Mult	1,880,055	0	8,192	0	8,192	0.4%
Multigrid	2,616,544	177,408	4,096	0	181,504	6.9%
N-body	2,130,878	0	233,622	432,944	666,566	31.3%
PCB Router	1,283,174	0	21,143	341,053	362,196	28.2%
Logic Sim.	1,056,782	0	50,996	0	50,966	4.8%
Total	8,967,433	177,408	318,049	773,997	1,269,424	14.2%

Table 5.7: Improvement Due to General Optimizations

by inlining objects into messages. The **Messages to Immediate Objects** were deduced from dynamic message counts collected by our simulator. We present the performance improvement estimates in Table 5.7.

Optimizing calls to self has an impact on only the multigrid application. The vast majority of messages to self involved the computation of X and Y coordinates in the grid. Calls to self require messages in Concurrent Aggregates because there is no way to share code except through message handlers. This means the sharing of a pure procedure requires two messages just to call and return. This is inefficient because no message passing is really required and if the message is being sent to self, the message send will result in no additional concurrency. An improved version of CA should compile these calls to self inline or provide programmers with a more efficient way of sharing procedures.

The improvements on immediates are mostly due to the optimization of operations defined on the class of integers. The benefits were exceptionally large in the case of the N-body simulation because we computed the square root by making tail-recursive calls on integers. Each tail-recursive call shows up in our simulation results as a message. Optimization of these calls transforms them into iteration, avoiding any message passing.

Improvements in inline objects are for two simple abstractions: point and vector abstractions. Both of these abstractions were immutable and have quite small amounts of state. Both the point and vector are simple data structures. The message interface is simply accessor functions. The two dimensional vector abstraction is inlinable for the N-body simulation. Vectors are used for the positions, velocities, forces and accelerations in the simulation. For the printed circuit board router, the inline object optimized was a point. The point object was used to keep track of what points a path for a net had passed through. Naturally, every time a path was extended, several operations on points were required.

Additional Optimizations We considered several other optimizations, but were unable to investigate their effectiveness. One optimization is expression hoisting, pushing message sends as far up as possible, to facilitate concurrency. This has the benefit of reducing the computation run time if the concurrency can be exploited. Expression hoisting is analogous to bubbling loads to the top of basic blocks. Another optimization is the collection of a

Application	Total Msgs	Agg Int	Locking	General	Total	% Improv
Matrix Mult	1,880,055	524,288	0	8,192	532,480	28.3%
Multigrid	2,616,544	788,885	292,576	181,504	1,262,965	48.2%
N-body	2,130,878	64,661	744,023	666,566	1,475,250	69.2%
PCB Router	1,283,174	18,159	5,154	362,196	385,509	30.0%
Logic Sim.	1,056,782	0	348,537	50,966	399,503	37.8%
Total	8,967,433	1,395,993	1,390,290	1,269,424	4,055,707	45.2%

Table 5.8: Summary of Optimization Statistics

number of mutually unordered requests to another object. We call this optimization request combining. When method A makes two requests, B and C, to another object, D, we may have four messages, two requests and two replies. If the requests are unordered, then we should be able to compile a special method that handles the aggregate request BC, and sends a reply with two values. By performing this optimization, we can reduce the message traffic to process the two requests by as much as a factor of two.

5.4.4 Summary of Optimization Improvement

We have found that a number of optimizations can significantly reduce the number of messages required in to execute Concurrent Aggregates programs. The reduction in message count ranged from 28% to nearly 70%. The overall average improvement was just over 45%. As communication is likely to ultimately be the limiting resource in fine-grain message-passing machines, this is a very significant reduction. One interesting outcome of our study is that no single class of optimizations dominated for all of the applications. In order to get nearly a 30% improvement in all of them, it is necessary to apply all of the optimizations. It also seems clear that type information for aggregates could be used to significantly improve program performance (see messages under Agg Int).

5.5 Summary

In this chapter, we have considered the key implementation issues for Concurrent Aggregates. First, we considered the use and implementation of first class continuations and messages. We carefully designed the semantics of system continuations in CA to fulfill their purpose, yet remain easy to implement efficiently. We measured the usage of first class messages and found it to be quite rare. In view of this, we presented an implementation that handles the common case – no explicit references to a message – very efficiently. Message references are trapped and handled specially. As long as the usage rate is quite low, this will be an efficient implementation.

Second, we considered the problem of aggregate to representative name translation. This is important for aggregate interfaces, *one-to-one-of-many* translation, and intra-aggregate addressing, indexing. Our application program simulations in Chapter 4 reflect a static mapping from aggregate to representative name. In Section 5.3.2, we explored a dynamic mapping that would allow objects to participate in multiple aggregates and allow aggregates to be formed dynamically. In order to support this dynamic translation efficiently, local translation caching for each node is an attractive alternative. Using an ideal cache model, we examined the potential of one class of dynamic translation schemes. We found that the caching can be quite effective for the aggregate interface translation, but is not very effective for the indexed translation. We hypothesized that this was due to the randomization of the aggregate interface translation in our simulations. If a more deterministic scheme is used, or some aggregate interface compiler optimizations are performed, we suspect that the caching will be more effective for indexed translation. Until we have results based on those different assumptions, we must conclude that dynamic translation schemes appear to be quite expensive, as simple caching is not likely to speed intra-aggregate addressing requests significantly.

Third and finally, we considered a number of optimizations for improving the performance of Concurrent Aggregates. These optimizations were divided into three categories: aggregate interface, locking and general optimizations. Two of these, locking and general optimizations, can be performed without any changes to the language. We found that these two sets of transformations were likely to cause 15.5% and 14.2% reductions in the number of messages for our application programs. The third set of aggregate interface optimizations require some more type information about the program. This information could be supplied by the programmer (in a statically-typed or type-annotated version of CA) or be inferred by the compiler. With complete information, we estimated that aggregate interface optimizations might yield an additional 15.6% reduction in necessary message traffic. Altogether, this set of optimizations promises the hope of reducing message traffic by 30-50% in Concurrent Aggregates programs.

Chapter 6

Conclusion

In this section, I summarize the work I have presented in this thesis. I begin by highlighting the research contributions and place them in perspective in the development of programming languages for fine-grained message passing machines. In closing, I discuss issues and ideas for future directions to pursue in programming systems for fine-grained message passing machines.

6.1 Summary of Present Work

Concurrent Aggregates represents an important step in the development of programming systems for fine-grained message passing machines. If there is an underlying lesson motivating the development of aggregates it is this:

> *Whatever tools you use to manage complexity, they must not reduce concurrency.*

The multi-access abstraction tools provided in Concurrent Aggregates comply with this requirement, allowing programs to be modularized without restraining concurrency. The *one-to-one-of-many* interface and simple intra-aggregate addressing primitives were sufficient for expressing a wide variety of multiple access abstractions. The major classes of these multi-access abstractions include consistent replication, loosely consistent replication, partitioned state, and structured cooperation. Allowing programmers to manage consistency and update of aggregate state within the programming model is useful in a wide variety of situations. It was typically used to support higher performance implementations of abstractions.

We have written a number of significant application programs in Concurrent Aggregates. The application programs we constructed include matrix multiplication, multigrid relaxation, N-body interaction simulation, printed circuit board router, concurrent B-tree, digital logic simulation and a parallel FIFO queue. These applications show us broad variety of uses for aggregates in concurrent programs. The application programs have also been used to evaluate CA with respect to programmability and concurrency. Our experience with the language has been quite positive. Aggregates facilitate the modularity of programs while allowing the programmer to exploit a great deal of concurrency.

We have found that aggregates can be used to effectively structure programs without reducing their concurrency. Further, novel features in Concurrent Aggregates have allowed us to express a number of different styles of parallelism: both data parallelism and control parallelism. More importantly, novel features such as first class and user continuations as well as first class messages are an important step towards building composable object-oriented programs. Concurrent Aggregates provides some basic tools, but real composability involves deeper issues than syntactic convenience and facilitation of code reuse. The interaction of composition and issues of concurrency control and resource management have yet to be explored.

An important source of efficiency in shared address space multiprocessors is the ability to do address calculation. This capability is particularly powerful in languages such as FORTRAN which support flat multi-dimensional arrays. Address calculation in sequential machines can be used to eliminate unnecessary memory references. In multiprocessors, this calculation can be used to eliminate unnecessary communication. In concurrent object-oriented languages, this kind of address computation has been largely ignored. Aggregates provide a framework for expressing address calculation in an object-oriented context. Coupled with some of the optimizations described in Chapter 5, aggregates provide the opportunity to recapture the efficiency due to address calculation.

We studied issues in an efficient implementation of Concurrent Aggregates. A number of features in Concurrent Aggregates are sufficiently novel to merit special consideration. First class continuations and user-defined continuations were carefully designed to require little special support. At their current level of usage ($< 5\%$), the cost of first class messages appears to be manageable. We studied aggregate naming translations in detail and proposed two different implementations. We found that a static mapping between aggregate and representative names is the most attractive. Such a mapping not only supports aggregate interface translation and intra-aggregate addressing efficiently, it is also compatible with the aggregate interface compiler optimization techniques studied in Chapter 5. We considered three classes of compiler optimizations for Concurrent Aggregates programs: aggregate interface, locking and general optimizations. It is clear that type information for aggregates can dramatically improve aggregate interface efficiency[1]. Together these optimizations

[1] It appears that the cost of maintaining the abstraction barrier at run time in the concurrent world is even larger than in the sequential world. In Smalltalk-80 [49], much work is required to increase the efficiency of the object interface [44, 24, 26]. Many of these optimizations may not be appropriate or effective in fine-grained concurrent machines. However, efficient interfaces are crucial because one cost of an inefficient interface is communication. A system designer must consider the price appropriate for facilitating code sharing and reuse.

reduced the number of messages in our computations by 30-50%. The improved performance seems likely to be competitive with other approaches in terms of communication requirements (see Section 4.2.4).

6.2 Future Work

If fine-grained message-passing machines are to have a significant impact in high performance computing, their programming systems must meet several criterion. First, they must support the expression of large amounts of concurrency. Second, it must be comparable in terms of complexity and convenience to write programs. This means that the programmers must have good tools for developing programs and managing their complexity. Third, the programs must provide some locality information, allowing the implementation to reduce the communication bandwidth requirement. Finally, programmers must be able to tune the performance of their programs by adding more information. At a reasonable level of effort, the efficiency must approach that achievable on sequential machines.

The work in thesis has focused primarily on the first two requirements. Our programs exhibit massive concurrency. They are still more difficult to write than sequential programs, but their complexity is now manageable. We turn our attention to the issues of locality and efficiency.

Programming systems must allow the clustering of data, so it can be placed on the same processor and accessed more efficiently. The perspective in fine-grained machines is fundamentally different from conventional machines. In fine-grained machines, we bring the computation to the data. In conventional machines, the data is brought to the computation. A process can take the time to build a "working set" in its cache or the local node memory. In fine-grained machines, tasks are short-lived. They do not exist long enough to build up a working set.

The composability of object-oriented programs must be improved. The tools provided in Concurrent Aggregates represent some first steps in this direction. However, many issues have yet to fully examined (interaction of locality, synchronization, and concurrency management across composition). Functional languages are quite good at composing functions. However, because they typically do not support expression of locality, they do not do well in composing data. Computation in such languages is generally done against a global shared memory. Functions can be composed, but the data structures cannot. Efficient performance in fine-grain message passing machines demands the development of a broader form of composition.

In this thesis, we have avoided a number of open research problems that must be solved to support flexible programming systems for fine-grained message passing machines. We briefly describe them here.

The locking structure (object concurrency control) incorporated in Concurrent Aggregates is simple, and in some cases too restrictive. We need to find an effective way of reconciling concurrency control, low overhead, and the desire to express recursive formulations without concern for deadlock. One possibility is to allow users to define locking protocols for objects by assuring that the language implementation made calls to special lock and unlock functions. These functions could perform computation on the local object state, allowing expression of any locking policy ranging from local locks to the more complicated per variable locks of CST.

Difficulty in supporting recursion is one significant criticism of most simple locking schemes. Simple recursion is a convenient programming idiom and easy to detect and implement with a sophisticated compiler. The recursive call should be compiled as a local procedure call. More complex forms or recursion are difficult to support as they require the breakdown of the simple exclusion model which is one important benefit of the object-oriented programming approach. Aggregates provide some hope in this area because requests to an aggregate could be viewed as the start of a request. With this view, it is possible to allocation a transaction ID and use that for exclusion. This is similar to many distributed systems that support call back in their RPC protocols [14]. The overhead for such a scheme is unclear.

One underlying premise of Concurrent Aggregates is that there is an efficient garbage collector to reclaim names. In distributed memory machines such as the J-machine, such collection may be quite expensive because each pointer traversal may require a message transmission. However, since our local memories are relatively small, this overhead is not as great as one might think. For a 4 kiloword per node machine with objects with average size 8 words, each node could support a maximum of 512 objects. Sending 512 messages per node during a garbage collection may be an acceptable amount of traffic. Unfortunately, the receipt of messages may not be nearly so uniform. An uneven distribution can cause one node to become the bottleneck, preventing the garbage collection from completing quickly. One way to avoid this kind of bottleneck is to use a randomized routing technique similar to that described by Valiant [98] coupled with dynamic combining.

Load balancing is a key problem in efficiently utilizing a parallel machine. For our application programs, randomized initial placement was sufficient. However, for computations with long-lived objects or less regular structure, we expect that object migration will be necessary to balance both memory occupancy and processor load. Larger collections of data such as aggregates form a natural basis for distribution over a machine. High-level analysis of programs based on their modular structure may prove useful in determining good placements for load balance.

Concurrency management can be a key factor in the performance of parallel machines. Exposing too much concurrency can reduce performance or worse yet, require so much resources that the program cannot complete. We have not explicitly considered concurrency management – how to deal with an overabundance of concurrency. We expect that meta-abstractions making use of first class messages may be useful here. Aggregates also provide a natural place to control concurrency. A compiler working in conjunction with a sophisticated run time system might scale the sizes of aggregates to control the amount of concurrency exposed. One advantage we have with a language like Concurrent Aggregates is that the object-oriented model, as an open system, allows blending of operating system and application program concerns. It is easy to build programs that condition their behavior upon request of the run time system.

Locality will be of increasing importance as we scale fine-grained message-passing machines into the tens of thousands to millions of nodes. Ultimately, the scalability of these fine-grained message passing machines will be network limited. Efficient management of communication resources is a crucial factor in system performance. Studies such as [59] have shown that in many cases it is possible to determine that objects will require significant communication with each other and place them on the same node. This type of information will allow us to reduce message-passing linkage overhead *and* communication requirements.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[3] P. Agrawal, R. Tutundjian, and W. Dally. Algorithms for Accuracy Enhancement in a Hardware Logic Simulator. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 1–4, Las Vegas, Nevada, June 1989. ACM/IEEE.

[4] Ramune Alauskas. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Computers.* Association for Computing Machinery, ACM Press, January 1988.

[5] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–5. AFIPS, 1967.

[6] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings of ECOOP*, pages 234–42. Springer-Verlag, June 1987.

[7] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *Conference on Programming Language Design and Implementation*, pages 11–20. ACM SIGPLAN, 1988.

[8] Arvind, R. Nikhil, and K. Pingali. Id Nouveau Reference Manual, Part I: Syntax. Computation structures group, MIT Laboratory for Computer Science, 1987.

[9] Arvind, R. Nikhil, and K. Pingali. Id Nouveau Reference Manual, Part II: Semantics. Computation structures group, MIT Laboratory for Computer Science, 1987.

[10] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[11] Association for Computing Machinery. *Proceedings of the Fifth Distributed Memory Computers Conference*, Charleston, South Carolina, April 8-12 1990. ACM Press.

[12] William C. Athas. *Fine Grain Concurrent Computations.* PhD thesis, California Institute of Technology, 1987. 5242:TR:87.

[13] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, pages 9–24, August 1988.

[14] B. S. Bacarisse, S. R. Wilbur, J. Crowcroft, and M. Riddoch. The Design and Implementation of a Protocol for Remote Procedure Call. Technical report, University of London, 1986.

[15] Henri E. Bal. *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems.* PhD thesis, Vrije Universiteit Te Amsterdam, Amsterdam, 1989.

[16] K. E. Batcher. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, C-29(9):836–840, September 1980.

[17] J. Bennett, J. B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. Technical Report Rice COMP TR89-98, Rice University, 1989.

[18] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings of OOPSLA '86*, pages 78–86. ACM, September 1986.

[19] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[20] Nan J. Boden. A Study of Fine-Grain Programming Using Cantor. Master's thesis, California Institute of Technology, 1988. Caltech-CS-TR-88-11.

[21] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWARP. In *Proceedings of the 17th International Symposium on Computer Architecture.* IEEE Computer Society, 1990.

[22] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*, pages 207–10. Computer Science Press, Inc., 1976.

[23] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[24] Patrick J. Caudill and Allen Wirfs-Brock. A Third Generation Smalltalk-80 Implementation. In *OOPSLA '86 Proceedings*, pages 119–30, September 1986.

[25] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[26] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–60, Portland, Oregon, June 1989. ACM Press.

[27] J. S. Chase, F. G. Amador, E. Lazowska, H. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of Twelfth Symposium on Operating Systems Principles*, pages 147–58. ACM SIGOPS, ACM Press, December 1989.

[28] Andrew A. Chien. CA Language Report, Version 1.0. MIT Concurrent VLSI Architecture Group Memo 26, August 1989.

[29] Andrew A. Chien. Application Studies for Concurrent Aggregates. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1990.

[30] Andrew A. Chien and William J. Dally. Concurrent Aggregates (CA). In *Proceedings of Second Symposium on Principles and Practice of Parallel Programming.* ACM, March 1990.

[31] Andrew A. Chien and William J. Dally. Experience with Concurrent Aggregates (CA): Implementation and Programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*, Charleston, South Carolina, April 8-12 1990. SIAM.

[32] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation Strategies for Continuations. In *Conference on Lisp and Functional Programming*, pages 124–31. ACM, 1988.

[33] William D. Clinger. Foundations of Actor Semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[34] G. W. Clow. A Global Routing Algorithm for General Cells. In *Proceedings of the 21st Design Automation Conference*, pages 45–51. IEEE, 1984.

[35] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–8, June 1979.

[36] O. J. Dahl and K. Nygaard. SIMULA – An Algol-Based Simulation Language. *Communications of the ACM*, 9(9):671–8, September 1966.

[37] William Dally and Andrew Chien. Object Oriented Concurrent Programming in CST. In *Proceedings of the Third Conference on Hypercube Computers*, pages 434–9, Pasadena, California, 1988. SIAM.

[38] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Boston, Mass., 1987.

[39] William J. Dally. Virtual Channel Flow Control. In *Proceedings of the 17th International Symposium on Computer Architecture.* IEEE Computer Society, 1990.

[40] William J. Dally. Express Cubes. *IEEE Transactions on Computers*, 1991. To Appear.

[41] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a Message-Driven Processor. In *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, pages 189–196. IEEE, June 1987.

[42] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A Fine-Grain Concurrent Computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.

[43] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–4. IEEE Computer Society, 1987.

[44] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[45] D. M. Dias and J. R. Jump. Analysis and Simulation of Buffered Delta Networks. *IEEE Transactions on Computers*, C-30(4):273–82, April 1981.

[46] J. Dion. Fast Printed Circuit Board Routing. Technical Report WRL Research Report 88/1, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California, 1988.

[47] J. Elliot and B. Moss. Managing Stack Frames in Smalltalk. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–40, 1987. SIGPLAN NOTICES Volume 22 Number 7 July 1987.

[48] P. Wegner G. Agha and A. Yonezawa, editors. *Workshop on Object-Based Concurrent Programming.* ACM SIGPLAN, ACM Press, April 1988.

[49] Adele Goldberg and David Robertson. *Smalltalk-80 The language and its implementation.* Addison-Wesley, 1985.

[50] Benjamin Goldberg. Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 313–21, Portland, Oregon, June 1989. ACM SIGPLAN, ACM Press.

[51] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Lecture Notes in Computer Science 78*, chapter Edinburgh LCF. Springer Verlag, 1979.

[52] L. Greengard and V Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73:325–48, 1987.

[53] Guy L. Steele, Jr. and W. Daniel Hillis. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 279–97, Cambridge, Massachusetts, August 1986.

[54] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. Technical report, Carnegie Mellon University, School of Computer Science, 1988.

[55] Danny Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[56] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–83, 1986.

[57] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and Implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–9. ACM SIGPLAN, ACM Press, 1989.

[58] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.

[59] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, Department of Computer Science, Seattle, Washington, 1987. TR-87-01-01.

[60] Simon L Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[61] R. Halstead Jr. Parallel Symbolic Computing. *IEEE Computer*, pages 35–43, August 1986.

[62] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, University of Washington, Department of Computer Science, Seattle, Washington, 1988. TR-88-12-06.

[63] H. T. Kung. Why Systolic Architectures? *IEEE Computer Magazine*, January 1982.

[64] Monica S. Lam. A Systolic Array Optimizing Compiler. Technical Report CMU-CS-87-187, Carnegie Mellon University, 1987.

[65] Leslie Lamport and Nancy Lynch. Chapter on Distributed Computing. Technical Report MIT-LCS-TM-384, Massachusetts Institute of Technology, February 1989.

[66] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. In *Proceedings of the Fall Joint Computer Conference*, pages 380–6, November 1986.

[67] C. Y. Lee. An Algorithm for Path Connection and its Applications. *IRE Transactions on Electronic Computers*, pages 346–65, September 1961.

[68] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–70, December 1981.

[69] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA '86*, pages 214–23. ACM SIGPLAN, ACM Press, 1986.

[70] Henry Lieberman. Concurrent Object Oriented Programming in ACT 1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[71] Bruce Lindsay. Object Naming and Catalog Mangement for a Distributed Database Manager. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 31–40, 1981.

[72] Barbara Liskov. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*, 23(5), May 1988.

[73] Olaf M. Lubeck and V. Faber. Modeling the Performance of Hypercubes: A Case Study Using the Particle-in-Cell Application. Technical Report LA-UR-87-15222, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, 1987.

[74] Carl R. Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, Massachusetts Institute of Technology, August 1987.

[75] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. *MP-1 Family Data-Parallel Computers.*

[76] W. J. A. Mol. On the Choice of Suitable Operators and Parameters in Multigrid Methods. Technical Report NW 107/81, Department of Numerical Mathematics, stichting mathematisch centrum, June 1981.

[77] Peter Naur. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[78] NCUBE, Beaverton, Oregon. *NCUBE 2 6400 Series Supercomputer: Technical Overview*, 1989.

[79] M. Noakes and W. J. Dally. System Design of the J-machine. In *Proceedings of Sixth MIT Conference on Advanced Research in VLSI*, 1990.

[80] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[81] Jonathan Rees, William Clinger (editors), et al. *Revised*^{3} Report on the Algorithmic Language Scheme. Memo 848a, M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, September 1986.

[82] R. M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21:63–72, 1978.

[83] M. Schwartz and T. Stern. Routing Techniques Used in Computer Communication Networks. *IEEE Transactions on Communications*, COM-28(4):265–78, April 1980.

[84] C. L. Seitz, J. Seizovic, and W. Su. The C Programmer's Abbreviated Guide to Multicomputer Programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, 1988.

[85] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[86] Charles L. Seitz. Submicron Systems Architecture: Semiannual Technical Report. Technical Report Caltech-CS-TR-90-05, Department of Computer Science, California Institute of Technology, March 1990.

[87] Charles L. Seitz, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig S. Steele, and Wen-King Su. The Architectre and Programming of the Ametek Series 2010 Multicomputer. In *Proceedings of the Third Conference on Hypercube Computers*, pages 33–6. Association for Computing Machinery, ACM Press, January 1988.

[88] H. Shin and A. Sangiovanni-Vincentelli. Mighty: A 'Rip-up and Reroute' Detailed Router. In *Proceedings of International Conference on Computer-Aided Design*, pages 2–5. IEEE, 1986.

[89] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[90] Burton J. Smith. A pipelined, shared resource MIMD computer. In *IEEE Proceeding of the International Conference on Parallel Processing*, pages 6–8. IEEE, 1978.

[91] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–356, 1969.

[92] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Reference Manual.*

[93] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, April 1987.

[94] Brian K. Totty. An Operating Environment for the Jellybean Machine. Bachelor's thesis, Massachusetts Institute of Technology, 1988.

[95] E. G. Ulrich. Time-Sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths. In *Proceedings of the 20th National ACM Conference*, pages 437–47, 1965.

[96] E. G. Ulrich. Exclusive Simulation of Activity in Digital Networks. *Communications of the ACM*, 12(2):102–10, February 1969.

[97] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*, pages 227–41. ACM SIGPLAN, ACM Press, 1987.

[98] L.G. Valiant. A Scheme for Fast Parallel Communication. *SIAM Journal of Computing*, 11(2):350–361, May 1982.

[99] W. A. Dees, Jr. and P. G. Karger. Automated Rip-Up and Reroute Techniques. In *Proceedings of the 19th Design Automation Conference*, pages 432–9. IEEE, June 1982.

[100] Paul Wang. An In-Depth Analysis of Concurrent B-tree Algorithms. Master's thesis, Massachusetts Institute of Technology, 1990.

[101] William E. Weihl and Paul Wang. Multi-Version Memory: Software Cache Management for Concurrent B-trees. 1990. Submitted for Publication.

[102] S. Winograd. *Complexity of Sequential and Parallel Numerical Algorithms*, chapter Some Remarks on Fast Multiplication of Polynomials, pages 181–96. Academic Press, New York, 1973.

[103] Chuan-Lin Wu and Tse-Yun Feng. On a Class of Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-29(8):694–702, August 1980.

[104] Yale University, New Haven, Connecticut. *Report on the Programming Language Haskell*, 1.0 edition, April 1990.

[105] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Object-Oriented Concurrent Programming – Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.

[106] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* Computer Systems Series. The MIT Press, Cambridge, Massachusetts, 1987.

[107] F. Zhao. An O(n) Algorithm for 3-Dimensional N-body Simulation. Technical Report MIT-AI-TR-995, MIT Artificial Intelligence Laboratory, 1987.

[108] F. Zhao and S. L. Johnsson. The Parallel Multipole Method on the Connection Machine. Technical Report YALEU/DCS/TR-749, Yale University, Department of Computer Science, 1989. Also available as a Thinking Machine Corporation Technical Report.

Appendix A

Program Examples

In this appendix, we describe a number of program example that exercise some of the novel features in CA. We also work through an application programs in detail. This provides the reader with a complete, documented example of a moderate-sized Concurrent Aggregates program. These examples should give the reader some feel for what is would be like to program in CA. Complete documentation of our application programs: source code listings and statistics are available in [29].

A.1 Examples of Novel CA Features

A.1.1 Using First Class Messages

First class messages can be used to implement operations on collections – exploiting "data parallelism." For instance, in an N-body interaction simulation, first class messages can be used to implement position update for a collection of bodies. Figure A.1 depicts such a scenario. The CA code required to implement the fan out `tree` aggregate is shown in Figure A.2. A single message sent to the fan out tree causes all bodies in the collection

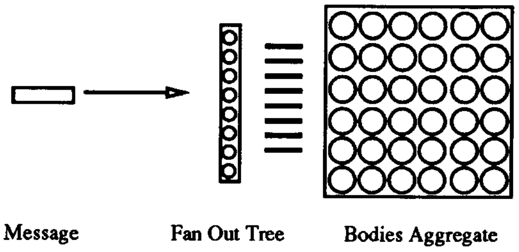


Figure A.1: An aggregate operation – move bodies

```
;; first, include the range abstraction
(load "range.ca")
;;
;; A fanout tree
;;
(aggregate tree
        (parameters treesize) (initial treesize))

(handler tree fan_out (mess bodies-agg range)
        (if (> (size range) 2)
            (let ((lrange (split_range range)))
                (do (fan_out group mess bodies-agg lrange))
                (do (fan_out group mess bodies-agg range)))
                ;; range has been side-effected
        (forall index from (low range) below (high range)
                (send mess (sibling bodies-agg index)))))
;;
;;  A bodies aggregate
;;
(aggregate bodies xpos ypos xvel yvel mass
        (parameters psize)
        (initial psize))

... other handlers ...

;; Sample Usage
;;
;;(fan_out <tree>
;;   (message (move <doesnt-matter> 10)) <bodies>)
```

Figure A.2: Abstract Implementation of operations on a Collection

to receive `move` messages with the current time step. Each time a `fan_out` message is executed, the message parameter, `mess`, is copied. Each fan_out message is processed by a representative determined by the runtime system. Thus, when we reach the leaves of the fan out tree, there is one copy of the message for each representative of the bodies aggregate. The last stage of the fan out tree transforms the messages (by writing the appropriate representative's name into message's destination field) and sends them.

The `tree` aggregate demonstrates an interesting use of aggregates – as computational bandwidth. The fan out function of the tree makes no use of state in the `tree` aggregate. The representatives are only used as computation sites. This means the fan out tree can be used by many fan out operations simultaneously. One could also use the size of the fan out tree aggregate to limit the fan out rate for the tree.

```
(load "pair.ca")
;;
;; A future Object
;;
(class future tag val deferred
        (initial (set_tag self 0)
                 (set_deferred self 0)))

(method future value ()
        (if (= 0 (tag self))
            (set_deferred self (new pair requester (deferred self)))
          (reply (val self))))

(method future set_value (value)
        (set_val self value)
        (set_tag self 1)
        (seq (do (forward_replies (deferred self) value))
             (set_deferred self 0))
             (reply value))
```

Figure A.3: Code for a Future Object

A.1.2 Manipulating System Continuations

The Concurrent Aggregates language allows programs to manipulate continuations as first class objects. Programs can get references to continuations by using **msg_at** on a message or accessing the **requester** pseudo-variable. **requester** contains the value of the current continuation.

Programmer manipulation of continuations can simplify programs. Figure A.3 shows first class continuations being used to implement futures [61]. The future object understands two messages: **value**, which returns the value of the future, and **set_value**, which defines the future's value. When empty (the value is undefined), a future object pushes the continuations of all **value** requests onto a list. When the future receives a **set_value** message, replies are sent to each continuation on the list. Subsequent **value** messages receive immediate responses.

The future object consists of three instance variables: **tag** – holds full or empty status, **val** – the future's value, and **deferred** – a linked list of continuations. When the value is received, it is stored in **val**, **tag** is modified to indicate full, and the value is sent to all of the continuations on the list. The **pair** object definition is not shown.

```
;;
;; A barrier synchronization abstraction
;;
(class barrier count maxcount next_message
        (parameters imaxcount inext)
        (initial (set_count self 0)
                 (set_maxcount self imaxcount)
                 (set_next_message self inext)))

(method barrier reply (val)
        (seq (set_count self (+ val (count self)))
             (if (= (count self) (maxcount self))
                 (send (next_message self)))))
```

Figure A.4: A Barrier Synchronization Object

A.1.3 Objects as Continuations

CA allows objects or aggregates to be used as continuations. In a concurrent message passing language, a continuation is simply an object expecting a `reply` message. CA programs can substitute any object or aggregate that handles the `reply` message for a continuation. This feature facilitates the construction of complex synchronization structures – now continuations can perform any user program computation. This makes it possible to factor the synchronization code out of programs and into abstractions, allowing the remaining program code to be written without regard for the synchronization context in which it is being used. For example, a barrier synchronization can be implemented as shown in Figure A.4. Each `reply` message to a barrier object is processed by the `reply` method. After `count` has reached `maxcount`, all elements of the set have arrived and the barrier is complete. Upon completion, the barrier abstraction sends an arbitrary message, `next_message`, which starts the next stage of the computation. The computations being synchronized need not know that they are being barrier synchronized – they obliviously send `reply` messages to their continuations. Of course, the barrier abstraction implementation could be concurrent – collecting the reply messages in a combining tree.

It is also possible to construct many other complex synchronization structures. As an example, we present code for a "race" object in Figure A.5. A race object was introduced in the Actor model as a way to describe speculative concurrency [70]. A number of computations are started and the race object takes on the value returned by the first to complete. `value` messages received before any of the computations has returned are deferred – the race object also performs future style synchronization. Later replies are discarded. A code fragment that uses the race object to get the first answer from three different solution strategies is also shown in Figure A.5.

```
(load "pair.ca")
;;
;;  a Race abstraction
;;
(class race val tag deferred
        (initial (set_tag self 0)
                 (set_deferred self 0)))

(method race reply (value)
        (if (= 0 (tag self))
            (seq (set_val self value)
                 (set_tag self 1)
                 (do (forward_replies (deferred self) value))
                 (set_deferred self 0))))

(method race value ()
        (if (= 0 (tag self))
            (set_deferred self (new pair requester (deferred self)))
          (reply (val self))))
;;
;; Using three strategies to solve a problem, using the
;; race object.
;;
(method osystem initial_message ()
        (let ((race_object (new race)))
           (do (strategy1 a b c) race_object)
           (do (strategy2 a b c) race_object)
           (do (strategy3 a b c) race_object)
           (reply (value race_object))))
```

Figure A.5: A Race object for speculative concurrency

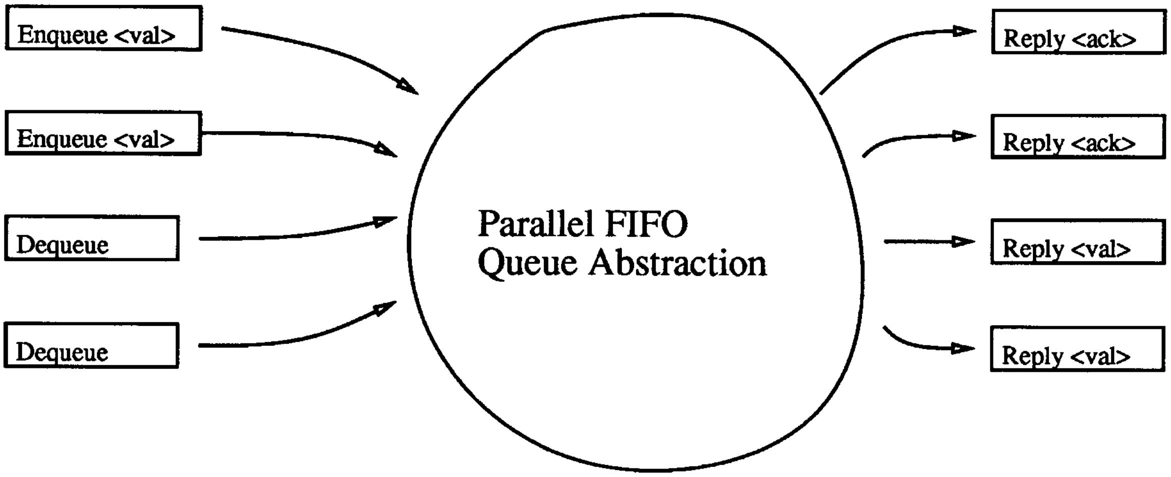


Figure A.6: Parallel FIFO Queue

A.2 Parallel Queue

A parallel FIFO (first-in-first-out) queue accepts enqueue and dequeue requests. For the enqueue requests, the value is stored in the queue and an acknowledgement is returned. For the dequeue requests, the value returned is the dequeued element. Our implementation of a parallel FIFO queue is based on an implementation of queues on the Ultracomputer. However, no combining hardware is required as we do combining in software. The behavior of our queue is FIFO when requests are presented sequentially, but deviates from that when many requests are presented to it concurrently. A parallel queue is depicted in Figure A.6.

The structure of our queue is shown in Figure A.7. It consists a queue interface, two dynamic combining trees, a fifo counter object, and a synchronizing array. The fifo counter manages the array as a circular buffer. The combining trees are used to increase the effective bandwidth of the fifo counter, and the queue interface aggregate is used to hold the whole thing together.

Operationally, the queue interface receives enqueue and dequeue messages and requests an index of the appropriate type – "in" or "out." Based on the type of index it needs, the queue interface makes a request to the appropriate combining tree. Once it receives an index, it writes (or reads) the corresponding location in the array to complete the operation. The code for the queue interface is shown in Figure A.8.

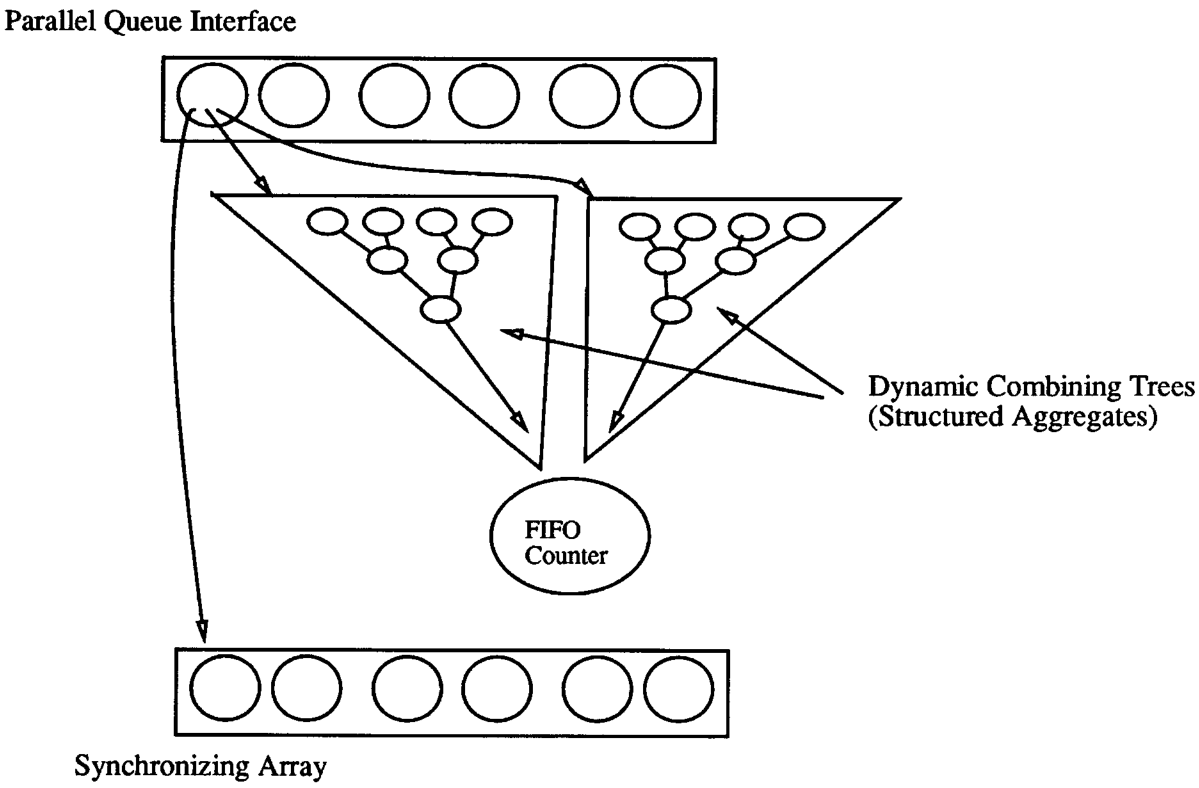


Figure A.7: Structure of the Parallel FIFO Queue

```
;;
;; parallel queue interface, 12/89, Andrew A. Chien
;;

(aggregate parallel_q in_tree out_tree data_array :no_reader_writer
    (parameters capacity asize)
    (initial capacity
             (let ((d_array (new pvector_presence asize)))
               (let ((f_count (new fifo_counter asize d_array)))
                 (let ((i_tree (new dcombtree (/ capacity 4)
                                              f_count in_increment))
                       (o_tree (new dcombtree (/ capacity 4)
                                              f_count out_increment)))
                   (init_parallel_q (sibling group 0) i_tree o_tree d_array))))))

(handler parallel_q init_parallel_q (i_tree o_tree d_array)
    (let ((base_index (* myindex 2)))
       (let ((index0 (+ base_index 1))
             (index1 (+ base_index 2)))
         (seq (conc (if (< index0 groupsize)
                        (init_parallel_q (sibling group index0)
                                         i_tree o_tree d_array))
                    (if (< index1 groupsize)
                        (init_parallel_q (sibling group index1)
                                         i_tree o_tree d_array)))
              (set_in_tree self i_tree)
              (set_out_tree self o_tree)
              (set_data_array self d_array)
              (reply done)))))

(handler parallel_q enqueue (value) :no_exclusion
     (let ((index (in_increment (in_tree self) 1)))
       (forward (atput (data_array self) value index))))

(handler parallel_q dequeue () :no_exclusion
     (let ((index (out_increment (out_tree self) 1)))
       (forward (at (data_array self) index))))
```

Figure A.8: A Parallel Queue Interface Aggregate

The code for the parallel queue interface in Figure A.8 shows the enqueue and dequeue message handlers, as well as the aggregate declaration and initialization. The initialization code creates all parts of the parallel queue.

One important part of the parallel queue is the dynamic combining trees. The CA code for trees is shown in Figure A.9.

The dynamic combining tree initialization code links the representatives in the aggregate together into a tree. The connections that for the tree are held in the **myparent** instance variable and used to define what direction is toward the root in the tree. The combining tree is "dynamic" in that the requests that will be combined are not statically determined. The combining sequence depends on the run time program behavior.

The **in_increment** and **out_increment** requests can be combined. Each node combines requests over a period of time then propagates the consolidated request up the tree when the node receives a **send_requests** message (which it sent to itself). Thus, all the messages that were waiting in the input queue of a tree node when the first request arrived get combined before the request is propagated up the tree. We use one program to instantiate both combining trees, so both trees could handle **in_increment** and **out_increment** requests. The program structure at run time assures that each tree only receives one type of request.

As requests are combined in the trees, a tree of continuations with matching structure is constructed. The construction of the continuation tree is shown in Figure A.10. At each combining tree node, the continuations for combined requests are pushed onto a list. The list is used as the continuation for the consolidated request. Thus, there is exactly one such continuation tree for each request that gets to the fifo counter object. The **reply** from the fifo counter is propagated through the **continuation_struct** tree.

The **continuation_struct** tree is used to assign indices to the combined requests. The message that arrives at the fifo counter is an **in_increment** or an **out_increment** message with an argument that represents the number of indices needed. The fifo counter allocates that number of indices and replies with the starting index in a range to the tree of continuations. The tree of **continuation_structs**, code shown in Figure A.11, maps this range of indices to the original requests. Each **continuation_struct** holds the weight of the requests in its left subtree. Thus as we descend through the tree, at each level, we simply allocate **(weight self)** indices to the left subtree and the remainder for the right subtree. The range of indices can be represented with one number, its starting index, because the size of the range is implicit in the number of request leaves in the tree.

```
;;
;;  A dynamic combining tree
;;
(load "continuation_struct.ca")

(aggregate dcombtree waiting_reqs
       myparent consolidated_value selector :no_reader_writer
       (parameters size counter comb_selector)
       (initial size
                (seq (forall index from 0 below groupsize
                         (init_help (sibling group index) comb_selector))
                                 ;; initial method received by sibling 0
                     (set_myparent self counter))))

(handler dcombtree init_help (comb_selector)
         (seq (set_waiting_reqs self 0)
              (set_consolidated_value self 0)
              (set_myparent self (sibling group (/ myindex 2)))
              (set_selector self comb_selector)
              (reply done)))
;;
;; if waiting requests, combine.  Else push and send ping
;;
(handler dcombtree in_increment (val)
     (seq (if (eq 0 (waiting_reqs self)) (do (send_requests self)))
          (set_consolidated_value self (+ val (consolidated_value self)))
          (set_waiting_reqs self (new continuation_struct
                                      requester val (waiting_reqs self)))))

(handler dcombtree out_increment (val)
... identical to in_increment ...)

(handler dcombtree send_requests ()
     (let ((mysel (selector self)))
       (seq (do (mysel (myparent self) (consolidated_value self))
                (waiting_reqs self))
            (set_waiting_reqs self 0)
            (set_consolidated_value self 0))))
```

Figure A.9: A Dynamic Combining Tree

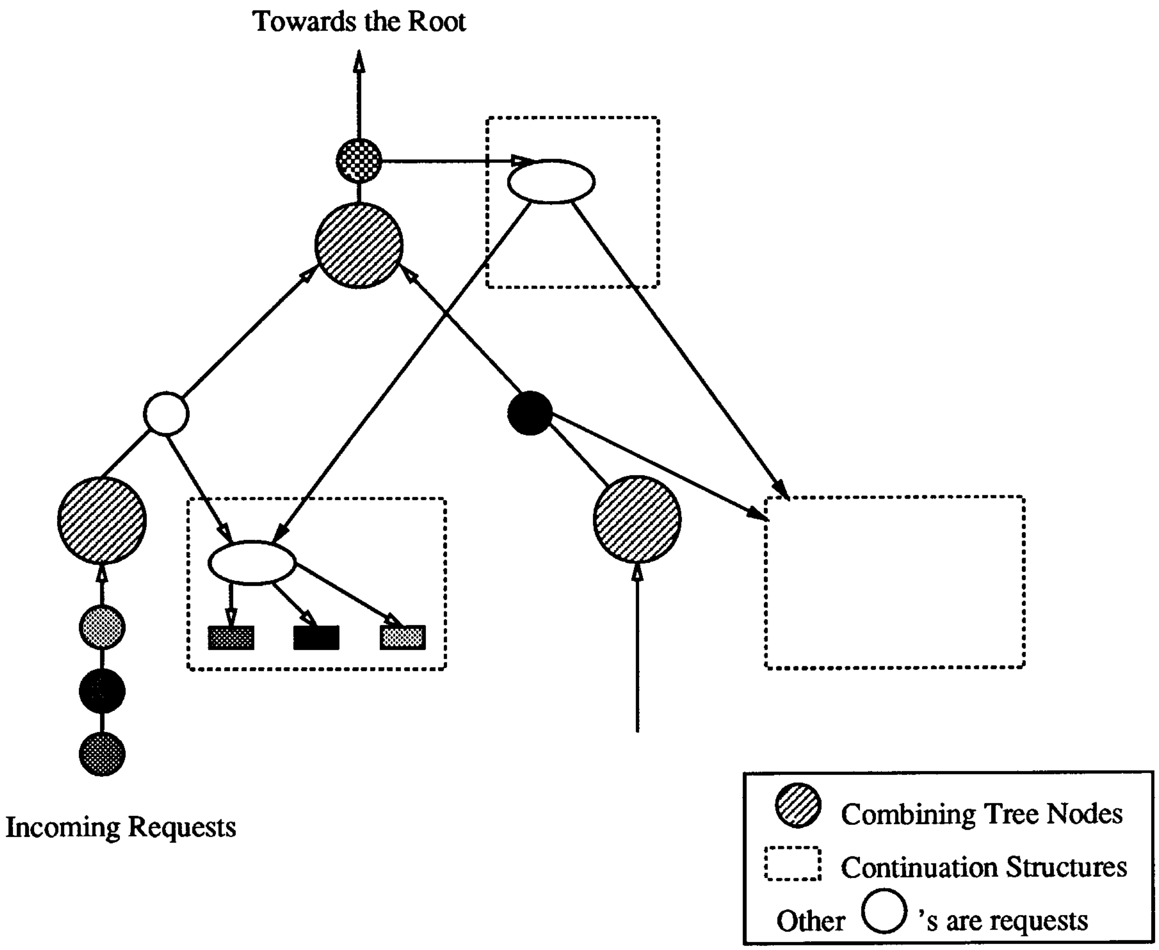


Figure A.10: Building Tree of Continuations in Dynamic Combining Tree

```
;;
;; continuation_struct
;;      used for pending requests in combining trees
;;      holds the continuation, and the number of indices required for
;;      this list
;;
(load "integer.ca")  ;; handle termination, which is a 0

(class continuation_struct left weight right :no_reader_writer
        (parameters ileft iweight iright)
        (initial
         (set_left self ileft)
         (set_weight self iweight)
         (set_right self iright)))

(method continuation_struct reply (val)
        (let
            ((continuation (left self)))
          (do (reply continuation val))
          (if (neq 0 (right self))
              (reply (right self) (+ val (weight self))))))
```

Figure A.11: Continuation Struct code maps a range of indices to the tree of request continuations. The tree structure determines which request gets which index.

The fifo counter abstraction is complicated because it manages the allocation and reclamation of array locations. It allocates indices in sets of contiguous numbers. For example an in_increment message with an argument of 20 would cause 20 indices for storing values (enqueueing) to be allocated. The allocation of indices is shown in Figure A.12. The fifo counter also reclaims array locations so that they can be reused. The array is managed as a *circular* FIFO. This reclamation occurs in chunks of the entire array. Whenever the amount of known free storage goes below a threshold, a time_to_reclaim message is sent to the fifo counter. This causes reset messages to be sent to the array fragment that we would like to reclaim. The code for reclaiming storage is in Figure A.13.

```
;;
;; fifo_counter -- an abstraction that implements the pointer operations
;;                 for a fifo queue
;;   Requirements: isize must be at least 8

;; Computations based on an index system that is monotonic and rooted
;; at the value of RECLAIM.  The circular buffer is cut here and all
;; comparisons are made in this index system.
;;
(class fifo_counter in in_since_reclaim out out_since_reclaim
       reclaim reclaim_lock size array :no_reader_writer
        (parameters isize pvect_array)
        (initial ...set up the state...))
;;
;;  in-val = if (in >= reclaim) in - size
;;              else in
;;
;;  overflow-flag = (>= (+ in-val val) reclaim)
;;  (if true, then we're going to overflow)
;;
(method fifo_counter in_increment (val)
        (seq (if (> (in self) (size self))
                 (set_in self (mod (in self) (size self))))
             (let ((in_val (if (>= (in self) (reclaim self))
                               (- (in self) (size self))
                             (in self))))
               (seq (set_in_since_reclaim self (+ (in_since_reclaim self) val))
                    (if (> (in_since_reclaim self) (/ (size self) 4))
                        (seq (set_in_since_reclaim self (- (in_since_reclaim self)
                                                           (/ (size self) 4)))
                             (do (time_to_reclaim self))))
                    (if (< (+ in_val val) (reclaim self))
                        (seq (reply (in self))  ;; not in overflow situation
                             (set_in self (+ (in self) val)))
                      ;; in overflow, spin on this message
                      (forward (in_increment self val)))))))
;;
;; out_increment is just like in_increment
(method fifo_counter out_increment (val)
        ... same as above...)
```

Figure A.12: A FIFO counter object that manages the array as a circular FIFO. Part I of the code.

```
;;
;;  Determination of whether or not to reclaim
;;
;;  base = reclaim - size
;;  in-val = if (in > reclaim) in - size
;;              else in
;;
;;  out-val = if (out > reclaim) out - size
;;              else out

;;;; now indices are linearized
;;
;;  clearspace = min(in-val,out-val) - base  (base is negative)
;;  if (clearspace > (size/4)) clear(reclaim, size/4)
;;
(method fifo_counter time_to_reclaim () :no_exclusion
    (seq (if (not (reclaim_lock self))
             (let ((in_val (if (>= (in self) (reclaim self))
                               (- (in self) (size self))
                             (in self)))
                   (out_val (if (>= (out self) (reclaim self))
                                (- (out self) (size self))
                              (out self)))
                   (base (- (reclaim self) (size self))))
                (let ((free_able_space (- (min in_val out_val) base))
                      (free_threshold (/ (size self) 4)))
                   (if (>= free_able_space free_threshold)
                       (reclaim_storage self free_threshold)))))
         (reply done)))

(method fifo_counter reclaim_storage (amount) :no_exclusion
        (if (reclaim_lock self) (reply done)
          (seq (set_reclaim_lock self 1)
               (reset (array self) (reclaim self) amount)
               (set_reclaim self (mod (+ (reclaim self) amount) (size self)))
               (set_reclaim_lock self 0)
               (forward (time_to_reclaim self)))))
```

Figure A.13: A FIFO counter object's reclamation code. Part II of the code.

The last part of the queue is the synchronizing array. This array aggregate partitions its state one element per representative. **at** and **atput** requests are forwarded to the appropriate representative. They are used to read and modify the state of the array. The synchronization and reclamation is implemented independently by each representative. The code for a synchronizing array aggregate is shown in Figure A.14. **Reset** is used to reset ranges of locations in the array.

```
;;  Each location contains presence bits with states:
;;  -1 :: one pending request, no value, 0 :: empty, no value
;;   1 :: full, value,                  10 :: used, ready for reset
;;
(aggregate pvector_presence state pbit :no_reader_writer
           (parameters size)
           (initial size (init_pvector_presence (sibling group 0) 0)))
(handler pvector_presence init_pvector_presence (pbit_val)
        ... initialize all the pbits to pbit_val and states to 0 ...)

(handler pvector_presence at (index)
         (forward (internal_at (sibling group (mod index groupsize)))))
(handler pvector_presence atput (value index)
         (forward (internal_atput (sibling group (mod index groupsize)) value)))

;; pbit must be 0 or 1  (empty or present)
(handler pvector_presence internal_at ()
         (if (= (pbit self) 0) (seq (set_state self requester)
                                    (set_pbit self -1))  ;; waiting state
           (seq (reply (state self))
                (set_pbit self 10))))  ;; prepare for the reset

;; pbit must be 0 or -1 (empty or pending req)
(handler pvector_presence internal_atput (value)
         (if (= (pbit self) 0) (seq (set_state self value)
                                    (set_pbit self 1)
                                    (reply done_atput))
           (seq (do (reply (state self) value))
                (set_pbit self 10)  ;; prepare for the reset
                (reply done_atput))))

(handler pvector_presence reset (startval nr_elts)
    (forward (internal_reset (sibling group startval)
                             startval (mod (+ nr_elts startval) groupsize))))
(handler pvector_presence internal_reset (startval endval)
    (if (= (pbit self) 10)
        (seq (set_pbit self 0)
             (let ((nextindex (mod (+ myindex 1) groupsize)))
                (if (!= nextindex endval)
                    (forward (internal_reset (sibling group nextindex)
                                             nextindex endval))
                  (reply finito)))
             (forward (internal_reset self startval endval)))))
```

Figure A.14: An Array Aggregate called pvector that synchronizes readers and writers. Locations are write-once, read-once and must be reset.

```
;; ... load all the other parts of the queue ...
;;
(load "fifo_counter.ca")
(load "pvector_presence.ca")
(load "dcombtree.ca")
(load "enqueuer.ca")
(load "dequeuer.ca")

;;
;; some globals for testing the queue
;;
(global nr_q_reps 64)
(global arr_size 1024)
(global nr_enqdeq 64)
(global ops_per_enqdeq 64)

(method osystem initial_message ()
    (let ((the_q (new parallel_q (global nr_q_reps) (global arr_size)))
          (ops_per_worker (global ops_per_enqdeq)))
      (seq (forall index from 0 below (global nr_enqdeq)
                   (let ((enq (new enqueuer (* index ops_per_worker)
                                   (* (+ index 1) ops_per_worker)
                                   the_q))
                         (deq (new dequeuer ops_per_worker the_q)))
                     (conc (do (go enq))
                           (do (go deq))
                           (do (starting_an_enq_deq_pair 50:0)))))
           (reply done_initial_message))))
```

Figure A.15: Some code to use the parallel queue.

Finally, we can put all of these abstractions together to build a parallel queue. The program that did this might look as is shown in Figure A.15. This program loads all of the abstractions we have defined so far. Then, it instantiates a parallel FIFO queue and a number of workers. Each of these workers performs a number of operations against the queue.

Vita

Andrew Andai Chien was born in Mt. Kisco, New York on March 6, 1964, son of Robert Tienwen and Sophie Loh Chien. He attended University High School in Urbana, Illinois from 1977 to 1980. After graduating from high school, Andrew studied at the Massachusetts Institute of Technology and received an S. B. in Electrical Engineering in 1984 with minor in Chinese Studies. He chose to continue his studies at M.I.T. in the area of parallel processing. He received an S.M. in Computer Science in 1987. His Master's Thesis title was "Congestion Control in Routing Networks."

Andrew completed his Sc.D. degree in June of 1990. As part of that program, he also completed a minor in Political Science – Governmental Systems. He is currently a faculty member in the Department of Computer Science at the University of Illinois in Urbana.

CS-TR Scanning Project
Document Control Form

Date : 6/22/95

Report # AI-TR-1248

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
□ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR)   □ Technical Memo (TM)
□ Other:

Document Information

Number of pages: 169 (177-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
□ Double-sided

Intended to be printed as :
□ Single-sided or
☒ Double-sided

Print type:
□ Typewriter   □ Offset Press   ☒ Laser Print
□ InkJet Printer   □ Unknown   □ Other:

Check each if included with document:

☒ DOD Form (2)   □ Funding Agent Form   ☒ Cover Page
☒ Spine   □ Printers Notes   □ Photo negatives
□ Other:

Page Data:

Blank Pages(by page number):

Photographs/Tonal Material (by page number): ii, iv, vi, viii

Other (note description/page number):

Description :   Page Number:

IMAGE MAP (1-18) PAGES #'ED i - xviii (INCLUDING TITLE PAGE
(19-169) PAGES #'ED 1-150 AND UN#'ED AUTOBI,
(170-174) SCANCONTROL, COVER, SPINE, DOD (2)
(175-177) TRGT'S (3)

Scanning Agent Signoff:
Date Received: 6/22/95   Date Scanned: 6/27/95   Date Returned: 6/29/95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
AI-TR 1248

2. GOVT ACCESSION NO.
A228 712

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*
Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines

5. TYPE OF REPORT & PERIOD COVERED
technical report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Andrew Andai Chien

8. CONTRACT OR GRANT NUMBER(s)
N00014-88-K-0738
N00014-87-K-0825

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, Virginia 22209

12. REPORT DATE
July 1990

13. NUMBER OF PAGES
172

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*
Office of Naval Research
Information Systems
Arlington, Virginia 22217

15. SECURITY CLASS. *(of this report)*
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

parallel programming	object-oriented
fine-grained	parallel processing
language	message-passing
massively parallel	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Fine-grained parallel machines exhibit great potential for high speed computation. Several machines such as the J-machine [42] and the Mosaic C [13] are projected to provide peak performance of 100's of billions of instructions per second in an air-cooled computer that fits in a cubic meter. While the hardware technology to build fine-grained machines is available, significant challenges remain in developing software systems to harness their computational power.

(con't on back)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(block 20 con't)

To program massively-concurrent MIMD machines, programmers need tools for managing complexity. One important tool used in the sequential programming world is hierarchies of abstractions. Unfortunately, most concurrent object-oriented languages construct hierarchical abstractions from objects that serialize – serializing the abstractions. In machines with tens of thousands of processors, this unnecessary serialization can cause significant loss of concurrency. Tools for managing concurrency should not restrict or reduce program concurrency.

Concurrent Aggregates (CA) provides multiple-access data abstraction tools, *Aggregates*, for managing program complexity. These tools can be used to implement abstractions with virtually unlimited potential for concurrency. Such tools allow programmers to modularize programs without reducing concurrency. These aggregates can be composed hierarchies of abstractions, allowing the structuring of a program to be highly concurrent at all levels.

In this thesis I describe the design and motivation of the Concurrent Aggregates language. I have used this language to construct a number of application programs. I present this programming experience with Concurrent Aggregates. Multi-access data abstractions are found to be useful for structuring applications without restricting concurrency. The tools provided to build such abstractions in CA allow the expression of a number of different styles of concurrency, including both data and control parallelism. I also present experience with the implementation of Concurrent Aggregates. A detailed evaluation of the efficiency of Concurrent Aggregates and the potential for further improving that efficiency is presented.