A Model for Parallel Programming

by

Lawrence A. Crowl

Thesis Proposal

for the Degree

Doctor of Philosophy

Supervised by Thomas J. LeBlanc

Department of Computer Science

University of Rochester

Rochester, New York

Friday 15 September 1995

Abstract

Current parallel programming languages support only a narrow range of programming styles; force programmers to bind the specification of parallelism early in program development; and limit the class of architectures for which they are effective. This proposal presents a model for parallel programming based on uniform encapsulation and control, concurrent operations on objects, synchronous invocation with early reply, references to code in an environment, and a value model of parameters and variables. This simple, coherent model enables programmers to use different styles of programming and to bind parallelism late in program development. In addition, the model has efficient implementations on a wide range of multiprocessor architectures.

Table of Contents

1 Introduction

Parallel computation brings new challenges to programming language design. Current approaches tend to provide separate facilities for parallel programming on top of a base sequential model of programming. These facilities are typically a distinct notion of process, monitor constructs, messages, *etc.* Languages with these facilities include Ada [U. S. DoD, 1983], CSP [Hoare, 1978], Distributed Processes [Brinch Hansen, 1978], and StarMod [Cook, 1980]. These languages provide concise expression of sufficient parallelism for distributed computation, but not for parallel computation. The dichotomy between sequential and parallel aspects of programming limits the effectiveness of programmers in exploiting multiprocessor hardware. Parallelism is not available at all levels within a program based on the dual model. This research presents a model of programming that unifies the sequential and parallel aspects into a single, simple, and powerful model.

1.1 Execution Environments

Most current parallel programming languages are oriented towards a specific execution environment. Many of these are oriented towards distributed computation. Such languages include Argus [Liskov and Scheifler, 1983], Emerald [Black *et al.*, 1986a; Black *et al.*, 1986b; Hutchinson, 1987], and SR [Andrews *et al.*, 1986]. Distributed and parallel programming are similar in that they both must address parallelism, but they place significantly different demands on the programming language. Parallel programming distinguishes itself from distributed programming by utilizing data parallelism as well as functional parallelism.

1.1.1 Distributed Programming

Distributed programs assume an environment in which the actual parallelism does not exceed the number of relatively independent functions to be performed. Relatively independent functions include file managers, transaction managers, screen managers, and the like. In this environment, programs can achieve sufficient parallelism by decomposition along functional lines. Such decomposition does not capture large amounts of parallelism.

The distributed execution environment provides actual parallelism on separate processors communicating over a network. In this environment, communication between two different processors (such as a remote procedure call) costs two orders of magnitude more than communication within a processor (such as a direct procedure call). Many distributed programming languages make this communication cost visible within the language under the assumption that programmers will manage visible costs more effectively than invisible costs, for example, PLITS [Feldman, 1979]. As a result, distributed programs communicate less frequently.

In distributed programming, many objects are very long lived. In order to prevent recompiling an entire distributed system because of a change in one portion, distributed languages often make concurrent objects largely independent. This in turn requires a typing structure in which new types and implementations may enter the system at run time. In addition, the long lived nature of distributed objects makes hardware failure, reliability, autonomy, and error handling significant concerns.

In short, distributed programming is characterized by functional parallelism among very long lived, independent objects in an environment providing of two widely disparate communication costs.

1.1.2 Parallel Programming

Parallel programs tend to assume an environment in which there is a great deal of parallelism (a hundred processors), and a small collection of highly coordinated functions to be performed on a large collection of data. Parallel programs address large problems, but may not constitute large systems. In this environment, decomposing along functional lines does not use enough of the available parallelism, so programs must parallelize over the data available, in addition to possibly parallelizing along functional lines.

Parallel execution environments tend to be more tightly coupled than distributed environments. Communication costs between processors are generally much lower, and may have a large set of costs depending on the relative positions of the processors. In addition, the relative disparity in local and remote communication costs is lower, which encourages more frequent communication within programs. Therefore, parallel languages must provide inexpensive mechanisms, both notationally and computationally, for process communication.

In parallel programming, objects are short lived. The language processor should be able to take advantage of the short lifespan in optimizing performance. Languages which force excessive independence between objects may incur a high overhead because the language processor cannot take advantage of static information. This is more important in parallel programming because the granularity of parallelism is typically much finer than in distributed programming. Types may be bound to objects earlier because programmers are more willing to recompile a parallel program than a distributed system. Likewise, reliability is less of a concern because the program may be re-executed. There is no need to repair or recover computations.

In summary, parallel programming is characterized by data parallelism (in addition to functional parallelism) among relatively short-lived objects in an environment providing of many different communication costs. Object reliability and independence are less needed than in distributed programming.

1.2 The Parallel Programming Problem

Current parallel programming languages are limited in their ability to fully exploit parallelism because they lack flexibility. They support a single style of programming and force programmers to bind the location parallelism too early in program development.

1.2.1 Limited Programming Styles

Many parallel programming languages are available, ranging from very loosely coupled message-based languages such as PLITS [Feldman, 1979] to tightly-coupled shared-memory languages such as MultiLisp [Halstead, 1986]. Most of these systems support only a single programming style. Different programming problems are often more naturally expressed in very different programming styles. These different problems often appear within a single application. The programmer must either use a single style even when unsuited, or use multiple programming systems.

The use of a single style unsuited to the problem at hand makes the program difficult to produce. For instance, the programmer may be forced to implement a filtering mechanism in a shared-memory model when it is most naturally expressed in a message-passing model.

The multiple programming systems approach means that programmers must deal with several programming models, all of which have different mechanisms and constraints. For example, programmers using the SMP system [LeBlanc *et al.*, 1986] must contend with procedural, shared-memory communication within a process; with data-flow, message-based communication between processes; and with transaction, file-based communication between programs. In this example, there are no less than three models of data access. This represents a considerable mental load because programmers must learn and remember the subtle interactions between different models. This burden is non-productive because it does not increase the set problem solving tools available to the programmer.

More important, managing the interface between different mechanisms in an application represents a considerable portion of the coding effort [Atkinson *et al.*, 1983]. For example, many programs spend a substantial amount of code on managing files. Under Chrysalis on the Butterfly [BBN, 1985a; BBN, 1985b], one continuing source of errors is the improper management of Chrysalis objects due to the considerable amount of interface details to manage.

An ideal programming model should support a wide range of programming styles from shared memory to message passing, and a wide range of sharing granularity from integers to large data structures, while providing a single, simple conceptual framework.

1.2.2 Early Binding of Parallelism

Many current programming systems provide two mechanisms for encapsulation, one for the parallel portions of the program, and one for the sequential portions within a parallel portion. This approach forces the programmer to bind the location and granularity of parallelism early in program development. This presents a problem because the appropriate choices for where to place the parallelism may not be known until after monitoring the performance of the actual program. Choosing the granularity and location of parallelism should be more a part of the optimization effort rather than the algorithmic effort.

Programmers may not know ahead of time which mechanism is appropriate for a given abstraction. Under this situation, programmers may choose the higher-cost, more general mechanism [Greif *et al.*, 1986]. If a programmer guesses incorrectly, the abstraction must be recoded and reintegrated into the environment. General purpose abstractions must often be provided in both mechanisms in order to provide programmers with the incentive to use them.

The reasons for a single abstraction mechanism also apply to control as well as data. Control mechanisms must be powerful enough so that critical control abstractions may execute either sequentially or in parallel. This allows the programmer to tune the program to a specific architecture with minimal changes to the program source.

A programming model that allows the programmer to delay the binding of actual parallelism must be based on abstraction mechanisms that apply to both sequential and parallel execution. Abstraction mechanisms are more important in parallel programming because abstraction affects performance.

One of the consequences of programming systems that bind parallelism to special language constructs is that many parallel programming systems work well on only one architecture because the underlying model assumes a specific, limited hardware configuration. The appropriate architecture is embedded in the level at which the language presents parallelism.

1.3 Focus of Thesis

This thesis addresses parallel programming, as opposed to distributed programming. No aspect of the proposed model excludes its further development in support of distributed programming. Indeed, the model very naturally extends into distributed programming. However, this thesis will not address the distribution aspects.

This thesis addresses parallel programming only in the context of imperative programming. Other approaches include functional [Backus, 1978; Hughes, 1984], dataflow [Wadge and Ashcroft, 1985], declarative [Baldwin, 1987; Baldwin and Quiroz, 1987], and connectionist [Blelloch, 1986] models.

The proposed programming model is a language model as opposed to a set of operating system, or library communication primitives because only a language approach has the potential for a uniform programming model, which in turn enables the effective late binding of parallelism and independence from a specific architecture. A previous approach at combining a multiprocessor operating system with a sequential programming language was not completely successful [Crowl, 1986; Crowl, 1987; Crowl, 1988]. The language model is object-oriented because objects provide a clear notion of the destination or target of an operation or message, which defines communication patterns and enables communication to scale with abstraction.

Our hardware model consists of a loose network of tightly-coupled MIMD multiprocessors. We assume the multiprocessors have non-uniform access to shared memory, *e.g.* the BBN Butterfly [BBN, 1985a]. This model is appropriate because it provides a range of access times to concurrent data, from local memory access through remote memory access and intra-machine messaging up to inter-machine messaging. A parallel programming model that uniformly addresses all levels of such a hardware model, necessarily provides some independence from specific hardware configurations.

We assume the programming environment for the proposed model includes a library of general-purpose abstractions. This provides a repository for the hard work in parallel algorithm development and debugging. In addition, we assume the programming environment includes debugging facilities capable of effectively providing repeatable execution and performance monitoring of highly parallel programs. Such environments are not generally available, but are the focus of current research [LeBlanc and Mellor-Crummey, 1987].

1.4 Statement of Thesis

A model of parallel programming consisting of uniform encapsulation and control, concurrent operations on objects, synchronous invocation with early reply, references to code within an environment, and a value model of parameters and variables enables the programmer to use multiple styles of programming to specify extensive concurrency, and to bind the actual parallelism late in program development. It also enables efficient implementation on a wide range of architectures.

A single program may have several programming problems, each suited to a different programming style. Current imperative programming languages support only one of three programming styles based on a communication model using loosely coupled messages, remote procedure calls, or shared memory. Given a task with problems appropriate to different styles, a programmer must chose inappropriate styles for some problems, or attempt to mix programming systems. Neither is an attractive solution. This thesis presents a programming model based on *concurrent operations* on objects with an *early reply*, which allows programmers to use multiple styles of programming within a single computational framework.

Programming languages that provide parallelism with separate and distinct mechanisms for encapsulation. force the programmer to bind parallelism too early, which inhibits parallelism. This thesis proposes a model based on *uniform encapsulation*, which provides abstraction independently of parallelism. Likewise, separate mechanisms for sequential and parallel control also forces early binding of parallelism. Control constructs apply uniformly to both sequential and parallel control.

The proposed model provides communication via *operation invocation.* This, coupled with a *value model* of variables and parameters, allows communication to scale with abstraction. Uniform encapsulation and scalable communication allows the programmer and language processor to exploit parallelism at many levels within a program. Because parallelism may be exploited at many levels, some programs may execute efficiently on many different architectures by rebinding the actual parallelism to different levels within the program.

1.5 Outline of the Proposal

Chapter 2 presents a set of desirable capabilities for parallel programming language that enable programmers to effectively exploit the hardware model. These capabilities include uniform encapsulation, sender-based concurrency, receiver-based concurrency, uniform control, and a value orientation.

Chapter 3 discusses existing programming systems in the context of these capabilities and points out their inadequacies. While many languages provide some subset of the desired capabilities, none provide the complete set.

Chapter 4 presents the proposed language model and indicates how it provides the desired capabilities. The model features include a single encapsulation mechanism, value-based variables and parameters, operation variables, references to objects, references to object and operation pairs, concurrent operations, early replies, and code blocks (or closures).

Chapter 5 provides some example programs in the proposed model. These include message passing, various message buffers, shared memory, Warshal's transitive closure algorithm, combiners, and a relaxation network.

Chapter 6 outlines a set of implementation strategies. These strategies address passing parameters, optimizing the early reply, taking advantage of locality, scheduling to minimize execution and run-time state, and recognizing frequent special cases.

Chapter 7 presents the thesis research plan. Issues include support for multiple styles of programming, multiple levels of parallelism within programs, effective use of multiple architectures, a prototype implementation, and a schedule.

2 Desired Capabilities

This chapter presents desired capabilities for a parallel programming system. The capabilities fall into five categories: uniform encapsulation, sender concurrency, receiver concurrency, uniform control, and a value model of data.

2.1 Uniform Encapsulation

A language provides a uniform encapsulation when all program objects, from primitive language objects to large user abstractions, use the same set of encapsulation and abstraction mechanisms, regardless of the intended or actual concurrency within the abstraction. Uniform encapsulation enables the programmer to bind the actual parallelism within an explicitly parallel program late by binding the implementation of an abstraction late. This late binding is necessary under explicit parallelism because it allows the programmer to shield changes in parallelism at one level from those at another.

To make uniform encapsulation effective, the mechanisms for managing the abstractions must be sufficient to handle dynamic needs and the mehcanisms for operating on the abstractions must be convenient.

2.1.1 Single Abstraction Mechanism

Many parallel programming systems have two mechanisms for abstraction, one for global and concurrent abstractions and one for local and sequential abstractions. This dual mechanism splits the programming environment into two qualitatively different models of interaction. When programming an abstraction, a programmer must decide early upon the abstraction mechanism. The appropriate mechanism is often not known until late in program development. This encourages the programmer to use the more global mechanism even when it may not be appropriate [Greif *et al.*, 1986]. In addition, in some programming systems, programmers using an abstraction must know in which mechanism the abstraction is implemented, because the code to access the abstractions is different under the two mechanism. Because of this early binding of abstraction to concurrency, rebinding parallelism while optimizing a program requires considerable effort.

This issue is less important in a distributed system than in a multiprocessor system because the performance difference between local operations and remote operations is so large that programmers tend to make distributed objects large so as to minimize interactions between them. In a multiprocessor, communication costs are low enough that the proper distribution and level of implementation for objects is not obvious before performance experiments on completed code. To facilitate rebinding of parallelism, a parallel programming system should provide a single, uniform model for encapsulation of abstractions and the operations on them. This model should be orthogonal to the placement of the actual parallelism in a program.

2.1.2 Dynamic Data Management

The appropriate partitioning of data in a parallel program is generally not known until execution. In order to handle this problem, the model must allow dynamic management of data. Data parallelism over dynamic data requires the dynamic allocation of parallel data objects. However, many monitor based systems do not allow for the dynamic allocation of monitors. Essentially, the monitor locks code and not data. Likewise, the partitioning of work among parallel data objects requires that the model provide indirect, or dynamic, references to data. Static names for static objects, such as monitors, require the programmer to partition the data *a priori*, and are therefore not sufficient.

2.1.3 Synchronous Invocation

Current imperative programming languages generally rely on synchronous invocations. The invoker of an operation waits on its completion. This notion of waiting on completion becomes is important under parallel programming because explicitly parallel programs require the programmer to handle time and synchronization. Programmers often wish to immediately wait on a reply value, so systems with only asynchronous invocations force programmers to pass explicit reply destinations and wait on arrival of the reply.

Synchronous invocation is becoming more prevalent in distributed systems as well as sequential systems. The evolution of the asynchronous, message-based ECLU [Liskov, 1979] into the synchronous, atomic-transaction-based Argus [Liskov and Scheifler, 1983] provides evidence for this trend. When waiting on the reply, the programmer does not want to deal with other messages that may be arriving at the client, so these systems must provide complicated mechanisms to filter the incoming messages. In PLITS [Feldman, 1979], there is no synchronous invocation. Programmers must allocate a transaction key for a conversation and then explicitly wait on the reply under that transaction key.

Waiting on the arrival of a reply is itself a synchronous operation, so even systems based on asynchronous invocation usually supply synchronous invocation for primitive operations. A system may be completely asynchronous by explicitly passing *continuations* as parameters to operations. Hewitt's Actor system [Hewitt, 1977; Hewitt and Atkinson, 1977; Agha, 1986b; Agha, 1986a] uses this model. In Actors, reply values are not returned to the caller. Rather, the callee passes a continuation as an additional argument. The invokee sends the reply to the continuation, which is code within the environment of the invoker. Because reply values are heavily used in most programs, this approach requires extensive use of continuations, which is clumsy if continuations are explicit and is more expensive than just waiting for the reply. The continuation model necessitates a general-purpose implementation because the active life of a continuation and its environment usually cannot be determined from the static program structure. This forces implementations to use general heap-based allocation of activation records instead of the more efficient stack-based allocation.

2.1.4 Implicit Reply

Many computations involve accepting a request to perform some work and then returning the result of that work. If the programmer must explicitly send results as independent communication, the burden on the programmer is high, both for the caller and the callee. The reply destination for the result of an operation should be implicit.

Message-based systems without implicit reply invariably involve the caller passing its identity in the message to the callee, which then uses that identity to send a reply message to the client. In PLITS, a callee obtains the identity of the caller and the transaction key from the message, and then explicitly sends to the caller.

2.2 Sender Concurrency

A programming model for parallel programming must effective specification of extensive concurrency among different objects. Denying the programmer the ability to conveniently express concurrency among different objects reduces the amount of parallelism programmers will specify.

2.2.1 Dynamic Process Creation

Because parallel programming involves data parallelism, and the amount of data and work are generally not known until execution, the programming model must support dynamic process creation. The lack of dynamic process creation can inhibit effective programming of client/server relationships in those languages with synchronous communication mechanisms [Liskov *et al.*, 1984].

2.2.2 Asynchronous Invocation

Often, a program can compute two or more values simultaneously. For example, within a search tree, a programmer can apply two different constraint mechanisms concurrently and then combine their results to form the actual constraint. In order to express this, the model must have a mechanism for starting an operation and then later in execution, request the value. In addition, the mechanism must be syntactically inexpensive, or programmers will not use it. More important, the caller's use of asynchronous invocation must be transparent to the callee. Otherwise, the programmer must provide and name two different forms for each operation.

2.2.3 Concurrent Invocation

Often, many invocations can start simultaneously. For example, each object in an array may be initialized in parallel. To express this, the model must provide a means to initiate several invocations. This capability is implied by asynchronous invocation because programmers may write a loop, but languages may provide concurrent invocation without asynchronous invocation. Several systems provide concurrent invocation, but only over limited structures such as integer ranges. These generally provide no mechanism for capturing return values. For example, the Uniform System [BBN, 1985c] provides a library of routines that concurrently execute a given procedure for each integer in a given range. While these mechanisms help, they are insufficient because they do not allow programmers to use more general, or user-implemented structures. Again, concurrent invocations must be transparent to the callee.

2.3 Receiver Concurrency

Section 2.1 argues that all abstractions must be uniformly encapsulated. Since many abstractions will necessarily be large, the programming language must enable extensive concurrency within a single abstraction and between the abstraction and its callers. Any limitation on the potential concurrency within the encapsulation mechanism will magnify itself at each level of abstraction. Because the appropriate place for maximum parallelism may not be known until late in program development, programmers must have the ability to change the implementation of an abstraction and increase the amount of available concurrency while not affecting the clients. Programmers can change the implementation of general-purpose abstractions by selecting a different implementation from a library.

2.3.1 Concurrent Operations

An *operation* is a request to an object that computes a *reply*. Shared-memory multiprocessors support concurrent operations on a single object. To fully exploit the hardware model, the programming model must allow multiple operations to execute concurrently within a single object. Programming systems for shared-memory multiprocessors that do not support concurrent operations violate Parnas's concept of transparency [Parnas and Siewiorek, 1975]. This multiple active invocation capability is used to good advantage in the implementation of concurrently accessible data structures, *e.g.* [Ellis, 1982; Ellis, 1985]. In addition, Bukys [1986] indicates that the system must be written in as parallel manner as possible. Otherwise, the applications will be serialized on access to critical system resources. This conclusion may seem obvious, but some things, such as the memory allocator in the Uniform System [BBN, 1985c] seem non-critical in development but turn out to be critical for later applications. A language that forced serial operations would prevent replacing the memory allocator with a more parallel version. A single active invocation language essentially forces serialization on system objects.

2.3.2 Implicit Dispatching

Several programming models allow concurrent operations within an object, but only after explicit dispatching. In explicit dispatching, the receiver explicitly indicates when the processing of a request begins. This explicit dispatching introduces a weak form of serialization in that the resources devoted to dispatching processes are limited. This serialization for dispatching limits the amount of potential parallelism, increases the latency of operations, and introduces unnecessary costs in many cases. Often, object semantics fall naturally towards implicit dispatching, so explicit dispatching would require excess programmer effort. Under implicit dispatching, the request itself dispatches the operation for execution.

Implicit dispatching allows better code generation because the synchronization will be explicit in the code. There is no need to constantly use implicit general-purpose queueing mechanisms, as is the case with Ada [U. S. DoD, 1983]. This allows the system to compile in-line those objects with minimal synchronization requirements and known implementations. For example, an object for gathering usage statistics could do bin selection in-line without synchronization and then atomically increment the count in that bin. Synchronization is delayed until updating the bin count, which may be done with a simple machine instruction.

To avoid excessive serialization on shared-memory multiprocessors, a parallel programming model should implicitly provide a new, concurrent thread of control within the object. Optimizations, such as borrowing the caller's thread of control for completely synchronous invocations, can control the cost of this model.

Implicit dispatching and concurrent operations together support a shared memory style of programming.

2.3.3 Asynchronous Processing

Many abstractions can solve the information needs of their clients long before all of the associated computations are complete. A mechanism for processing an operation asynchronously to the caller reduces the non-essential synchronization between processes. Allowing the programmer of an object to minimize synchronization with the external world increases the potential concurrency within a program.

The use of asynchronous processing must be transparent to the caller, otherwise programmers will tend to avoid it.

2.4 Uniform Control

Parallel programming distinguishes itself from distributed programming by utilizing data parallelism as well as functional parallelism. To support this, a parallel programming model must provide mechanisms to control the partitioning of work. This means that the programmer must be able to reference work in addition to data.

The desire for a uniform encapsulation coupled with the need to reference work, implies that control constructs uniformly apply to parallel control as well as sequential control. Since the appropriate parallelization of control is generally dependent on user data structures, a programmer must be able to define new control structures as well as use primitive control structures. The issue of programmable control abstractions is not unique to parallel systems, they are common to programming in general.

2.4.1 Handles for Parameters

Control abstractions will often need to delay the actual invocation of an operation, or apply an operation to a number of objects. For example, one might wish to print each of a set of number numbers in a given format. In order to describe general-purpose mechanisms to perform such tasks, an intermediary must be able to handle parameters as a unit. Message based systems naturally provide this capability by referring to the message. RPC based systems do not provide a mechanism to refer to the set of parameters. Suitable changes would enable RPC systems to refer to the set of parameters.

2.4.2 Handles for Operations

A model that does not provide an indirect handle on operations (one that has no mechanism for non-literal reference to operations) has two consequences. First, it prevents the programmer from building general-purpose control abstractions that can control the timing and ordering of operations. Second, it inhibits the dynamic networking of communicating objects.

For instance, consider an abstraction to apply an operation to a number of objects. If the operation must be directly named at the point of call, it is not possible to build a general-purpose abstraction to perform such operations. In such abstractions the operation to be performed becomes data. Without mechanisms to support operations as data, the programmer must construct such abstractions on an *ad hoc* basis, which in turn inhibits the use of such constructs. Inhibiting these constructs limits the programmer's ability to exploit data parallelism. Another consequence of requiring the invoker to know the name of the receiver's operation name is that it is not possible to build general-purpose network monitoring intermediaries.

One of the advantages of message passing systems is that they may be connected into rich networks where the nature of intermediaries is not necessarily known in advance. Passing references to neighboring objects allows effective construction of networks. However, if the operations must be directly named, each sender in a network must know the name of the operation of the recipient. This requires the sender and receiver to agree on an operation name *a priori*. This implies that programmers may have to place in the communication path many intermediary objects whose sole purpose is to translate operation names. Network construction under these circumstances will be an *ad hoc* task.

2.4.3 Handles for Work in a Context

Often work needs to be done within the context of another executing operation. This operation's context is non-global. Passing a handle to an operation alone is not sufficient, because the operation generally loses access to the local context of the work. The most effective support of work within a context is represented by Smalltalk's *blocks* [Goldberg and Robson, 1983], or Lisp's *closures* [Steele, 1984]. Blocks are portions of code that have access to an operation's local variables, but may be executed by another object. This is similar to passing nested procedures in Pascal [Jensen and Wirth, 1975; Cooper, 1983], but with the added notational convenience of being defined in-line. Blocks provide Smalltalk with substantial power. For instance, in Smalltalk the programmer may define a tree that accepts a block and then executes the block for each element in the tree.

The need for blocks is even greater for parallel languages because of the need to provide many abstractions for generating parallelism. If there is no such control abstraction, each programmer must build *ad hoc* mechanisms for creating parallelism. This increases the cost of developing highly parallel programs and markedly increases the cost of changing the method of parallelism. This in turn inhibits the specification of extensive parallelism and especially data parallelism. For instance, without a handle for work within a context, users cannot define general purpose control abstractions, such as a parallel for-loop, without explicitly collecting references to the relevant portion of the environment.

2.5 Value Orientation

There are two major models of variables and parameters in imperative languages. The value model, in which variables contain values, is used by languages such as Fortran [ANSI, 1978], Algol [Nauer, 1963], and its derivatives. The reference model, in which variables refer to objects, is exemplified by languages such as Smalltalk [Goldberg and Robson, 1983] CLU [Liskov *et al.*, 1977], and Emerald [Black *et al.*, 1986a; Black *et al.*, 1986b; Hutchinson, 1987]. The desire for a uniform encapsulation mechanism implies that a parallel language must choose one model and stick with it. This includes variables for parallel abstractions, parameters, and variables local to an abstraction. Most imperative parallel languages provide a reference model for concurrent abstractions, but a value model for parameters and local variables. Examples include Ada [U. S. DoD, 1983] and SR [Andrews *et al.*, 1986].

The reference model has the attractive property of requiring only one type of variable. Variables can directly represent non-hierarchical, or cyclic, structures. On the other hand, the value model requires the introduction of a separate pointer variable to represent non-hierarchical structures.

The reference model introduces an "infinite regression" in its pattern of objects being composed of references to other objects. When does one get through the references to the real data? Under the reference model, there exists a canonical object representing each primitive value, such as the integer 3. These objects are immutable, meaning that no operation will change their value, so the implementation is free to copy their representations without actually referring to the canonical object. Note that under some systems, such as Emerald, the representation of an immutable object may change over time, so long as its abstract value does not.

Unfortunately, the reference model has some serious disadvantages in the context of parallel and distributed programming. These disadvantages include uncontrolled aliasing, remote arguments, object contention, and more expensive storage management. The proposed language model adopts the value model. Real programs must deal with values as well as references, so the choice of the variable model in a language represents a bias, and not an absolute choice. The bias of the value model is more appropriate for parallel programming.

2.5.1 Controlled Aliasing

The reference model naturally provides for extensive aliasing among objects. It is not generally possible to determine *a priori* when two variables will refer to the same object. The result is that the programmer and language system must assume that the variables may refer to the same object. This requires either that the programmer and system not attempt to operate on the two variables concurrently, or that the object explicitly controls asynchronous accesses to itself. Improperly managing the aliasing will introduce obscure and unexpected side effects that are difficult to debug. In addition, aliasing inhibits dependency detection.

In contrast, the value model ensures that each variable refers to a different object. Thus the programmer and the implementation are free to operate on two different variables concurrently. The value model allows compilers to effectively parallelize sequential code. The current research effort in parallelizing sequential programs [Ellis, 1986, Chapter 5], may aid in further parallelizing parallel programs if the parallel program adopts a value model of variables. This approach is likely to complement the coarser-grained parallelism that programmers provide.

2.5.2 Remote Arguments

The reference model implies heavy communication between machines as operations traverse back and forth across machine boundaries to reach objects passed as parameters. This potential inefficiency on distributed systems is such that Argus, which has an internal reference model based on CLU, uses an external value model. The inter-node cost of the reference model can be mitigated by moving argument objects to the node with the called object, as in Emerald, and by heavy use of immutable objects. However, the use of immutable objects is essentially explicit programmer use of the value model of parameters.

2.5.3 Object Contention

In multiprocessor systems, mutable objects may be a source of contention. Many programs will require access to the current states of objects, not to their potential future state. For example, in Gaussian elimination, the pivot row provides a considerable source of contention. In the reference model, the programmer has two options, to make the rows immutable or to explicitly make a copy of the pivot row.

In the case of making the rows immutable, the compiler is free to copy each argument row to the local node. Unfortunately, this involves increased dynamic allocation and deallocation of row objects in the normal maintenance of the matrix. Immutable rows cannot be updated in-place. The new value of the row must be computed in new storage. This is not the case with the value model because objects may be modified in place.

An explicit copy would also increase the amount of dynamic allocation and deallocation within the system. In addition, this explicit copy requires the programmer to provide additional code that will be suboptimal in the case where the row is being passed to another row on the same node. More importantly, the copy approach represents the value model, but with higher run-time and programmer costs. Implicit copy under the value model often requires less run-time support, and hence costs less.

2.5.4 Parameter Implementations

Because multiprocessors have a high degree of sharing and complicated mechanisms for sharing, the point at which it is more efficient to pass arguments by pointer or by copy changes more often than in distributed systems. Under the reference model, only immutable objects may take advantage of the most appropriate implementation. Mutable objects must be passed by pointer.

For instance, the Butterfly allows value parameter implementations including register transmission, local memory copy, remote memory copy, reference (assuming no one modifies the argument), and processor-to-processor messages. Small objects are almost always more efficiently passed by copy. Medium-sized objects are most efficiently passed by pointer when the target is on the same node, and by copy when the target is on a different node. For large objects that are infrequently accessed, it is more efficient to pass the objects by pointer. For large objects that are frequently accessed, it is more efficient to pass the objects by copy.

In order to delegate chosing the appropriate implementation for argument passing, the programmer must pass only immutable objects. The programmer essentially adopts a value model of communication within a language designed for the reference model. A value-based language can support this style of programming more effectively.

In short, a value model scales better and allows more potential implementations of parameter passing, which is important in multiprocessor systems.

2.5.5 The Object as a Reference Parameter

One criticism of a strictly value model of parameters in algorithmic languages, such as C [Kernighan and Ritchie, 1978], is that they do not allow argument modification. Programmers must pass explicit references to modify data. This criticism is less valid in object-oriented systems because the object being operated on is implicitly passed by reference. For example, the implicit reference to the object means that operations on large tables need only pass indexing values as parameters and may pass the table itself as the object being operated on.

2.5.6 Storage Management

Because references to an object under the reference model are spread freely, it is generally not possible to determine the lifetime of an object *a priori*. Variables under the reference model contain pointers. It is possible to determine the lifetime of the pointer, but since many variables may point to the same object, it is not possible to determine the lifetime of the object. On the other hand, the variables under the value model contains objects. This allows static determination of the lifetime of an object, because the lifetime of the variable containing it is known.

The undeterminable lifetime of objects under the reference model implies dynamic heap allocation and system-wide garbage collection. To perform garbage collection, the system must examine the entire set of references of the system to insure that no references to an object exist before deleting the object. This is possible on multiprocessors, but on very large or widely distributed systems, the number of possible locations for a reference is too large to be efficiently searched. This issue is important because institutions invest in parallel programming for speed. They are willing to purchase performance with engineering effort. The reference model inhibits performance.

2.6 Summary

Abstraction mechanisms are more important in parallel programming because abstraction and parallelism interact. Abstraction affects performance, so the abstraction mechanisms must not inhibit parallelism.

There are a number of desired capabilities that parallel programming languages must provide to allow programmers to effectively exploit a wide range of current multiprocessors.

Uniform encapsulation allows programmers to bind the actual parallelism within their programs late in program development. Uniform encapsulation requires dynamic management of objects, a single data abstraction mechanism, synchronous invocation on objects, and implicit reply destinations for operation results.

Sender concurrency allows the programmer to specify concurrency among different computations. It requires dynamic process creation, asynchronous invocation of objects (where the result is explicitly received later in the computation), and concurrent invocation on many objects.

Receiver concurrency allows the programmer to simultaneously serve many requests to a single abstraction. It requires concurrent operations on a single abstraction, implicit dispatching of operations, and processing of requests asynchronously to the caller.

Uniform control allows programmers to use control abstractions and delay the specification of actual parallelism within the control constructs. Uniform control requires handles for parameters as a unit, handles for operations, and handles for work within a context.

The value model allows more efficient, parallel access to data. Its uniform application requires that concurrent abstractions, parameters, and local variables all use a value model.

3 Previous Approaches

This chapter analyzes relevant parallel and distributed programming languages with respect to the capabilities presented in the previous chapter. A language provides a capability through built-in language features, through a simple combination of built-in features, or through implementation in a general-purpose user abstraction. The special-purpose, or *ad hoc*, implementation of these capabilities by the user is not sufficient because programmers must re-implement these capabilities for each use.

The languages discussed are ABCL/1 [Yonezawa *et al.*, 1986], Actors [Hewitt and Atkinson, 1977; Agha, 1986b; Agha, 1986a], Ada [U. S. DoD, 1983], Argus [Liskov and Scheifler, 1983], ConcurrentSmalltalk [Yokote and Tokoro, 1986], CSP [Hoare, 1978], Distributed Processes [Brinch Hansen, 1978], Emerald [Black *et al.*, 1986a; Black *et al.*, 1986b; Hutchinson, 1987], Lang's model [Lang, 1982], Lynx [Scott, 1987], Mesa [Lampson and Redell, 1980; Xerox, 1984], MultiLisp [Halstead, 1986], NIL [Strom and Yemini, 1983], Orient84/K [Ishikawa and Tokoro, 1986], PLITS [Feldman, 1979], Presto [Bershad *et al.*, 1987], Smalltalk [Goldberg and Robson, 1983], SR [Andrews *et al.*, 1986], and StarMod [Cook, 1980]. Table 3.1 presents a short summary of the capabilities that each language fully supports.

3.1 Uniform Encapsulation

No operating systems, and few languages provide for uniform encapsulation of data. Table 3.2 summarizes the support provided by the languages. The major weakness in the support of uniform encapsulation is the presence of two mechanisms, one for sequential objects and one for concurrent objects.

3.1.1 Single Abstraction Mechanism

Most languages provide two abstraction mechanisms, one for concurrent (or global) objects and one for sequential objects (local to other objects). In essence, the languages overload the concepts of parallelism and abstraction. The dual abstraction approach forces programmers to manage the interface between the internal world and the external world. In addition, it inhibits the language processor in choosing the appropriate level of parallelism. The conversion of an object from a local abstraction to a global

	uniform encapsulation	sender concurrency	receiver concurrency	uniform control	value model
ABCL/1		$\surd$			
Actors	$\surd$	$\surd$			
Ada					
Argus					
ConcurrentSmalltalk		$\surd$			
CSP					
Distributed Processes					
Emerald	$\surd$		$\surd$		
Lang's model					
Lynx					
Mesa		$\surd$	$\surd$		
MultiLisp	$\surd$	$\surd$	$\surd$	$\surd$	
NIL					
Orient84/K		$\surd$			
PLITS		$\surd$			
Presto					
Smalltalk	$\surd$	$\surd$	$\surd$		
SR			$\surd$		
StarMod					

Table 3.1: Language Capability Summary

abstraction is often non-trivial, which in turn inhibits the programmer selection of the appropriate level of parallelism. Languages with this distinction include Ada, Argus, CSP, Distributed Processes, Lang's model, Lynx, Mesa, NIL, PLITS, and StarMod.

ConcurrentSmalltalk distinguishes between atomic and non-atomic objects, but the parallel control constructs apply to both kinds of objects. So, one would think that ConcurrentSmalltalk achieves a level of uniformity by using atomic objects only for synchronization. However, there is no mechanism for blocking a process within an atomic object. This means that one cannot implement a semaphore using atomic objects. The programmer must use atomic objects whenever synchronization is needed. This breaks the potential uniformity within ConcurrentSmalltalk.

SR provides a single encapsulation mechanism, the resource. However, the primitive data types and type construction mechanisms (similar to Pascal) are separate from resources. In addition, primitive types are passed by value while resources are passed by reference. So, while SR provides users with a single encapsulation mechanism, it still provides a dual abstraction model. One consequence is that local objects cannot be encapsulated. The programmer must promote an abstraction to a global resource to get encapsulation. Resources have higher run-time costs than internal types. ABCL/1 and Presto also have this same problem, although Presto's abstraction mechanism is cheaper.

Actors, Emerald, MultiLisp, and Smalltalk support a uniform model. In these sys-

	dynamic management	single abstraction	synchronous invocation	implicit reply
ABCL/1	√		√	√
Actors	√	√	√	√
Ada	√		√	√
Argus	√		√	√
ConcurrentSmalltalk	√		√	√
CSP				
Distributed Processes			√	√
Emerald	√	√	√	√
Lang's model	√		√	√
Lynx	√		√	√
Mesa	√		√	√
MultiLisp	√	√	√	√
NIL	√		√	√
Orient84/K	√		√	√
PLITS	√			
Presto	√		√	√
Smalltalk	√	√	√	√
SR	√		√	√
StarMod			√	√

Table 3.2: Uniform Encapsulation Capabilities

tems, the abstraction mechanisms do not imply any parallelism. This allows programmers to bind parallelism later in program development and avoids the costs of managing two different models of interaction.

3.1.2 Dynamic Data Management

Monitor languages have a tendency to require monitors to be allocated and named statically in the source. Distributed Processes has this problem. Mesa avoids this problem by allowing pointers to monitors. Most other languages allow dynamic references to encapsulations.

In CSP, processes are the unit of data abstraction. CSP allows processes to be created dynamically, but only in a hierarchical fashion. Arbitrary creation is not possible. Therefore, concurrent objects may only be created in a hierarchical fashion.

3.1.3 Synchronous Invocation

Many message passing systems support asynchronous invocation, and hence do not support synchronous invocations, on concurrent abstractions. (The synchronous invocation includes waiting on the result.) Systems in this category include Actors, CSP, and PLITS.

	dynamic processes	asynchronous invocation	concurrent invocation
ABCL/1	√	√	√
Actors	√	√	√
Ada	√		
Argus	√		
ConcurrentSmalltalk	√	√	√
CSP		√	√
Distributed Processes			
Emerald	√		
Lang's model	√		
Lynx	√		
Mesa	√	√	√
MultiLisp	√	√	√
NIL	√		
Orient84/K	√		√
PLITS	√	√	√
Presto	√		√
Smalltalk	√	√	√
SR	√		
StarMod	√		

Table 3.3: Sender Concurrency Capabilities

In contrast, most RPC, object, and rendezvous based systems support synchronous invocation, and hence do not support asynchronous invocation. These systems include ABCL/1, Ada, Argus, ConcurrentSmalltalk, Distributed Processes, Emerald, Lang's model, Lynx, Mesa, MultiLisp, NIL, Smalltalk, SR, and StarMod.

3.1.4 Implicit Reply

Those systems providing synchronous invocation also provide implicit reply. The Actor model is based on explicit continuations as parameters to individual operations. However, Actor languages often provide syntactic transformations to support implicit reply.

3.2 Sender Concurrency

Table 3.3 summarizes the languages' support for sender-based concurrency. The major weakness here is the support for asynchronous invocation.

3.2.1 Dynamic Process Creation

Some languages support only a fixed number of processes. These languages clearly will not support data parallelism and are therefore inadequate in the support of parallel programming. Distributed Processes has this problem. Note that a fixed number of names for processes does not necessarily imply a fixed number of processes. They may be created anonymously, as in the Uniform System [BBN, 1985c].

CSP provides dynamic process creation, but only in a hierarchical manner, *i.e.* the `cobegin` model. Processes may not live independently of their context.

StarMod allows dynamic process creation, but requires a static declaration of the virtual processors.

3.2.2 Asynchronous Invocation

Asynchronous invocation enables a process to initiate an operation and pick up the result later. ABCL/1, ConcurrentSmalltalk, and Orient84/K provide a *future object*, which allows the programmer to start an operation and then later explicitly retrieve the reply value.

In MultiLisp, and its derivative the Butterfly Lisp System [Steinberg *et al.*, 1986], the futures are built into the primitive types. This allows the waiting for the value of a future to be transparent to subsequent users of the value.

Most RPC based systems do not provide asynchronous invocations, and generally provide insufficient mechanisms to build general-purpose abstractions to provide asynchronous invocation. These problems are discussed below. Systems with this problem include Ada, Argus, Distributed Processes, Emerald, Lynx, NIL, and StarMod. (Note that NIL and StarMod have mechanisms to implement asynchronous invocations but that they require the cooperation of the callee and therefore do not meet the transparency requirement.) Lang's model and SR have asynchronous invocation only for operations that return no values. They suffer the same problems in defining a value-returning asynchronous invocation abstraction.

There are three major problems these languages have with providing a user abstraction for asynchronous invocation. First, the lack of generic definitions prevents declaring general-purpose abstractions where the types of arguments, temporaries, and results are not known at development time. SR specifically abandoned generic and polymorphic resources because the designers did not find a need for them in their distributed programming environment. However, Emerald, also a distributed language, provides generic definitions. Second, some languages require operations or procedures to be named directly, *i.e.* there are no procedure parameters. This forces the abstraction to know the operation name *a priori*. Emerald has this problem, though SR does not. Finally, most RPC systems allow multiple parameters. This forces the abstraction to know the number of arguments to an operation *a priori*, or to maintain an argument convention in user code. Both Emerald and SR have this problem. The maintenance of an argument convention often requires some form of polymorphism, which Emerald provides and SR does not.

3.2.3 Concurrent Invocation

Concurrent invocation enables the programmer to initiate a large set of operations in parallel. This capability is implied by the capability to do asynchronous invocation. With asynchronous invocation, the programmer may build concurrent invocation by writing a loop containing asynchronous invocations. To incorporate concurrent invocations into general-purpose abstractions, the language must provide a handle on operations. A set of predefine primitive mechanisms for concurrent invocation will not provide sufficient support for concurrent invocation on user-defined structures.

The SR language provides for concurrent invocation in the form of the `co` statement. Because the `co` statement is limited to an array-style specification of the operations, general parallelism over user-defined data structures is not possible. The programmer must pre-prepare the computation in an array form. In addition, the implementation of the `co` statement is likely to be more expensive than necessary in most cases. Argus has the same problem. No language appears to support general concurrent invocation without also supporting asynchronous invocation.

3.3 Receiver Concurrency

Table 3.4 summarizes the support for receiver-based concurrency supplied by each language. The main weakness is support for concurrent operations on an abstraction.

3.3.1 Concurrent Operations

Many parallel and distributed programming systems make processes the unit of concurrent encapsulation. Since the encapsulation provides only one active thread of control, such systems cannot fully exploit shared-memory multiprocessors. Encapsulating a data structure means that exactly one process will operate on it. Concurrent access is then not possible in principle. The following systems fall into this category: Ada, CSP, Distributed Processes, Lynx, and PLITS.

Object-based systems avoid the pitfall of making processes the unit of encapsulation; however, they sometimes implicitly enforce mutual exclusion between operations on an object. Such systems also cannot fully exploit shared-memory multiprocessors. ABCL/1, Lang's model, and Orient84/K have this problem. As discussed earlier, ConcurrentSmalltalk has this problem because it cannot block processes within atomic objects.

Justifications for a single active invocation generally center around the desire to insulate the programmer from concurrent programming within an object. However, this insulation is often not realized because real applications often need to interrupt processing within an object to handle a more pressing task. This single active invocation approach forces the programmer to code as though the object were concurrently accessed, even though there is no actual concurrency. For example, ABCL/1 provides express messages, which may preempt the executing invocation. This forces the programmer to deal with concurrent invocations arising from express messages. Likewise,

	concurrent operation	implicit dispatch	asynchronous processing
ABCL/1		$\surd$	$\surd$
Actors		$\surd$	$\surd$
Ada			$\surd$
Argus	$\surd$	$\surd$	
ConcurrentSmalltalk		$\surd$	$\surd$
CSP			$\surd$
Distributed Processes		$\surd$	$\surd$
Emerald	$\surd$	$\surd$	$\surd$
Lang's model		$\surd$	
Lynx		$\surd$	$\surd$
Mesa	$\surd$	$\surd$	$\surd$
MultiLisp	$\surd$	$\surd$	$\surd$
NIL			$\surd$
PLITS			$\surd$
Presto	$\surd$	$\surd$	
Orient84/K		$\surd$	$\surd$
Smalltalk	$\surd$	$\surd$	$\surd$
SR	$\surd$	$\surd$	$\surd$
StarMod	$\surd$	$\surd$	$\surd$

Table 3.4: Receiver Concurrency Capabilities

Orient84/K messages have priorities. Higher priority messages will preempt lower priority messages. The designers of Orient84/K recommend placing critical regions only in the code for handling the highest possible priorities. This is a linguistically more complicated approach than explicitly managing critical regions.

Hewitt's Actor system provides a model of computation in which actors accept the next message, communicate with other actors and specify a replacement actor. The parallelism is greater in an Actor model than in single active invocation systems because an actor may specify its replacement early and allow the replacement actor to accept messages concurrently with the processing remaining in the first actor. This approach does not provide as much parallelism as the multiple active invocation approach because an incoming message must wait until a replacement actor is specified for the previous message.

3.3.2 Implicit Dispatching

Several systems provide implicit dispatching even when operations are serialized. These include ABCL/1, Distributed Processes, and Lynx. Hewitt's Actors provides for implicit dispatching, but the operations are serialized until each actor specifies its replacement.

Implicit dispatching is a natural outgrowth of concurrent operations. The following systems provide implicit dispatching with concurrent operation: Argus, Concurrent-

Smalltalk, Emerald, Mesa, MultiLisp, Smalltalk, SR, and StarMod. Only operating systems, such as Accent [Rashid and Robertson, 1981] and Mach [Accetta *et al.*, 1986] provide for concurrent operations while requiring an explicit dispatch.

SR provides both implicit and explicit dispatching. This forces the system to find out at run time which form is being used and schedule among the possible explicit receipts. This increases the run-time cost of invocation. In addition, SR overloads additional "semaphore" semantics on certain forms of invocation and explicit receipt. Andrews *et al.* [1986] state "In general, however, there is no guarantee that a process will find a selectable invocation even if selectable invocations continually arrive; another process might continually obtain access and then select and service all such invocations. The single exception is that processes servicing *semaphore-like* operations are guaranteed access in a FCFS (first-come/first-served) order."

3.3.3 Asynchronous Processing

Languages based on asynchronous messages inherently provide asynchronous processing because the handling of the message proceeds asynchronously to the sender. Languages in this class are CSP and PLITS.

Many operation, or rendezvous, languages enable the receiver to continue processing after the code block that handles the message. Ada, Distributed Processes, Lynx, SR, and StarMod fall into this category. StarMod's implicit dispatch is not simultaneously available with asynchronous processing, so it does not fully support receiver concurrency, even though it supports the pieces. Some programming systems provide an *early reply*, which supplies a reply value but enables the operation to continue executing. The languages with this mechanism include ConcurrentSmalltalk, Lynx, and SR.

In languages that do not have built-in mechanisms for asynchronous processing (such as early reply), programmers must create asynchrony by dynamically creating a process and then communicating the task to be done to the new process. This process must have access to enough of the operation's context to effectively continue the computation. ABCL/1, Argus, Emerald, and Mesa provide sufficient mechansims to implement asynchronous processing, but make access to this context difficult.

Emerald

Even though Emerald supports asynchronous processing, the support is weak. This analysis concentrates on a style of programming in which messages are handled one at a time asynchronously to the callers. There are two immediate approaches to achieving this in Emerald. The first is based on a single server process that accepts messages from a queue. The second is based on creating a separate process for each message and then synchronizing in a monitor.

In the first approach, the object maintains a process to sequentially handle the messages. Because Emerald processes may not directly communicate, separate operations on the object accept the message and place it in a queue for later retrieval by the handler process. See the example receiver1 in Appendix A for the Emerald code. There are several problems with this approach. Individually, they may be overcome. Together, they represent a significant disadvantage. The queue approach is inefficient and notationally cumbersome.

- To solve the problem of simultaneously handling multiple parameters, operation parameters must be packed into a single record argument. This increases notational overhead and may increase run-time overhead.

- A separate queue element record must be defined containing a field that allows the handler process to distinguish between the different operations.

- The queue element record must contain a polymorphic component to account for different operation parameter records. This is possible in Emerald, but incurs run-time checking when removing the parameters from the queue element records. Note that a generic queue record does not meet the need, because multiple types may be in the queue simultaneously.

- The extra programming effort also involves considerable run-time overhead. The queue requires dynamic allocation of queue elements in addition to the dynamic allocation of the accepter's activation record.

- The code necessary to package the record, place it into the queue, remove it from the queue, and invoke the appropriate processing operation is large. In addition, the programmer must define the operations that accept messages separately from the operations that process the messages. This represents an increased development and maintenance effort in comparison to synchronous processing and therefore inhibits programmers from using asynchronous processing.

- One potential approach to optimize parallel programs is to replace asynchronous activities by synchronous activities. Such optimization is difficult in this approach because the processing is very decoupled from the request. The programmer could do this task manually, but the effort required is sufficiently large to discourage such optimizations except in extreme cases. This in turn inhibits the programmer from indicating more parallelism than is *obviously* necessary.

- Because the process must exist throughout the lifetime of the object, the system must maintain more dynamic state.

The second approach to the asynchronous processing problem creates a new process for each message, which then handles the message asynchronously to the original invocation. A monitor synchronizes the handler processes so that requests are handled one at a time. The basic problem with this approach is that the programmer must step outside the current object to get a new process, which then must use a separate operation to get back into the object. See the example receiver2 in Appendix A for the Emerald code. There are several problems with this approach also.

- The implementation is likely to allocate an empty object just to have an object in which the new processes may live.

- Because the handler process is in a separate object, it cannot access the receiver object's variables. It must invoke a special monitored processing operation. This in turn requires the programmer to define the accepting operation separately from the processing operation.

- The programmer must make the processing operations visible outside the receiving object in order to make them visible to handler process.

- This approach requires the parameters to be redefined in the monitored operations as well as the accepting operations.

- Creation of the handler process requires the dynamic allocation of a process activation record. Assuming the callee is on a different node from the caller, the implementation also dynamically allocates a process activation record to serve the original activation. Asynchronous handling can be done with the creation of exactly one process activation record, but Emerald requires two.

  While a sufficiently intelligent language processor can provide a single process activation record for both the remote invocation and the handler process, the straightforward implementation will still allocate two process activation records. The merging of these records is non-trivial because the processing of local activations does not create a new process activation record. If the operation invocation does not create a new process activation record, the process creation must create a process activation record.

Emerald could be fixed with an early reply, but such an addition would make the process section redundant and would radically change the implementation.

3.4 Uniform Control

Table 3.5 summarizes the support for uniform control in each of the languages. Support is weak across the board, with the exception of MultiLisp.

3.4.1 Handler for Parameters

Message based systems can generally refer to the message as a unit. This allows them to pass messages through intermediaries that do not necessarily know the structure of the message. PLITS is the primary example here. Not all message-based systems provide access to the message as a whole. These include CSP and NIL.

RPC based systems provide for multiple parameters to an operation. However, these systems have no mechanism to refer to the set of arguments to an operation. One cannot pass the set of arguments to an operation to another object and have that object execute the operation without the object having knowledge of the number of the arguments.

	parameter handles	operation handles	pair handles	context handles
ABCL/1				
Actors				$\surd$
Ada				
Argus				
ConcurrentSmalltalk		$\surd$		$\surd$
CSP				
Distributed Processes				
Emerald				
Lang's model				
Lynx				
Mesa				$\surd$
MultiLisp	$\surd$	$\surd$	$\surd$	$\surd$
NIL			$\surd$	
Orient84/K				
PLITS	$\surd$	$\surd$		
Presto			$\surd$	
Smalltalk		$\surd$		$\surd$
SR			$\surd$	
StarMod		?		?

Table 3.5: Uniform Control Capabilities

Languages with this problem include ABCL/1, Actors, Ada, Argus, ConcurrentSmalltalk, Distributed Processes, Emerald, Lang's model, Lynx, Mesa, MultiLisp, Smalltalk, SR, and StarMod.

This problem can be solved by a programming convention in which all operations accept only one argument. When more than one argument is needed, the arguments must be packaged together in a record. While this convention works, it requires definition of an additional record type for each operation. It could also introduce more run-time overhead in packing and unpacking the record, since the typical approach would be to pack the record and then copy it to the parameter list.

3.4.2 Handles for Operations

In many message-based languages, the message contains a field indicating the desired operation. This allows the system to easily reference operations indirectly. PLITS falls into this category.

No RPC-based language appears to support operation references. SR, however, provides for operation references when the operation is bound to a specific object. There is no general indicator for an unbound operation.

Operation references are not incompatible with the basic RPC model, but require additional syntax to allow the specification of a reference to an operation and to invoke an operation through the reference.

3.4.3 Handles for Work in a Context

ConcurrentSmalltalk, and Smalltalk provide for the notion of work within a context with the *block* mechanism. Without such a mechanism, it is not possible to build a general purpose abstractions for a parallel for-loop.

The notion of work within a context should apply equally to parallel constructs as well as sequential constructs. For example, the SR `fa` (for all) statement is a powerful looping mechanism that has no parallel analog and therefore requires considerable work to change from sequential execution to parallel execution.

Passing operation references is not sufficient unless the language has nested operations. Otherwise, the programmer would lose access to the main operation's local variables.

3.5 Value Orientation

Table 3.6 summarizes the support for a value orientation within each of the languages. The main weakness is support for concurrent objects as value-based parameters and variables. Because of this, no existing language appears to support a uniform value-based model.

Most systems support a conventional value model for local variables and parameters, but use a reference model for distributed abstractions. Mesa and Presto have a nearly uniform value model, which would imply a value model for distributed abstractions also. However, objects containing synchronization objects cannot be copied, and hence do not have a direct value. These languages force a reference model for objects with synchronization. An appropriate change in semantics could fix this problem.

SR has value parameters, but since the resources provide the only encapsulation mechanism and must be passed by reference, a significant fraction of the actual arguments will be passed by reference.

3.5.1 Controlled Aliasing

Emerald defines immutable as a hint to the implementation that it may copy the representation at any time. The representation may change over time, so long as the abstract value of the object does not change. This means that immutable objects do not help the aliasing problem, although they can be used to alleviate the contention problem.

3.5.2 Remote Arguments

One consequence of the reference model is that parameters are passed by reference. This may cause unacceptable performance penalties in distributed systems due to heavy

	value abstractions	value locals	value parameters
ABCL/1			
Actors			
Ada		√	√
Argus			√
ConcurrentSmalltalk			
CSP		√	√
Distributed Processes		√	√
Emerald			
Lang's model		√	√
Lynx		√	√
Mesa		√	√
MultiLisp			
NIL		√	√
Orient84/K			
PLITS		√	√
Presto		√	√
Smalltalk			
SR		√	√
StarMod		√	√

Table 3.6: Value Orientation Capabilities

crossing of machine boundaries. This is enough of a problem so that Argus, which uses a reference model on a single machine, uses a value model between machines.

SR copies non-resource arguments. However, resources are passed by reference. SR provides no mechanisms to alleviate the performance penalty associated with passing resources as parameters.

Object Movement in Emerald

Emerald introduces the notions *call-by-move* and *call-by-visit*. Call-by-move indicates that the argument object is to be moved to the node containing the called object. Call-by-visit indicates that the argument object is to be moved to the node containing the called object for the duration of the call and then moved back to the caller's node upon completion. Because objects are generally small and contain a number of references to other objects, Emerald provides a mechanism for moving *attached* objects. Moving an object also moves the objects attached to it.

The caller indicates the appropriate transmission method. This is appropriate since the caller understands the context of the call, and the movement is independent of semantics. Unfortunately, if the caller is a general purpose abstraction, the caller does not understand the context of the call. So, embedding the movement semantics in the source again restricts the programmer to *ad hoc* abstraction.

Because general references to objects throughout the system are too expensive, the reference model requires translating references on move. General, system-wide references are too expensive because they are used heavily for everything except primitive immutable objects.

The Emerald compiler [Hutchinson, 1987] attempts to determine when an object is completely contained within another to reduce the cost of object movement and run-time access.

3.5.3 Object Contention

SR has no immutable resources, so programmers must do explicit copying to avoid contention for data contained in critical resources. This problem is aggravated by the resource as the sole encapsulation mechanism.

3.6 Summary

No single language provides all the desired capabilities for parallel programming. Most languages provide two mechanisms for abstraction, one for global, concurrent objects and one for local, sequential objects. This prevents uniform encapsulation. Sender-based concurrency is the most supported of all the capabilities, though many languages do not provide support for asynchronous invocation. The most prevalent failure of existing languages in supporting receiver-based concurrency was the lack of support for concurrent operations within a single object. Support for uniform control was particularly weak. With the exception of MultiLisp, the languages failed to support for handles on two or more of a set of parameters, operations, bound object and operation pairs, or code within a context. No language uniformly supports the value model of variables and parameters, though Mesa and Presto come close. The following chapter presents a model that provides all of the capabilities discussed.

4 Proposed Model

This chapter presents a model of parallel programming that provides the capabilities discussed earlier. The model relies on a combination of a few simple concepts and user-programmable abstractions to achieve a rich programming environment. After describing the model, the chapter demonstrates how the model supports each of the required capabilities.

In describing SR, Andrews *et al.*[1986] state “Thus a distributed programming language necessarily contains more mechanisms than a sequential programming language.” The proposed model contradicts this statement. It is based on a few, carefully chosen, general mechanisms that apply to sequential and parallel programming uniformly.

4.1 Model Description

This section describes the features of the proposed model. This description is not a complete language definition. It is meant to provide an overview of the nature of the model. This section also introduces the notation used in later examples.

4.1.1 Entities

The entity is the unit of abstraction and encapsulation. All items within the language are entities. An entity is composed of a set of variables, a set of operations, and a set of expressions. The expressions serve to initialize the entity. Only the operations and expressions of an entity may access the entity's variables. Operations on entities are the fundamental communication mechanism in the model.

4.1.2 Variables

Variables contain entities, that is, the variables adhere to the value model. A variable name refers to an entity within the least containing scope that defines that variable. Because of the value model of variables, two different simple variables names must refer to two different entities. A literal also refers to a distinct entity

Because the value model cannot directly represent non-hierarchical structures, the model introduces explicit references, *i.e.* pointers. Reference variables must be explicitly dereferenced. There are two kinds of references, those that reference entities, and those that reference bound entity and operation pairs. References to bound entity and operation pairs are called a *port* references. This term is analogous in function to the port of message based systems.

Operations are usually referred to with literal identifiers, although variables may also indicate an operation.

4.1.3 Methods

The method, or procedure, for handling an operation is defined by a sequence of activation variable definitions and a sequence of expressions.

An operation in execution is called an activation. Activations may proceed concurrently both between and within an entity. The model does not supply synchronization. If the programmer needs to synchronize activations, the programmer must explicitly program such synchronization. Synchronization can arise from a base model without synchronization [Lamport, 1986a; Lamport, 1986b]. However, programming at such a level is cumbersome, so the library associated with any implementation of the proposed model must provide a some entities whose semantics provide synchronization between operations on them. For instance, a library could provide semaphores, test-and-set bits, read-write locks, *etc.*

The proposed model does not support an explicit notion of process. Instead, the language model supports the activation as the fundamental unit of execution. An activation is a method in execution. It contains the local variables for the method. An activation may wait on other activations, but the activation itself is a self-contained unit of parallelism.

4.1.4 Invocations

An invocation is signaled by the joining of a port reference with an argument entity. (Record constructors can combine multiple arguments into a single argument entity.) The invocation constitutes an expression yielding an entity. The caller always waits on the reply value. The model does not provide asynchronous invocations directly, however, they can be implemented easily in user code.

4.1.5 Early Replies

The result of an operation may be returned at any point in the execution of the operation. The execution of the operation continues concurrently with the caller. This early reply is the sole source of concurrency within the proposed model, no additional mechanism, such as processes, is needed. Early reply, coupled with generic definitions, is sufficient to allow the straightforward definition of asynchronous invocation.

4.1.6 Blocks

Blocks provide a form of *closure*. They may be passed outside the enclosing scope to be executed by other entities. Blocks maintain access to the variables defined within the scope of the block.

Activations of an operation are themselves entities. The existence of the environment is identical to the existence of the entity representing the environment. Any mechanisms for determining the existence of a normal data entity also apply to environments.

The model considers blocks as operations on the activation of an operation on an entity. As such, they are implicitly an entity and operation pair, so the definition of a block is a port reference to an anonymous operation on the activation. Blocks also have reply values.

4.1.7 Notation

The following notation is somewhat informal. It provides a means to present examples and does not constitute a language definition. However, the notation is relatively complete. A prototype implementation requires little more syntax, if any. The notation is entirely symbol based and uses no keywords. The following table presents a summary of the symbols used.

language primitive	symbol	language primitive	symbol
type declaration	`=`	generic definition	`<< >>`
record type	`[ ]`	entity type	`[[ ]]`
variable declaration	`:`	variable separator	`;`
block constructor	`{ }`	argument separator	`|`
operation invocation	`!`	operator selection	`.`
expression separator	`;`	reply indicator	`#`
record instance	`[ ]`	component separator	`,`
instance declaration	`::`		

The symbols to select an operation '`.`' and invoke an operation '`!`' have equal precedence and are left associative. The resulting expressions are best thought of as pipes. The sequence `a.b!c.d!e` should be read as apply `b` to `a` with argument `c`, taking the result and applying `d` to it with argument `e`. Other symbols, such as '`+`', '`:=`', *etc.* are abbreviations for .identifier`!` where the identifier is the appropriate name of the operation.

A record is a special form of entity. The compiler automatically generates the operations necessary to access record components. An empty record is signified by '`[]`'. It is used as a place holder for unused arguments and results.

Traditional control constructs are absent from the notation. Control flow is handled in the context of operations on entities. For example, boolean entities provide an `if` operation, which executes the code block passed as a parameter if the value of the entities is `true`. The operation identifiers have been chosen to be similar to keywords in more traditional languages.

The following modified BNF specifies the syntax of the notation used in the remainder of this document. The notation is defined with a modified BNF. Items enclosed in braces ($\{\}$) indicate a sequence of zero or more items. Items separated by a bar ($|$) indicate alternatives.

```
type_def     ≡ normal_def | generic_def
normal_def   ≡ identifier = type
generic_def  ≡ identifier << identifier { , identifier } >> = type
type         ≡ identifier | record_type | entity_type
entity_type  ≡ [[ { variable } { expression } { operation } ]]
record_type  ≡ [ { variable } ]

variable     ≡ normal_var | generic_var
normal_var   ≡ identifier : type ;
generic_var  ≡ identifier : type << identifier { , identifier } >> ;
operation    ≡ identifier : block
block        ≡ { result argument | { variable } { action } }
result       ≡ | type ;
argument     ≡ | variable | record_type ;
action       ≡ # expression ;   | expression ;

expression   ≡ pair ! entity | entity
pair         ≡ identifier | block | expression . operation | ( expression )
entity       ≡ literal | identifier | record_inst | entity_inst | ( expression )
operation    ≡ literal | identifier | ( expression )

record_inst  ≡ [] | [ expression { , expression } ]
entity_inst  ≡ :: type_def
```

4.2 Uniform Encapsulation

The proposed model provides for scalability by associating communication with abstraction. Since programmers use layers of abstractions, the implementation can choose to cross machine boundaries at any layer of abstraction. Higher layers of abstraction communicate less frequently than lower layers of abstraction.

Reference variables provide for dynamic reference to data. Entities are created dynamically by execution of a constructor. Entities are the only abstraction mechanism.

The model provides implicit reply by definition. The callee does not specify the destination of the reply.

4.2.1 Synchronous Invocation

The model provides synchronous invocation by definition. It provides synchronous over asynchronous invocation because programmers usually want synchronous invocation and asynchronous invocation has potential notational and implementation problems.

Yonezawa *et al.* [1986] provide a construction for collapsing synchronous invocations into asynchronous invocations. This construction uses an intermediate object and a blocking select construct. In essence, they have traded the re-activation of a known invoker for the blocking filtering of incoming messages. This construction only works because the blocking select construct is not an object invocation, which means the model is non-uniform. The ability to wait on a reply appears primitive. The mechanism for suspending the current invocation while completing a requested invocation is already common in standard sequential programming languages as procedure calls.

4.3 Sender Concurrency

Each operation invocation implicitly creates a new process. The operation starts asynchronously with respect to any other processes within the target entity and executes concurrently with them.

4.3.1 Asynchronous Invocation

While programmers need asynchronous invocation, a sufficiently powerful programming model need not support asynchronous invocation as a primitive operation. Our model does not provide direct support for asynchronous invocations. However, such support is easily defined by a user abstraction based on generic definitions, and early reply. Asynchronous invocation support in the programming model is not necessary, synchronous invocation support is crucial. In addition, such an approach encourages the implementation of different types of asynchronous invocations, each tuned to the needs at hand. For instance, possible types of user-programmable asynchronous invocation are single asynchronous reply, first of several asynchronous replies (including timeouts), and priority ordering of replies as they arrive.

The generic definition of a simple *future* variable for obtaining a single asynchronous value via early reply follows. The `do` operation initiates the request and the `wait` operation picks up the result.

```
-- define some example variables
i : integer ;  r : real ;
-- define a future variable for an operation which
-- accepts a empty record and returns an integer
f : future<< integer, [] >> ;
f.do![r.trunc,[]] ;  -- start the trunc operation on r, empty argument
... -- do some other computation
i := (f.wait![]) ;  -- wait on the result of trunc and then assign it
```

The definition of the future type is straightforward. This is user code, so users are free to implement similar abstractions as best suit their applications.

```
future<< result_type, argument_type >> =
    [[  -- start of entity type definition
    -- local variables
    hold : result_type ;  -- to hold result of operation
    ready : binary_semaphore ;
    -- initialization expressions
    ready.lock![] ;

    -- operation methods
    do : { -- no result defined, assumed []
           -- define argument
           [ -- description of operation
             object_oper : pair_ref<< result_type, argument_type >> ;
             -- description of operand
             argument : argument_type ] ;
           | -- end of result and argument definitions
           # [] ;   -- reply early allowing caller to continue
           hold := (object_oper!argument) ;  -- do operation and assign
           ready.signal![] ;  -- synchronize with wait operation
         }

    wait : { -- result is of result_type
             result_type ;
             -- no argument defined, assumed []
             |  -- end of result and argument definitions
             ready.wait![] ;  -- synchronize with do operation
             # hold ;  -- return result
           }
    ]]  -- end of entity type definition
```

4.3.2 Concurrent Invocation

It is easy to define concurrent invocations in the proposed model using the block construct. The following example specifies the the execution of a parallel for-loop over a range of integers. It prints each integer in parallel.

```
::range<< 1, n >>.parafor!{ j : integer ; | j.print![] ; }
```

The `::range` construct defines a entity instance. The `parafor` operation is applied to this instance, with the code block define in braces as the argument. The code block accepts and integer `j` as an argument and then applies the `print` operation to the integer.

The ability to specifying concurrent invocation is not sufficient, one must also have the ability to implement concurrent invocation. The following presents two implementations for a parallel for-loop over a range of integers. The first has sequential startup.

The second has a tree structured startup. A programmer or implementation could choose either implementation at compile time.

```
range<< l, h >> =
    [[
    low : integer := l ; high : integer := h ;

    parafor : { block : pair_ref<< [], integer >> ; |
        -- an array [ low .. high ] of futures
        f : array<< low, high, future<< [], integer >> >> ;
        i : integer := low ;
        -- start all operations
        while![ { boolean ; | # i <= high ; },
            {| -- start operations at index i
              f.index!i.do![ block, i ] ;
              i += 1 ;
            } ] ;
        -- wait for all completions
        i := low ;
        while![ { boolean ; | # i <= high },
            {| -- wait for completion at index i
              f.index!i.wait![] ;
              i += 1 ;
            } ] ;
        }
    ]]

range<< l, h >> =
    [[
    low : integer := l ; high : integer := h ;

    parafor : { block : pair_ref<< [], integer >> ; |
        f : future<< [], integer >> ;
        (h = l).ifelse![
            {| block!h ; },  -- h = l so only one invocation
            {| m : integer ;
              m := ((h + l) / 2) ;  -- split in two
              -- start lower half
              f.do![ ::range<< l, m >>.parafor, block ] ;
              -- do upper half
              ::range<< m+1, h >>.parafor!block ;
              -- wait on lower half
              f.wait![] ;
            } ] ;
        }
    ]]
```

4.4 Receiver Concurrency

The model provides concurrent operation by definition. Separate invocations on an entity must be synchronized by the entity itself. The model provides implicit dispatching by definition. Operations on entities start without the cooperation of other operations already executing within the entity.

4.4.1 Asynchronous Processing

The model provides for asynchronous processing with the early reply. This enables an invocation to continue after satisfying the caller. This approach has an advantage over explicit process creation because it combines message acceptance and processing into one method definition. It also naturally only needs one process activation record where the explicit process creation would generally require two. Because the asynchrony is coupled with the operation activation, it is possible to optimize the operation into a synchronous operation.

4.5 Uniform Control

The support for uniform control within the model is fairly direct, but requires some explanation.

4.5.1 Handles for Parameters

Since operations take only one parameter, multiple parameters must be packaged into a record. Referring to the record effectively provides a handle on the set of parameters. For example, consider an entity that forwards an invocation on to another operation instead of doing the work itself. The operation is encoded in the port reference `downstream`.

```
forward =
    [[
    downstream : port_ref<< any, any >> ;

    action : { any ; argument : any ; |  -- result and argument types
        # downstream!argument ;
        }
    ]]
```

Multiple arguments may pass through `forward` because they are wrapped in a record. This wrapping is notationally convenient because of the direct support for a record constructor.

```
forward.action![arg1, arg2, arg3] ;
```

4.5.2 Handles for Operations

Operation variables provide for indirect reference to operations. For example, the following entity takes an argument indicating an operation to be performed on another entity. It then invokes that operation once with the argument `1` and once with the argument 2.

```
double =
    [[
    other : other_type ;

    do : { opervar : operation ; |  -- argument
        other.(opervar)!1 ;
        other.(opervar)!2 ;
        }
    ]]
```

See section 5.4 for a more complete example.

4.5.3 Handles for Work in a Context

Blocks provide handles for work in a context. In the following example, The block passed to the `parafor` operation has access to the enclosing definitions, namely `var1` and `var2`.

```
{
var1 : type ; var2 : type ;

::range<< 1, 4 >>.parafor!
    { i : integer ; |
    var1.do!i ;
    var2.do!i ;
    }
}
```

4.6 Value Orientation

The proposed model provides a value orientation on variables and parameters by definition. Because the value model is not capable of directly representing non-hierarchical structures, the model also provides explicit references. References must be explicitly dereferenced.

In object-oriented systems, the object being operated on is implicitly passed by reference. When the object being operated on is also the object being modified, no explicit references are needed. The object-oriented nature of the proposed model means that explicit references are less common than in non-object-oriented (or procedural) models.

The value orientation of the proposed model reduces the alias problem (which increases potential parallelism), reduces object contention, allows many possible implementations of parameter passing, and allows more efficient storage management.

5 Examples

This chapter presents several example entities. These show some of the range of programming styles possible within the basic model.

5.1 Message Passing

Message passing styles are easy. For this example, messages are processed serially. Each object replies immediately upon receipt of an activation, synchronizes the activation with other activations, and completes the computation.

```
sample =
    [[
    ready : binary_semaphore ;
    ready.unlocked![] ;

    send : {  message : message_type |  -- argument
        # [] ;  -- let caller continue
        ready.wait![] ;  -- synchronize
        ... do work ...
        ready.signal![] ;
        }
    ]]
```

Compare this solution to asynchronous processing with the Emerald solutions in Appendix A. Sending a message to an instance of `sample` is done as follows:

```
... instance.send!message ...
```

A reference to an object and operator pair acts essentially as a port. This allows the caller to be unaware of the specific object and operation.

```
    other : port_ref<< any, any >> ;
    other!value ;
```

One potential problem with the above approach is the potentially unbounded number of operations that may be waiting on the semaphore. By delaying a reply until after synchronization, an object can provide flow control. This is similar to CSP [Hoare, 1978]. Consider the following member of a pipeline.

```
segment =
    [[
    downstream : port_ref<< [], parameter_type >> ;
    ready : binary_semaphore ;
    ... other variables ...
    ready.free![] ;

    incoming : { param : parameter_type ; |
        ready.wait![] ;  -- synchronize
        # [] ;           -- then allow caller to proceed
        ... process parameter ...
        downstream!newvalue ;
        ...
        ready.signal![] ;
        }
    ]]
```

Note that because of the anonymous nature of reference to object and operation pairs, we can insert a buffer between two elements of the pipeline, without affecting the correctness.

This example uses the dynamic allocation of activation records to provide storage for the buffer.

```
buffer<< n, elemtype >> =
    [[
    downstream : port_ref<< [], elemtype >> ;
    ready : counting_semaphore ;
    ready.free!n

    setup : { ref : port_ref << elemtype >> ; |
        downstream := ref ;
        }

    insert : { parameter : elemtype ; |
        ready.wait![] ;
        # [] ;  -- reply after synchronization
        downstream!parameter ;
        ready.signal![] ;
        }
    ]]
```

Since using dynamic activation allocation to store buffer elements may be too expensive, the second solution uses an array. Note that the continuing activation of the setup operation provides "push" semantics to the buffer.

```
buffer<< n, elemtype >> =
    [[
    downstream : port_ref<< elemtype >> ;
    container : array<< 1, n, elemtype >> ;
    empty : counting_semaphore ;
    full : counting_semaphore ;
    empty.free!n
    full.free!0

    setup : { ref : port_ref << elemtype >> ; |
        # [] ;  -- allow caller to continue
        downstream := ref ;
        -- push messages downstream as they come
        loop.forever!
            {| full.wait![] ;
              ... get parameter from array ...
              downstream!parameter ;
              empty.signal![]
            } ;
        }

    insert : { parameter : elemtype ; |
        empty.wait![] ;
        # [] ;  -- caller waits to avoid using activations for storage
        ... place parameter in array ...
        full.signal![] ;
        }
    ]]
```

Note that these two implementations may be used interchangeably. With adequate compiler support, this choice could be rebound without directly affecting the source.

5.2 Shared Memory

Shared memory systems are characterized by processes calling procedures concurrently on shared objects. This is easy to implement under the proposed model by delaying the reply until the end of the operation.

```
abstraction =
    [[
    ... abstraction local variables ...

    procedure : { return_type ; [ ... parameters ... ] |
        ... procedure local variables ...
        ... statements ...
        # return_value ;
        }
    ]]
```

5.2.1 Warshall's Algorithm

Warshall's algorithm is an adjacency matrix based algorithm for determining the transitive closure of a graph M. The first section of code provides a more conventional syntax for the algorithm.

```
sequfor k in 1 .. n do
    parafor i in 1 .. n do
        if M[ i, k ] then
            -- M[ i ] |= M[ k ] ;
            parafor j in 1 .. n do
                M[ i, j ] |= M[ k, j ]
                end ;
            end ;
        end ;
    end ;
```

A straightforward translation to the model, using the tools developed in the previous chapter, gives the following code. A language system could bind the actual implementations of the parallel for-loops at any time.

```
::range<< 1, n >>.sequfor!
    { k : integer ; |
    ::range<< 1, n >>.parafor!
        { i : integer ; |
        M.index!i.index!k.if!
            {|
            ::range<< 1, n >>.parafor!
                { j : integer |
                M.index!i.index!j |= (M.index!k.index!j) ;
                } ;
            } ;
        } ;
    } ;
```

5.3 Combining Results

Often many results need to be combined in a manner like a summation. These two implementation are analogous to the two parallel **for-loop** implementations.

```
range<< l, h, combtype >> =
    [[
    low : integer := l ; high : integer := h ;

    combine : { combtype ;
        [ op : operation, block : pair_ref<< combtype, combtype >> ] ; |
        -- array [ low .. high ] of future for combtype yields combtype
        f : array<< low, high, future<< combtype, combtype >> >> ;
        c : combtype ;
        i : integer := low ;
        while![ { boolean ; | # i > high ; },
            {| f.index!i.do[ block, i ] ;  -- start operations
              i += 1 ;
            } ] ;
        c := c.identity!op ;  -- the mathematical identity over op
        i := low ;
        while![ { boolean ; | # i > high ; },
            {| c := c.(op)!(f.index!i.wait![]) ;  -- combine results
              i += 1 ;
            } ] ;
        # c ;
        }
    ]]

range<< l, h, combtype >> =
    [[
    low : integer := l ; high : integer := h ;

    combine :
        { combtype ;
          [ op : operation, block : pair_ref<< combtype, combtype >> ] ; |
          c : combtype ;
          (h = l).ifelse![
              {| c := block!h ; },
              {| m : integer ;
                f1 : future<< combtype, combtype >> ;
                f2 : future<< combtype, combtype >> ;
                m := ((h + l) / 2) ;
                f1.do![ range<< l, m >>.foreach, block ] ;
                f2.do![ range<< m+1, h >>.foreach, block ] ;
```

```
            c := (fe.wait![]).(op)!(f2.wait![]) ;
          } ] ;
      # c ;
      }
  ]]
```

Again, either implementation may be chosen at compile time, without affecting the calling code. Using these forms, it is relatively easy to sum an array.

```
range<< 1, n, integer >>.combine[ add, { i : integer ; | A.index!i ; } ] ;
```

A better solution is for the array itself to implement the combining operation directly.

```
A.combine!add ;
```

5.4 Relaxation

This is a cell in a relaxation network. In this example, I use a combine operation defined over a set. The combine operation takes an operation to execute on an element instead of a code block.

```
cell =
    [[
    neighbors : set<< entity_ref<< cell >> >> ;
    oldvalue : real ; newvalue : real ;

    getvalue : { real ; | # oldvalue ; }

    compute : {| newvalue := neighbors.combine[ add, getvalue ] ; }

    update : { real |
        result := false ;
        (newvalue <> oldvalue).if!
            {| oldvalue := newvalue ;
            result := true ;
            } ;
        # result ;
        }
    ]]
```

The control for the algorithm uses a combiner to determine when the network has stablized.

```
-- declare cells
cells : set << cells >> ;
-- for each cell, apply the compute operation
cells.parado!compute ;
while![
    -- while the combination of updates has some unstable members
    { boolean ; | # cells.combine[ or, update ] ; },
    {| -- for each cell, apply the compute operation
      cells.parado!compute ;
    } ] ;
```

6 Implementation Strategies

This chapter discusses several possible implementation strategies for the proposed model. These strategies can work together to provide a system that is both flexible and efficient. The flexibility allows the programmer to adapt the parallelism to the level needed for a specific architecture.

One of the advantages of the proposed model is that programmers may bind the actual parallelism late in program development. There are two aspects to this late binding. First, programmers may choose an implementation for a control abstraction that is most appropriate at a given level in the program. For example, the programmer may choose to implement the parallel-for-loop abstraction as a simple sequential loop at a very low level in the program, and as a tree of processes as in the example in section 4.3. Second, programmers (or the compiler) may choose the most appropriate implementation for the primitive language mechanisms. For example, a pragma could indicate that an invocation be compiled in full generality, or expanded as an in-line procedure call. Since the choice of implementations for user abstractions or language mechanisms does not affect the semantics of the program, the programmer (or the compiler) may bind the choice very late.

The rest of the discussion requires a note on terminology. An activation record stores the state the local variables of an active invocation. Blocks as well as operations will have activation records. An activation stack stores a number of activation records that are allocated and deallocated in a strictly *last-in-first-out* manner.

6.1 Passing Parameters

Since parameters are passed by value, the straightforward mechanism is to copy the value. This will typically involve a memory to memory copy. In the case of a record constructor as the argument to an operation, the record can be built at the destination, no copying of the record as a whole need be done. For small values, the parameters could be passed in registers. For very large structures, an appropriate virtual memory system could implement copy-on-write. In the case of immutable objects, the implementation could either pass a reference to the object or copy the object, whichever is more efficient.

Mutable objects can also be passed by reference under certain conditions. If the compiler determines (or the programmer specifies) that the parameter is not modified by the callee, and the argument is not concurrently modified (by another process or via aliasing), then the implementation can pass by reference. This determination can be done with standard flow analysis techniques developed for compiling sequential programs on parallel machines. This thesis will not address such issues.

The choice of parameter passing implementation can be made by the compiler or by the programmer in the form of a pragma. In either case, the binding to implementation is done very late in program development.

6.2 Invoking Operations

Consider the case of an operation which replies just before termination. The caller's activation will exist for the entire time that the callee's activation record exists. This means that we can allocate the callee's activation record on the end of the caller's activation record and avoid the cost of a more general allocation strategy. The callee's activation *record* is allocated on the caller's activation *stack*. The caller's activation record may or may not be the base record on its activation stack. This allocation of activation records on a stack is the typical implementation of procedure calls.

Procedure calls can often be expanded in-line. In this case, the implementation can embed the activation record for the invoked operation within the caller's activation record. Invocations run at full machine speeds in this case. In-line implementations are appropriate when the implementation is known, when the operation is not recursive, and when the increased execution speed justifies more extensive recompilation when the in-lined operation changes.

Operations may reply early, in which case they continue concurrently with the caller. The called operation may continue after termination of the caller. Because of this, activation records cannot be allocated on a stack. This means that activation stack allocation must use more general, and more expensive, allocation methods. This allocation can use the typical caching techniques to minimize allocation costs. This form of implementation is the more general form.

Consider the case of an operation that issues its reply before termination. If the operation is "deadlock free", an implementation can save the reply value and delay delivery until the termination of the invocation. This would allow the procedure call mechanisms above to be used instead of the more general dynamic allocation of activation stacks. An operation is "deadlock free" the delaying of the reply cannot cause deadlock. This distinction is important, because if they were not deadlock free, the calling activation could deadlock with the called activation. If the reply can be delayed, the less expensive procedure call implementation of operation invocation can be used. When the reply cannot be delayed, the implementation must use the mere expensive general invocation. An implementation is known to be deadlock free when it does not block in any objects which it invokes. The programmer could also use a pragma to indicate when a operation is deadlock free.

The appropriate implementation of operation invocation can be bound late through the use of pragmas.

6.3 Remote Invocations

When the called entity is on the same node as the calling entity, the implementation can use any of the techniques described earlier. When the called entity is on a different node, but one to which remote memory accesses are available, then the implementation can treat the entity as in the local case and rely on shared memory to implement communication. This approach works best when operations are short or contain few references to entity data.

In the case of intensive reference to data within an entity, it is not efficient to execute the operation remotely. In the case of shared memory machines, the caller's node can allocate an activation stack on the callee's node and insert the activation into the callee's node's scheduling queue. This approach requires a concurrent allocator and scheduling queue.

In non-shared memory machines, remote manipulation of memory is not possible. In this case, the parent node needs to package the invocation description into a message that is sent to the child node. This child node then allocates an activation stack and places the activation in the scheduling queue. This strategy also applies to networked machines.

These choices may also be make late in program development, either by the compiler or by the programmer in the form of a pragma.

6.4 Scheduling Invocations

For activations that should run on a specific node, the implementation can insert the activation into the scheduling queue for that node. For shared memory machines, such as the Butterfly, it is feasible to have another node execute a locally generated activation. In this case, the implementation must maintain a global scheduling queue. Not all activations need be placed in the global queue. Each node is likely to work on any activations in its local queue before working on those in the global queue. This too can be specified with a pragma.

Only scheduling between different users or "jobs" need be preemptive. For many of the activations, a run until block implementation will be sufficient and most efficient.

One of the potential problems with the dynamic parallelism of the proposed model is an explosion in the number of activation records required to describe the current state of the computation. For example, consider a parallel tree search. Most current parallel programming languages encourage a style in which a fixed number of processes cooperate in allocating search nodes to processes. In contrast, the proposed model encourages a style in which a new concurrent invocation is created for each search node. For large search trees, the number of concurrent activations can become very large. Policies must exist to control this explosion of activations. One policy is to assign priority to the most recent activations. This, in effect, performs a depth first execution of the tree of possible parallelism. For a single processor, the storage required is therefore O(log n) in the number of possible concurrent activations instead of O(n). This depth first priority can be implemented efficiently by using scheduling *stacks* instead of queues.

6.5 Frequent Special Cases

Consider a future entity (as described in section 4.3) waiting on the completion of a remote invocation. The straightforward implementation of futures would require the allocation of a new activation stack on the local node because of the future's early reply and then the allocation of a new activation stack on a remote node for the execution of the specified operation. By recognizing the future as a special case, the implementation can allocate the future's activation stack on the remote node and use it to execute the specified operation.

In the case of parallel `for` loops, it is sometimes more efficient to chunk the iterations so that only groups of iterations are executed concurrently. Iterations within a group execute sequentially with standard sequential for loop mechanisms. An implementation of parallel `for` loops can take the number of iterations to chunk as a parameter. Straightforward user code can implement this optimization.

Another possible common case is parallel `for` loops where each iteration is placed in a global scheduling queue. This represents O(n) work for the node initiating the loop. An alternative implementation places an iteration description on the global scheduling queue. This distributes the iteration control among participating processors and reduces the initiator's work to O(1). The compiler would have to recognize the range type as a primitive in order to properly generate code for this optimization.

7 Research Plan

The proposed model of parallel programming is based on a single abstraction mechanism consisting of concurrent operations on objects, synchronous invocation with early reply, blocks (or closures) as operations on activations, and a value model of variables. These features combine to provide flexible coupling between caller and callee (which provides a concise base for different programming styles), uniform encapsulation and control (which allows parallelism at many levels within the program and allows the programmer to bind actual parallelism late in program development), extensive concurrency from both the sender and receiver sides of communication, and efficiency over a wide range of architectures.

A demonstration of the effectiveness of this model requires several example programs showing that several programming styles are naturally and concisely expressed within the model; showing that these programs may have exploitable parallelism at many levels; and demonstrating that the model may be efficiently implemented across a wide range of architectures. An implementation will demonstrate the latter two points.

7.1 Multiple Styles

Several substantial examples, including shared memory and message passing styles, will demonstrate that the entity model supports several styles of programming. The Gaussian elimination example has already been well studied at the University of Rochester [LeBlanc, 1986a]. This example will provide a means to directly compare the model with existing mechanisms. Some significant message passing program is needed to demonstrate the support for this style. This example should include pipes and various buffers. The specific example has not been chosen. A concurrent hash table will provide direct evidence for the support shared memory styles [Ellis, 1985]. Batcher's Sort should provide a good example of a combination of styles.

7.2 Multiple Levels and Late Binding of Parallelism

Programs in the entity model will contain exploitable parallelism at many levels of abstraction within the program. The capability to bind parallelism late in program development enables the programmer to choose the level at which parallelism will actually take place. This allows the programmer to tune the program to a specific architecture and execution profile. A demonstration of this requires several examples.

One such example is the subgraph isomorphism problem [Costanzo *et al.*, 1986]. The Chrysalis implementation was essentially constrained to coarse-grained search parallelism up front. An entity-based program for this problem will be able to parallelize among the various constraint propagation mechanisms within a search node, and within the matrix computations for each constraint mechanism. This will also enable convenient experiments on the most appropriate way to parallelize a program.

To demonstrate that a language processor can take advantage of the many levels while avoiding excessive cost for those levels it does not parallelize, invocation must be implementable many different ways. Known possible invocation implementations are inline, procedure call, and asynchronous invocation. These were discussed in the previous chapter.

7.3 Multiple Architectures

To demonstrate that programs written in the entity model can be efficiently mapped to a number of different architectures, one must provide arguments that show that at least some programs have levels of parallelism and communication appropriate to different architectures and that different levels of parallelism and communication have efficient implementations on several architectures. However, some styles of programming may have sufficient internal communication to prevent efficiently mapping them to any architecture in which communication is expensive.

An implementation on the Butterfly can be constrained to emulate a number of different architectures. By varying which implementations are available to the code generator, the Butterfly can simulate different architectures from a single processor machine, its native non-uniform access shared memory machine, and a loosely coupled network. For instance, disallowing remote-memory accesses in the code generator of the compiler essentially treats the Butterfly as a loosely-coupled multiprocessor, such as the Hypercube.

7.4 Implementation Outline

A simple implementation of the model on the Butterfly can demonstrate the points discussed above. This implementation will not be a production language. In many respects it will be very crude. The compiler complexity is on the order of that expected in a compiler course. The notation described in section 4.1 is mostly complete. Issues that remain in turning the notation into a language include the precise semantics of value copying and entity initialization.

The compiler will rely on pragmas to indicate the appropriate implementations. It will do no analysis.

Mechanisms for controlling where entities are to be allocated, *i.e.* on which processor, have not been addressed.

The main primitive type will be the integer. The compiler will have to recognize it and generate the appropriate code. These will be generated in-line. Boolean values will be overloaded onto integers in the manner that C [Kernighan and Ritchie, 1978] does. Other primitive types are needed for synchronization. These include semaphores and read-write locks. The read-write locks allow integration of Instant-Replay [LeBlanc and Mellor-Crummey, 1987] with minimal programming cost.

Pointers will provide for indirect reference. The specific mechanisms to indicate pointer dereference have not been chosen. Pointers will be typed by the types to which they refer. Pointers may refer to entities and ports. Port references will be implemented as a pointer to data and a pointer to code.

The structuring mechanisms in the prototype implementation include only entities, records, and arrays. Records and arrays are a special case of entities, but require special compiler recognition to implement effectively.

Control constructs will take the form of code blocks and operations on integers. Some simple pre-defined control constructs, such as the `if` operation, will be compiled in-line. Code blocks will provide some of the more complicated code generation because they are essentially nested procedures and must maintain a static chain of activation records.

The prototype implementation will not attempt to catch dangling references to entities or activations.

The model relies on generics, but a prototype implementation may use textual substitution to implement generics. For instance, a `sed` command can turn a generic definition into an instance definition. This does not represent a good programming environment, but will be sufficient to demonstrate the thesis.

The construction of a uniform address space under Chrysalis avoids issues in name translation, which are not relevant to parallel programming, though they are relevant to distributed programming.

The invocation mechanisms will provide the hardest code generation problem. Procedure calls are relatively simple to implement. A shared address space coupled with a procedure call implementation of invocation, immediately supports shared-memory access. Local asynchronous invocations require dynamic allocation of activation stacks. Remote asynchronous invocations can have much the same implementation, but with remote operations on stack allocators and scheduling queues. This approach does not interrupt the remote processor. An interrupt-driven message passing implementation of remote invocation can emulate loosely coupled systems.

The scheduling of activations will be accomplished with simple ready stacks. The implementation will not use sophisticated scheduling algorithms.

7.5 Schedule

Addressing the language definition issues remaining will take one month. This portion will be finished when a language definition document is complete. Since the language is simple, the document should not be long. The language will evolve as the project progresses.

Writing and documenting the example programs will take three months. I expect these programs to be of moderate size. This portion is complete when the listings and their descriptions are complete.

The initial implementation and debugging of the model will take five months. The implementation will include the features outlined in the previous section. This portion is complete when simple test programs run reliably.

The integration of the example programs with the implementation will take another three months. This portion is complete when the sample programs have been executed under different architectural simulations, and the results have been documented. The work involved in tuning the programs to different architectures must be documented.

The final writeup of the thesis will take four months. This portion is complete when the thesis presents a convincing argument that the proposed model enables multiple programming styles, late binding of parallelism, multiple levels of parallelism within a program, and effective implementation on a range of multiprocessor architectures.

This schedule totals sixteen months.

A Emerald Examples

A.1 Receiver1

```
object receiver1
  export accept1, accept2

  ... variables ...
  const queuerec == record queuerec
    var kind : Integer
    var data : Any
  end queuerec
  const queue == Queue.of[ queuerec ].empty

  const message1rec == record message1rec
    var param1 : Integer
    var param2 : String
  end message1rec
  operation accept1[ params : message1rec ]
    var toqueue : queuerec <- new queuerec
    toqueue.kind <- 1
    toqueue.data <- params
    queue.insert[ toqueue ]
  end accept1
  operation handle1[ params : message1rec ]
    ... handle message ...
  end handle1
```

```
  const message2rec == record message2rec
    var param1 : String
    var param2 : String
    var param3 : Boolean
  end message2rec
  operation accept2[ params : message2rec ]
    var toqueue : queuerec <- new queuerec
    toqueue.kind <- 2
    toqueue.data <- params
    queue.insert[ params ]
  end accept2
  operation handle2[ params : message2rec ]
    ... handle message ...
  end handle2

  initially
    ... initialize ...
  end initially

  process
    var qr : queuerec
    loop
      qr <- queue.remove
      if qr.kind = 1 then
        handle1[ qr.data ]
      else
        handle2[ qr.data ]
      end if
    end loop
  end process
end receiver1
```

A.2 Receiver2

```
object receiver2
  export accept1, accept2, handle1, handle2

  monitor
    ... variables ...
    operation handle1[ param1 : Integer, param2 : String ]
      ... handle message ...
    end handle1
    operation handle2[ param1 : String, param2 : String,
                        param3 : Boolean ]
      ... handle message ...
    end handle2
  end monitor

  operation accept1[ param1 : Integer, param2 : String ]
    const rself == this
    const handler == object handler
      process
        rself.handle1[ param1, param2 ]
      end process
    end handler
  end accept1

  operation accept2[ param1 : String, param2 : String,
                     param3 : Boolean ]
    const rself == this
    const handler == object handler
      process
        rself.handle2[ param1, param2, param3 ]
      end process
    end handler
  end accept2

  initially
    ... initialize ...
  end initially
end receiver2
```

Bibliography

[Accetta *et al.*, 1986] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development," In *Proceedings of the Summer 1986 Usenix Conference*, 1986.

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

[Agha, 1986b] Gul A. Agha, "An Overview of Actor Languages," *ACM SIGPLAN Notices*, 21(10):58–67, October 1986.

[Andrews *et al.*, 1986] Gregory R. Andrews, Ronald A. Olsson, Michael H. Coffin, Irving J. P. Elshoff, Kelvin Nilsen, and Titus Purdin, "An Overview of the SR Language and Implementation," Technical Report 86–6a, Department of Computer Science, University of Arizona, Tuscon, Arizona, 85721, June 1986.

[ANSI, 1978] American National Standards Institute, *American National Standard Programming Language: FORTRAN, X3.9-1978*, 1978.

[Atkinson *et al.*, 1983] M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm, and R. Morrison, "An Approach to Persistent Programming," *The Computer Journal*, 26(4), 1983.

[Backus, 1978] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Syle and Its Algebra of Programs," *Communications of the ACM*, 21(8):613–641, August 1978.

[Baldwin, 1987] Doug Baldwin, "Why We Can't Program Multiprocessors the Way We're Trying to Do It Now," Technical Report 224, Department of Computer Science, University of Rochester, August 1987.

[Baldwin and Quiroz, 1987] Doug Baldwin and César A. Quiroz, "Design of the CONSUL Programming Language," Technical Report 208, Department of Computer Science, University of Rochester, February 1987.

[BBN, 1985a] BBN Laboratories, Cambridge, Massachusetts, *Butterfly Parallel Processor Overview*, June 1985.

[BBN, 1985b] BBN Laboratories, Cambridge, Massachusetts, *Chrysalis Programmer's Manual*, June 1985.

[BBN, 1985c] BBN Laboratories, Cambridge, Massachusetts, *The Uniform System Approach To Programming the Butterfly Parallel Processor*, October 1985.

[Bershad *et al.*, 1987] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming," Technical Report 87-09-01, Department of Computer Science, University of Washington, September 1987.

[Black *et al.*, 1986a] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry Levy, "Object Structure in the Emerald System," In *Proceedings of 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, September 1986, reprinted in ACM SIGPLAN Notices 21(11), November 1986.

[Black *et al.*, 1986b] Andrew P. Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, December 1986.

[Blelloch, 1986] Guy E. Blelloch, "AFL-1: A Programming Language for Massively Concurrent Computers," Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1986.

[Brinch Hansen, 1978] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, 21(11):934–941, November 1978.

[Bukys, 1986] Liudvikas Bukys, "Connected Component Labeling and Border Following on the BBN Butterfly Parallel Processor," Butterfly Project Report 11, Department of Computer Science, University of Rochester, October 1986.

[Cook, 1980] Robert P. Cook, "*MOD—A Language for Distributed Programming," *IEEE Transactions on Software Engineering*, SE-6(6):563–571, November 1980.

[Cooper, 1983] Doug Cooper, *Standard Pascal: User Reference Manual*, W. W. Norton & Company Inc., 500 Fifth Avenue, New York, New York 10110, 1983.

[Costanzo *et al.*, 1986] John Costanzo, Lawrence Crowl, Laura Sanchis, and Mandayam Srinivas, "Subgraph Isomorphism on the BBN Butterfly Multiprocessor," Butterfly Project Report 14, Department of Computer Science, University of Rochester, Rochester, New York, 14627-0226, October 1986.

[Crowl, 1986] Lawrence A. Crowl, "Chrysalis++," Butterfly Project Report 15, Department of Computer Science, University of Rochester, Rochester, New York, 14627-0226, December 1986.

[Crowl, 1987] Lawrence A. Crowl, "An Interface Between Object-Oriented Systems," Technical Report 211, Department of Computer Science, University of Rochester, Rochester, New York, 14627-0226, April 1987.

[Crowl, 1988] Lawrence A. Crowl, "Shared Memory Multiprocessors and Sequential Programming Languages: A Case Study," In Bruce D. Shriver, editor, *Proceedings of the Twenty-First Annual Hawaii International Conference on Systems Sciences*, volume II, pages 103–108. University of Hawaii, IEEE Computer Society Press, January 1988.

[Ellis, 1982] Carla S. Ellis, "Extendible Hashing for Concurrent Operations in Distributed Data," Technical Report 110, Department of Computer Science, University of Rochester, October 1982.

[Ellis, 1985] Carla S. Ellis, "Concurrency and Linear Hashing," Technical Report 151, Department of Computer Science, University of Rochester, March 1985.

[Ellis, 1986] John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1986.

[Feldman, 1979] Jerome A. Feldman, "High Level Programming for Distributed Computing," *Communications of the ACM*, 22(6):353–368, June 1979.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Greif *et al.*, 1986] I. Greif, R. Seliger, and W. Weihl, "Atomic Data Abstractions in Distributed Collaborative Editing Systems," In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986.

[Halstead, 1986] Robert H. Halstead, Jr., "Parallel Symbolic Computing," *Computer*, 19(8):35–43, August 1986.

[Hewitt and Atkinson, 1977] C. Hewitt and R. Atkinson, "Parallelism and Synchronization in Actor Systems," In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 267–280, Los Angeles, California, January 1977.

[Hewitt, 1977] Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[Hoare, 1978] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, 21(8):666–677, August 1978.

[Hughes, 1984] John Hughes, "Why Functional Programming Matters," Technical Report 16, Methodology Group, University of Goteborg and Chalmers, University of Technology, Goteborg, Sweden, 1984, ISSN 0282-2083.

[Hutchinson, 1987] Norman C. Hutchinson, "Emerald: An Object-Based Language for Distributed Programming," Technical Report 87-01-01, Department of Computer Science, University of Washington, January 1987, Ph.D. Dissertation.

[Ishikawa and Tokoro, 1986] Yutaka Ishikawa and Mario Tokoro, "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its features and Implementation," *ACM SIGPLAN Notices*, 21(11):232–241, November 1986, OOPSLA'86.

[Jensen and Wirth, 1975] Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, second edition, 1975.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1978.

[Lamport, 1986a] Leslie Lamport, "The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication," *Journal of the ACM*, 33(2):313–326, April 1986.

[Lamport, 1986b] Leslie Lamport, "The Mutual Exclusion Problem: Part II—Statement and Solutions," *Journal of the ACM*, 33(2):313–326, April 1986.

[Lampson and Redell, 1980] Butler W. Lampson and David D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, 23(2):105–118, February 1980.

[Lang, 1982] Charles R. Lang, Jr., "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture," Technical Report 5014, Computer Science Department, California Institute of Technology, Pasadena, California, May 1982.

[LeBlanc, 1986a] Thomas J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986, also [LeBlanc, 1986b].

[LeBlanc, 1986b] Thomas J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," Butterfly Project Report 3, Department of Computer Science, University of Rochester, Rochester, New York, 14627-0226, January 1986, also [LeBlanc, 1986a].

[LeBlanc *et al.*, 1986] Thomas J. LeBlanc, Neal M. Gafter, and Takahide Ohkami, "SMP: A Message-Based Programming Environment for the BBN Butterfly," Butterfly Project Report 8, Department of Computer Science, University of Rochester, July 1986.

[LeBlanc and Mellor-Crummey, 1987] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, April 1987.

[Liskov, 1979] Barbara H. Liskov, "Primitives for Distributed Computing," In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 33–42. Association for Computing Machinery, December 1979.

[Liskov *et al.*, 1984] Barbara H. Liskov, Maurice P. Herlihy, and Lucy Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," Programming Methodology Group Memo 41-1, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1984.

[Liskov and Scheifler, 1983] Barbara H. Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Liskov *et al.*, 1977] Barbara H. Liskov, Alan Snyder, R. R. Atkinson, and J. C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, 20(8):564–576, August 1977.

[Nauer, 1963] P. Nauer, "Report on the Algorithmic Language Algol 60," *Communications of the ACM*, 6(1):1–7, January 1963.

[Parnas and Siewiorek, 1975] D. L. Parnas and D. P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Communications of the ACM*, 18(7):401–408, July 1975.

[Rashid and Robertson, 1981] Richard F. Rashid and George G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75, 1981.

[Scott, 1987] Michael L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[Steele, 1984] Guy L. Steele, Jr., *Common Lisp: The Language*, Digital Press, 1984.

[Steinberg *et al.*, 1986] Seth Steinberg, Don Allen, Laura Bagnall, and Curtis Scott, "The Butterfly Lisp System," In *Proceedings of the 1986 AAAI*, volume 2, pages 730–734, Philidelphia, Pennsylvania, August 1986.

[Strom and Yemini, 1983] Robert E. Strom and Shaula Alexander Yemini, "NIL: An Integrated Language and System for Distributed Programming," Research Report RC 9949, International Business Machines Corporation Research Division, April 1983.

[U. S. DoD, 1983] United States Department of Defense, Washington D. C., *Reference Manual for the Ada Programming Language*, June 1983, ANSI/MIL-STD-1815A.

[Wadge and Ashcroft, 1985] William W. Wadge and Edward A. Ashcroft, *Lucid, the Dataflow Programming Language*, A. P. I. C. Studies in Data Processing. Academic Press, London, 1985.

[Xerox, 1984] Xerox Corporation, Palo Alto, California, *Mesa Language Manual*, November 1984.

[Yokote and Tokoro, 1986] Yasuhiko Yokote and Mario Tokoro, "The Design and Implementation of ConcurrentSmalltalk," *ACM SIGPLAN Notices*, 21(11):331–340, November 1986, OOPSLA'86.

[Yonezawa *et al.*, 1986] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama, "Object-Oriented Concurrent Programming in ABCL/1," *ACM SIGPLAN Notices*, 21(11):258–268, November 1986, OOPSLA'86.