Architectural Adaptability in Parallel Programming

by

Lawrence Alan Crowl

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by Thomas J. LeBlanc

Department of Computer Science

University of Rochester

Rochester, New York

September 1995

Curriculum Vitae

Lawrence Alan Crowl was born on the 25th of July 1959 in Sacramento, California. Since then he has lived in Kansas, Florida, New Hampshire, Alabama, California, New Mexico, Rheinland-Pfalz (Germany), Ohio, Virginia, Colorado, and New York. His pursuit of a doctorate at the University of Rochester caused his longest stay in any one state!

Starting in September 1977, Lawrence attended Denison University in Granville, Ohio. There he served as a member of the Computer Center Advisory Committee and the Special Committee on the Future of Computer Service, and worked for the Computer Center as a programmer. He was inducted into the Sigma Xi (Scientific Research), Pi Mu Epsilon (Mathematics) and Sigma Pi Sigma (Physics) honoraries. He received the Gilpatrick Award for Excellence in Mathematics, and was on the Dean's List two years. In May 1981, he received a Bachelor of Science Magne cum Laude in Computer Science and Physics. His Honors Thesis was "A Terminal Oriented Master/Slave Operating System".

In September 1981, Lawrence started graduate school in computer science at the Virginia Polytechnic Institute and State University in Blacksburg, Virginia. There he worked as a graduate teaching assistant for the Computer Science Department. He was inducted into the Upsilon Pi Epsilon (Computer Science) honorary. In July 1983, he received a Master of Science in Computer Science and Applications. His Master's Thesis was "A Macro System for English-Like Commands".

From March 1983 through August 1985, Lawrence worked as a software engineer for Hewlett-Packard Company in Loveland, Colorado. There he developed system software for an integrated-circuit tester and a printed-circuit-board tester.

In September 1985, Lawrence entered the University of Rochester Computer Science Department. He worked primarily as a research assistant, but also as teaching assistant for the Problem Seminar (the graduate immigration course) and Programming Languages.

Lawrence is a member of the Sigma Xi Scientific Research Society, the Association for Computing Machinery, and its Special Interest Group on Programming Languages.

Acknowledgments

First, I would like to thank Thomas J. LeBlanc, my advisor, for his untiring efforts in helping me separate the wheat from the chaff in this dissertation. I would also like to thank my committee Douglas L. Baldwin, Robert J. Fowler, Michael L. Scott, and Edward L. Titlebaum, for their effort.

Three students deserve special recognition for their continuing aid in exploring the ideas in this dissertation. They are Alan L. Cox, John M. Mellor-Crummey, and César A. Quiroz.

I also thank my family for their long-distance support and encouragement.

This work was supported by the National Science Foundation under research grants CCR-8320136 and CDA-8822724, the Office of Naval Research under research contract N00014-87-K-0548, and the Office of Naval Research and Defense Advanced Research Projects Agency under research contract N00014-82-K-0193. The Government has certain rights in this material.

Abstract

To create a parallel program, programmers must decide what parallelism to exploit, and choose the associated data distribution and communication. Since a typical algorithm has much more potential parallelism than any single architecture can effectively exploit, programmers usually express only the exploitation of parallelism appropriate to a single machine. Unfortunately, parallel architectures vary widely. A program that executes efficiently on one architecture may execute badly, if at all, on another architecture. To port such a program to a new architecture, we must rewrite the program to remove any ineffective parallelism, to introduce any parallelism appropriate for the new machine, to re-distribute data and processing, and to alter the form of communication.

Architectural adaptability is the ease with which programmers can tune or port a program to a different architecture. The thesis of this dissertation is that control abstraction is fundamental to architectural adaptability for parallel programs. With control abstraction, we can define and use a rich variety of control constructs to represent an algorithm's potential parallelism. Since control abstraction separates the definition of a construct from its implementation, a construct may have several different implementations, each providing different exploitations of parallelism. By selecting an implementation for each use of a control construct with annotations, we can vary the parallelism we choose to exploit without otherwise changing the source code.

We present Matroshka, a programming model that supports architectural adaptability in parallel programs through object-based data abstraction and closure-based control abstraction. Using the model, we develop several working example programs, and show that the example programs adapt well to different architectures. We also outline a programming method based on abstraction. To show the implementation feasibility of our approach, we describe a prototype language based on Matroshka, describe its implementation, and compare the performance of the prototype with existing programs.

Table of Contents

List of Tables

List of Figures

1 Introduction

Likewise, when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.
— General L. F. Manabrea, 1842; referring to Charles Babbage's Analytical Engine, the first computer.

Most current computers execute sequentially, one operation at a time. The great speed at which these computers execute their operations gives them their computing power. Unfortunately, further increases in the speed of execution are becoming expensive; so there is a practical limit on the computational power of cost-effective sequential computers. We can get more cost-effective computing power by using computers that execute many operations at a time, in parallel. Unlike sequential computers, there are many different ways to organize parallel computers. Furthermore, programmers often assume one particular organization when writing parallel programs. While the resulting programs execute efficiently on one computer, they often execute poorly on another. Rewriting the programs to execute efficiently on the second computer is often as difficult as starting over from scratch. This dissertation describes how to write programs that take little effort to make them execute efficiently on a wide variety of parallel computers.

1.1 Architectures and Programming

In developing a computer program, programmers have two tasks. First, they must identify an algorithm for solving the problem; and second, they must implement that algorithm on the computer at hand. For the past forty years, nearly all computers have had a von Neumann architecture [Burks *et al.*, 1946], in which operations proceed sequentially. Programs that execute efficiently on one von Neumann computer will almost always execute efficiently on another. As a result of this stability in architecture, most programmers and programming languages can safely assume a von Neumann architecture. The assumption has become so safe that it is implicit in most programs and programming languages. Indeed, the reliance on sequential execution is so prevalent that the term 'algorithm' usually means a sequential algorithm unless explicitly stated otherwise.

The von Neumann architecture has remained prevalent because manufacturers have been able to increase the speed of computers by increasing the speed of their electrical components, and by using parallelism in the implementation of the architecture. Unfortunately, both of these techniques are reaching their limits. First, increasing the speed of the basic electrical components is progressively more expensive. Second, the frequency of conditional branches within most von Neumann programs limits the effectiveness of parallelism in the implementation of a von Neumann architecture. We are now reaching the limits at which we can cost-effectively provide increased computing power solely though faster implementations of the von Neumann architecture. As computational speeds increase, architectures that provide parallelism will be more cost effective than the von Neumann architecture.

In contrast to sequential computers, parallel computers have a wide variety of architectures. They may provide a single instruction stream that operates on many data streams (SIMD), or they may provide many independent instruction and data streams (MIMD). For example, the Illiac-IV broadcasts the same instruction to 64 processors while each Cm* processor executes instructions independently. Existing parallel computers provide from one (*e.g.* the Cray 1) to 65536 (*e.g.* the Connection Machine) processors. Computers may provide information storage in three different ways. They may provide storage that all processors access equally (*e.g.* the Sequent Balance); they may split the storage among processors so that accessing another processor's portion is more expensive (*e.g.* the BBN Butterfly); or they may not provide any access to another processor's portion (*e.g.* the Hypercube). When processors cannot directly access non-local storage, they must communicate with other processors for the necessary information. Computers may communicate via high-speed inter-processor networks, medium-speed local-area networks, and low-speed long-distance networks. Any single difference in these characteristics leads to qualitatively different architectures, so there are many potential architectures.

Although an algorithm may have an efficient implementation on a wide range of architectures, each class of architecture may exploit a different subset of the parallelism inherent in the algorithm. Unfortunately, the implementation of a parallel algorithm on one architecture may provide little leverage in finding an efficient implementation on another architecture. To implement an algorithm on a particular machine, we must do three things.

Identify and Exploit Parallelism: Algorithms generally contain more *potential* parallelism than any one machine can effectively *exploit*; so we must select the subset of potential parallelism that we wish to exploit. This subset depends on the number of processors, the overhead associated with starting a parallel activity, and the overhead associated with any necessary synchronization. Since these factors differ depending on the architecture, the appropriate exploitation of parallelism will depend on the architecture. For example, the Transputer has hardware support for quickly creating and managing parallel activities, so programs executing on the Transputer can efficiently manage more parallel activities than programs on many other machines.

Distribute Data and Processing: Parallel architectures can execute more quickly when the data a processor needs is close to the processor. For example, programs on the BBN Butterfly can access memory local to the processor five to fifteen times faster than memory local to another processor. When we distribute data and computational tasks so that tasks that share data are close to the data and to each other, the overall efficiency of the program will be greater.

Choose Communication: Communication in shared-memory multiprocessors (*e.g.* the Sequent Balance) can be several orders of magnitude faster than communication in distributed memory machines (*e.g.* the Hypercube). The cost of communication affects the parallelism that programmers can exploit efficiently. Some architectures provide several forms of communication so that programmers can exploit a wider range of parallelism. Programmers must choose the communication mechanisms that are appropriate to the parallelism exploited.

Because of the wide variety of parallel architectures and the many possible interleavings of statement executions, implementing a parallel algorithm is a difficult problem. There are two primary approaches to solving this problem. The first approach relies on the programmer to express explicitly the parallelism in an algorithm and its implementation on an architecture. The second approach relies on the programming language translator to accept non-parallel descriptions of an algorithm (sequential or declarative) and find the appropriate parallelism for a machine. While the second approach results in less work for the programmer, current translators produce programs that execute slowly relative to explicitly parallel programs. We use parallel computers for the speed advantage they provide over sequential computers. However, parallel computers have modest potential [Snyder, 1986], at best they can improve computational speed linearly in the number of processors. So, users are often willing to invest considerable effort in making efficient use of parallel computers. Any language translator that introduces much inefficiency will exclude the set of users that care most about performance — exactly those users of parallel computers. Because of the desire for efficient execution, this dissertation concentrates on explicitly parallel imperative languages.

1.2 Architectural Adaptability

When writing an explicitly parallel program, programmers typically limit consideration to the parallelism in the algorithm that a given machine can effectively exploit, and ignore any other potential parallelism. The resulting programs embed assumptions about the effective granularity of parallelism, the distribution of processing and data, and the cost of communication and synchronization. While this approach may result in an efficient implementation of the algorithm under a single set of assumptions, the program is difficult to adapt to a different set of assumptions because the distinction between potential and exploited parallelism has been lost. All that remains in the program is a description of the parallelism that is most appropriate for our original assumptions about the underlying machine. When an architecture violates any of these assumptions, the program must be restructured to avoid a potentially serious loss of performance. This restructuring can be complex, because the underlying assumptions are rarely explicit, and the ramifications of each assumption are difficult to discern. For example, programs written for a shared-memory machine will communicate through variable access without explicitly noting the resulting communication. Emulating this shared memory access on a computer without shared memory may or may not be effective, depending on the program. Assuming characteristics of a given machine in the development of a parallel program will limit the range of machines for which the program is efficient.

We might want to change the architectural assumptions in a parallel program for two reasons:

Tuning: We may not be able to predict *a priori* those sources of parallelism in an algorithm that are most appropriate for an architecture (or a particular class of input values). Changing an incorrect exploitation of parallelism can be a complex, *ad hoc* task, similar to the problem of changing data representations in a program lacking data abstraction.

Porting: We may wish to port programs from one architecture to another and to vary the number of processors in use. Since parallel architectures vary widely, different implementations of the same program will usually exploit different opportunities for parallelism. Uncovering and exploiting these opportunities can result in a massive restructuring of the program.

Architectural adaptability is the ease with which programs can be tuned or ported to different architectures. We can measure architectural adaptability by the extent of source code changes necessary to adapt a program to an architecture, and the intellectual effort required to select those changes.

A programming system provides *architectural independence* over a range of architectures when it automatically selects the exploitation of parallelism for a particular architecture in that range, and the programmer makes no architecture-specific changes. Given the difficulty of achieving true architectural independence, this dissertation relies on a simple mechanism, *annotations*, that enables the programmer to select the exploited parallelism without significant changes to the rest of the source program. Changing annotations will usually suffice to adapt a program to an architecture. Where changes to source code are necessary, we would like to minimize both the number of changes and the effort required to make the changes.

1.3 Statement of Thesis

Achieving architectural adaptability is easier when the program separates the expression of an algorithm from its implementation on a given machine. For instance, in explicitly parallel imperative programs we need to specify the potential parallelism in an algorithm and then separately specify its exploitation in a given implementation. With this separation, we can specify the potential parallelism during program design, and later choose an implementation during program debugging and tuning.

The separation of the specification of potential parallelism from its implementation is an example of *abstraction*. In programming, abstraction is the *process* of separating the use of something from its implementation. Programming language designers almost necessarily use abstraction in the development and definition of their languages. While this is an effective use of abstraction, the process of abstraction is most useful when application programmers can continue the process in the development of their programs. In applying abstraction to parallel programming, we can use abstractions to represent potential parallelism, distribution and communication, and then use implementations of those abstractions appropriate for a given machine. While any given implementation may exploit only a small subset of the potential parallelism, the program expresses all potential parallelism.

Explicitly parallel imperative programs use control flow constructs, such as `fork`, `cobegin`, and parallel `for`, to introduce parallel execution. Since the expression of parallelism in these programs is fundamentally an issue of control flow, control abstraction should aid architectural adaptability. Control abstraction is the process of separating the use of a control construct from its implementation. In particular, control abstraction can separate the semantics of statement sequencing from the implementation of statement sequencing.

Control abstraction aids architectural adaptability in three ways, corresponding to our original list of implementation problems.

Identify and Exploit Parallelism: We can use control abstraction to define control constructs that represent an algorithm's potential parallelism, and define several implementations for those constructs that exploit different subsets of the potential. The *algorithm* determines the control constructs used to represent potential parallelism; the *architecture* determines the implementations used to exploit parallelism.

Distribute Data and Processing: We can use data abstraction to define data structures that may be distributed, and exploit different distributions with different implementations of the data abstractions. However, data distribution alone is not enough, we must also distribute processing. Again, we can use control abstraction to define control operations that distribute processing and to define control operations on distributed data structures that distribute the processing with the data.

Choose Communication: We can use argument passing in procedural abstraction (a form of control abstraction) to represent potential communication. We can then choose different exploitations of communication by selecting the appropriate implementation of procedure invocation. Implementations include the typical machine branch implementation and remote procedure call implemented with messages.

Control abstraction is a central part of a general solution to each implementation. We intend to show that *control abstraction is an effective means for achieving architectural adaptability in explicitly parallel imperative programs.*

1.4 Dissertation Overview

A parallel programming system must provide more than just control abstraction to be effective. To test and support the ideas presented in this dissertation, we

- designed the Matroshka (Матрёшка)[1] parallel programming model to support architectural adaptability in parallel programming;
- designed the Natasha (Ната́ша)[2] prototype programming language using the Matroshka model;
- implemented a compiler and runtime library for Natasha;
- programmed several example applications; and
- executed these examples to test their effectiveness.

Chapter 2 discusses some early parallel languages, the problems they present for architectural adaptability, and related work in solving these problems. Chapter 3 introduces the Matroshka model for parallel programming, with examples using the Natasha prototype programming language. (Appendix A provides the complete Natasha language definition.) Chapter 4 introduces control abstraction and its application to architectural adaptability. Chapter 5 presents several extended examples of programming for architectural adaptability. Chapter 6 discusses a method for achieving architectural adaptability with abstraction. Chapter 7 presents the prototype implementation of Natasha and the optimizations that make it competitive in performance with standard sequential languages. Finally, chapter 8 presents the conclusions.

[1]Matroshka (also transliterated as Matrëška) are the wooden Russian dolls where the smallest nests within the next smallest and so on. They look somewhat like squat bowling pins and are painted to depict Russian peasant women.

[2]Natasha (also transliterated as Natáša) is an instance of a Russian peasant woman.

2 Related Work

Pereant, inquit, qui ante nos nostra dixerunt.
[Confound those who have said our remarks before us.]
— Aelius Donatus, fourth century A.D.

To create a parallel program, a programmer must decide what parallelism to exploit, how to distribute data and processing among processors, and how to communicate between parallel tasks. This chapter first discusses early programming language support in solving these problems and then presents several techniques that address each of these problems in the context of architectural adaptability.

2.1 Early Parallel Languages

Early parallel programming languages were intended as tools to program a specific architectural model rather than as a general means for specifying parallel algorithms. As such, early languages provided separate mechanisms that represented different features of the class of real machines for which the languages were intended. These mechanisms reflected the parallelism, distribution and communication costs of their architectures. For example, Concurrent Pascal [Brinch Hansen, 1975] provided `cobegin ... coend` construct to represent the concurrent execution of statements. Statements shared storage at a fine grain. This reflected Concurrent Pascal's expected used as a language for concurrent programming on a uniprocessor. Distributed computing, in which computers share no storage but communicate by passing messages, attracted many languages. These include PLITS [Feldman, 1979], which provided extensive facilities for handling messages between autonomous processes, and Distributed Processes [Brinch Hansen, 1978], which provided an early form of remote procedure call. In essence, these languages presented virtual machines that closely matched real machines. The advantage these languages provided was that the virtual machines presented by the languages were easier to program than the real machines.

In a distributed system, communication between different processors typically costs two orders of magnitude more than communication within a processor. Many distributed programming languages such as PLITS [Feldman, 1979] and *MOD [Cook, 1980], make distributed objects visible within the language under the assumption that programmers will manage visible costs more effectively than invisible costs. The resulting programs often execute efficiently only on that architecture or architectures that can emulate it efficiently. This issue is especially important because, unlike the von Neumann architecture for sequential computers, there is no generally accepted archetype for parallel computers. For example, programming languages based on shared memory do not execute well on distributed systems.

2.1.1 The Multiple Mechanism Problem

When parallel and sequential mechanisms are distinct, programmers must decide to implement a given object either as a parallel object or as a sequential object early in program development. Because communication among distributed objects is expensive, application programmers tend to make distributed objects large to minimize the interactions between them and the resulting overhead. On the other hand, programmers of libraries do not know the context in which their code will be used, so they often choose a higher-cost, more general, parallel implementation rather than a cheaper sequential implementation [Greif *et al.*, 1986]. This, in turn, inhibits use of the library by application programmers concerned with performance. Libraries must provide general purpose abstractions in both mechanisms to provide programmers with the incentive to use them. Multiple mechanisms tend to discourage the specification of parallelism and the construction of libraries.

The difference between local and remote communication costs in multiprocessors is much lower than in distributed systems, which encourages more frequent communication within programs. Unfortunately, the appropriate binding of parallelism to program components in such an environment is often not obvious before doing performance experiments on completed code. For example, an algorithm for finding subgraph isomorphisms using constraint propagation has potential parallelism in many places. The two primary sources of parallelism are in searching the constraint tree, and in the matrix calculations that prune the tree at each node. The coarsest parallelism is at the tree search, so we expect it to have the least overhead. However, tree search results in speculative parallelism, so there may substantial wasted work [Costanzo *et al.*, 1986]. Parallelizing the matrix operations may provide better performance in spite of the higher overhead. In multiprocessors, predicting the appropriate mechanism among many may be difficult.

When programmers must choose parallelism early in program development, fixing an incorrect choice or porting the program to a different architecture involves substantial changes to the program. If a choice of a mechanism is incorrect, the programmer must recode portions of the program and reintegrate it into the remainder of the program. In addition, the mechanism used to implement an abstraction is often visible at the points where the abstraction is used. Because the use of an abstraction may be distributed throughout the program, a change in mechanism could involve rewriting much of the program. Programmers will only make such changes under extreme circumstances. This inhibits tuning the program to make optimal use of the parallelism available and severely handicaps someone attempting to port the program to another architecture. This latter problem is difficult enough so that programmers often write another program based on the algorithm of the original program rather than port the program itself.

When language mechanisms enable programmers to bind parallelism late in program development, choosing the granularity and location of parallelism becomes part of the optimization effort, rather than the algorithm development effort.

2.2 Exploiting Parallelism

Recent approaches to the problem of specifying and exploiting parallelism typically rely on a general strategy of representing much of the potential parallelism in an algorithm, and then selecting an appropriate subset.[1] This strategy is a significant departure from earlier practice, where programs described only the parallelism exploited on a given machine, and therefore were difficult to adapt to a new architecture.

2.2.1 Parallel Function Evaluation

This dissertation focuses on explicitly parallel imperative languages because of their performance advantage over functional languages. However, work on architectural adaptability in functional languages can provide incites useful to explicitly parallel programming. Functional programs have no side effects, so expressions may be evaluated in any order. Therefore we can evaluate all expressions in parallel, and parallelism is implicit in functional programs. There are two sources of parallelism in function evaluation, parallel evaluation of multiple arguments to a function and lazy evaluation of the value of a function. Owing to the difficulty of automatically finding and exploiting the optimal sources of parallelism in a functional program, several researchers have suggested the use of annotations to specify lazy, eager, parallel, and distributed function evaluation [Burton, 1984; Halstead, 1985; Hudak, 1986].

ParAlfl [Hudak, 1986; Hudak, 1988] is a functional language that provides annotations to select eager evaluation over lazy evaluation, resulting in parallel execution, and to map expression evaluation to processors. A *mapped expression* in ParAlfl can dynamically select the processor on which it executes. An *eager expression* executes in parallel with its surrounding context. By using a combination of eager and mapped expressions, a programmer can select the parallelism to be exploited and map it to the underlying architecture. The use of mapped and eager annotations does not change the meaning of the program, which in a functional programming language does not depend on the evaluation order. Thus, ParAlfl achieves a significant degree of architectural adaptability, requiring only changes to annotations to port a program between architectures. ParAlfl achieves this goal only in the context of functional languages, however. Many of the issues that we must address before we can achieve architectural adaptability for imperative programs do not arise in functional programs, including the expression of potential parallelism, the effect of exploiting parallelism on program semantics, and the relationship between explicit synchronization and parallelism.

Although pure Lisp is functional, most Lisp-based programming languages are imperative. Like ParAlfl, an imperative Lisp can exploit parallelism in function evaluation

[1] This idea is also effective in structuring parallelizing compilers [Quiroz, 1991].

by selecting either lazy or eager (and potentially parallel) evaluation. For example, Multilisp [Halstead, 1985] provides the function `pcall` for parallel argument evaluation, and `future` for parallel expression evaluation. Qlisp [Goldman *et al.*, 1990] is similar, but provides more facilities for the conditional exploitation of parallelism. Unlike ParAlfl, Multilisp is an imperative language with assignment. Since parallel execution may affect the order of assignments, the use of `pcall` and `future` to introduce parallelism can affect the semantics of the program. In particular, a programmer can use `future` only when certain that it will not produce a race condition. Halstead advocates a combination of data abstraction with explicit synchronization and a functional programming style to minimize the extent to which side-effects and parallelism conflict.

To the extent that only the side-effect-free subset of Multilisp is used, both `pcall` and `future` can be thought of as annotations that select a parallel implementation without affecting the semantics of the program. Like ParAlfl, a side-effect-free Multilisp program can adapt easily to a new architecture with the addition or deletion of `pcall` and `future`. However, Multilisp was not designed to be used in such a limited fashion. A Multilisp program that uses side-effects to any significant degree cannot adapt easily to a new architecture, since exploiting alternative parallelism in the program requires that the programmer understand the relationship between side-effects and the intended use of `pcall` or `future`.

2.2.2 Data Parallelism

Data parallel languages provide high-level data structures and data operations that allow programmers to operate on large amounts of data in an SIMD fashion. The compilers for these languages generate parallel or sequential code, as appropriate for the target machine. Fortran 8x [Albert *et al.*, 1988] and APL [Budd, 1984] provide operators that act over entire arrays, which could have parallel implementations. The Seymor language [Miller and Stout, 1989] provides prefix, broadcast, sort, and divide-and-conquer operations, which also have parallel implementations. These languages achieve architectural independence for one class of machine (*i.e.* vector or SIMD) by providing a set of parallel operations that have efficient implementations on that class of machine.

The Paralation model [Sabot, 1988] and the Connection Machine Lisp [Steele and Hillis, 1986] support data parallelism through high-level control operations such as iteration and reduction on parallel data structures. These operations represent a limited use of control abstraction, demonstrating that it can be used to define data parallelism. Such operations are not a general solution to the problem of specifying parallelism however, since parallelism is defined solely in terms of a particular data structure.

2.2.3 Fixed Control Constructs for Exploited Parallelism

Explicitly parallel languages typically provide a limited set of parallel control constructs, such as `fork`, `cobegin`, or parallel `for` loops, which programmers use to represent and exploit parallelism simultaneously. If the degree of parallelism specified using these constructs is not appropriate for a given architecture, the resulting program is not efficient. In general, the correspondence between the parallelism described in the program and the parallelism exploited at run time is too restrictive in early explicitly parallel languages; selecting an alternative parallelization often requires almost completely rewriting programs.

2.2.4 Fixed Control Constructs for Potential Parallelism

Fortran $8x$ loosens the correspondence between potential and exploited parallelism with the `do across` construct, which has both sequential and parallel implementations. Programmers use `do across` to specify potential parallelism, and the compiler can choose either a sequential or parallel implementation as appropriate. Compilers on different architectures may make different choices, thus providing a limited degree of architectural independence. These choices are usually predefined by the compiler implementor; the programmer has no mechanism to extend the set of choices.

The Par language [Coffin and Andrews, 1989; Coffin, 1989] (based on SR [Andrews *et al.*, 1988]) extends the concept of multiple implementations for a construct to user-defined implementations. Par's primary parallel control construct is the `co` statement, which is a combination of `cobegin` and parallel `for` loops. The programmer may define several implementations of `co`, called *schedulers*, which map iterations to processors and define the order in which iterations execute. Using annotations, a programmer can choose among alternative schedulers for `co`, and thereby tune a program to the architecture at hand.

Any single control construct may not easily express all the parallelism in an algorithm, however. Languages that depend on a fixed set of control constructs for parallelism limit their ability to express certain algorithms easily. When the given constructs do not easily express the parallelism in an algorithm, the programmer must either accept a loss of parallelism, or use the available constructs to express excessive parallelism, and then remove the excess using explicit synchronization. The former approach limits the potential parallelism that can be exploited, while the latter approach results in programs that are difficult to adapt to different architectures. In the particular case of Par, programmers must express all parallelism with `co`. It is tempting to create new parallel control constructs by embedding synchronization within an implementation of `co`. This approach changes the semantics of `co` however, and leaves a program sensitive to the selection of implementations, violating the Par assumption that annotations do not change the meaning of the program.

2.2.5 User-Defined Control Constructs

The problem with any approach to architectural adaptability based solely on selecting alternative implementations of a small fixed set of control constructs is that our ability to describe potential parallelism is limited to compositions of the parallelism provided by the constructs. Chameleon [Harrison (Alverson) and Notkin, 1990; Alverson, 1990; Alverson and Notkin, 1991] represents a first step towards user-defined control constructs. Chameleon is a set of C++ [Stroustrup, 1986] classes designed to aid in the porting of parallel programs among shared-memory multiprocessors. It provides schedulers for tasks, which are a limited form of control abstraction. Each task is a procedure representing the smallest unit of work that may execute in parallel. Schedulers call tasks via procedure pointers. Because Chameleon uses dynamic binding in the implementation of schedulers, a compiler cannot implement tasks in-line. In addition, programmers must explicitly package the environment of the task and pass it to the scheduler. The resulting overhead is acceptable only when tasks are used to specify the medium and large grained parallelism appropriate to shared-memory multiprocessors.

2.3 Distributing Data and Processing

For machines that distribute storage among processors so that access to another processor's storage is either slower or not possible, programs execute faster when data is co-located with the processor that uses the data. In the absence of sharing, this co-location would not be a problem. However, programs do share data, to varying degrees. Techniques for maintaining the sharing of data while still co-locating it with processors include process movement, data movement, and data replication. Languages that expect to execute in environments with distributed storage usually provide mechanisms to control the distribution of data and/or processing.

2.3.1 Static Distribution

Distributed Processes [Brinch Hansen, 1978] provided a static mapping of processes to processors. Since, all data was local to a process, the mapping of data was implicit in the mapping of processes. The static mapping of processes meant that any change in machine configuration required re-mapping the processes, and in the worst case, rewriting the program to include more processes so that it could take advantage of additional processors.

2.3.2 Embedding a Virtual Machine

To avoid the problem of adjusting programs to every change in machine configuration, several early distributed and parallel languages presented a means to define a program-dependent virtual machine, and then program in terms of this virtual machine. This virtual machine usually had a finer grain than the real machines. The programmer than separately specifies the mapping from the virtual machine onto the physical machine. Several virtual processors may reside on a single physical processor, but no virtual processor may reside on more than one physical processor. Languages taking this approach include *MOD [Cook, 1980], NIL [Strom and Yemini, 1983], Hermes [Strom *et al.*, 1991] and Poker [Snyder, 1984]. Poker assumes virtual processors will be small, whereas *MOD and NIL assume they will be at least moderately sized.

2.3.3 Dynamic Distribution

Later languages provide mechanisms for dynamically determining the mapping of processing and data to processors. This enables programs to adapt to different numbers of processors by computing an appropriate mapping. ParAlfl provides *mapped expressions* that assign the computation of a function to a specific processor. Since ParAlfl is functional, data is implicitly mapped with expressions.

Emerald [Jul *et al.*, 1988] also provides means for computing an appropriate distribution. Unlike the functional ParAlfl, Emerald was intended for an evolving environment. To adapt to a changing environment, data and processing will need to move from one processor to another. Emerald provides mechanisms for explicitly moving data and processing.

Object Movement in Emerald

Emerald is an object-based language. When invoking an operation on an object, Emerald will normally send the invocation to the processor containing the object for execution by that processor. However, Emerald also provides mechanisms to determine the location of objects, move an object to a node, fix an object on a given node, unfix an object, refix an object (an atomic unfix, move and fix). Since an Emerald object may contain a process in addition to data, object migration subsumes both data and process migration.

Emerald adopts a reference model of variables in which objects consist primarily of a few references to other objects (see section 3.3). Emerald proves a uniform semantics for all objects, local, co-located, and remote. Moving an object may mean that others will also need to move soon. Because of the fine-grained nature of Emerald objects, the explicit management of every object in a program would become an unacceptable programming burden. To solve this problem, the Emerald compiler attempts to find objects that are only referenced from within a second object, and therefore should move with the second object [Hutchinson, 1987]. Moving several objects at a time is much cheaper than moving them individually. In cases where the compiler cannot discover restricted referencing, Emerald provides the notion of *attached* objects. Attaching an object to another means that when the second object moves, the first will move with it. This enables programmers to build collections of related objects that maintain their relative locality dynamically.

Emerald also supplies two hints for moving arguments to operation invocations, *call-by-move* and *call-by-visit*. Call-by-move indicates that the argument object should move to the node containing the called object. Call-by-visit indicates that the argument object should move to the node containing the called object for the duration of the call and then move back. The caller indicates the appropriate transmission method. Call-by-move and call-by-visit hints at the point of call are 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.

In Emerald, objects that will not change (*immutable objects*), such as code or static tables, can not only be moved, they can be replicated. Replication enables information that is shared, but not updated, to be accessed efficiently from any node. Emerald only requires that the *abstract value* of the object not change; the representation of the value is free to change over time.

2.3.4 Distribution via Data Abstraction

Data abstraction is a useful tool in parallel programming [Murtagh, 1983] as well as in sequential programming. Recent languages rely on data abstraction to hide the distribution of data and processing. With data abstraction, the implementation of a data structure can change as the distribution needs change. For example, an array abstraction has several possible implementations. These include a contiguous representation on a single node, a distributed representation where elements are divided among nodes, a distributed representation where elements are associated with nodes in a modular fashion, a fully replicated representation where each node contains a copy of the entire array, and a partially replicated representation where each node duplicates only a portion of the array.

Par

In Par [Coffin and Andrews, 1989], programmers define data abstractions for data structures that may be distributed. Later, programmers annotate abstractions to select an implementation appropriate to the current architecture. For example, programmers use an array abstraction, but later select a contiguous or distributed implementation of the array. An implementation also exports a set of mapping operations. Programmers invoke these mapping operations when the pattern of access to a data abstraction changes. Mapping operations allow the representation of the abstraction to change to meet new access patterns. For example, when the program changes from read/write access to an exclusive portion of the array to read-only access to most of the array, programmers would insert an call to a mapping operation that changes the representation of the array from distributed among processors to replicated across processors.

Not only is data abstraction useful for the distribution of data structures, it is also useful for the distribution of *single values* [Coffin, 1990]. For example, many relaxation algorithms have the form:

```
repeat
    changed := false
    for each element
        compute new state
        if new state ≠ old state
            changed := true
while changed
```

The straightforward parallel implementation of this algorithm distributes the elements and state computations among processors. The problem with this implementation is that each state computation will access the same shared "changed" variable. The resulting contention will cause poor performance for large numbers of processors. The solution is to define a distributed boolean type. The distributed boolean type could implement the assignment by updating a local copy of the boolean, and then when the value is requested via the read, examine each processor's copy and return the latest value. Of course, we could also provide the standard single-valued implementation in place of a distributed implementation. In either case, the *use* of the boolean variable does not change.

In Par, programmers distribute data by building data abstractions based on distributed arrays. Programmers distribute processing with *schedulers*, which are distributed throughout the machine. The programmer is responsible for maintaining, via annotations, the appropriate correspondence between data distribution and process distribution.

Chameleon

Chameleon [Harrison (Alverson) and Notkin, 1990; Alverson, 1990; Alverson and Notkin, 1991] is a C++ library providing several data abstractions and their corresponding schedulers. For example, the array representations include contiguous, replicated, and distributed. The Chameleon library selects the appropriate implementation at runtime.

Chameleon programmers represent work in terms of *chores*. A chore consists of a work procedure and a set of characteristics describing the procedure. These characteristics include unit cost, for determining granularity; and parameter access type (read-only or read-write), for managing software-caching. The scheduler works in terms of *tasks*, which are a composite of chores and a preferred schedule of execution. A *partitioner* procedure defines the schedule and calls the work procedure.

Chameleon improves on Par by more tightly integrating scheduling with the data it accesses, but at the cost of considerable programmer effort in describing the chores and tasks. Part of Chameleon's descriptive cost arises because C++ lacks mechanisms to represent control abstractions. Because programmers must describe loop bodies as separate procedures, environments and parameters must be explicitly packaged, which inhibits the wide-spread use of Chameleon data abstractions. This, in turn, means that programmers will describe less potential parallelism within their programs, and therefore limit the class of architectures for which the programs are effective.

2.4 Choosing Communication

Communication costs vary significantly across architectures, and the degree of parallelism and the distribution of data among processors determines the need for communication. When programmers must explicitly specify communication, there are two possible inefficiencies. First, programmers may specify communication at such a fine grain that the communication overhead results in poor performance. Second, programmers may underspecify communication, so that too many processors wait for work.

Most programming languages provide no help to the programmer in specifying an appropriate balance in communication. Instead, they rely on the programmer to program at a granularity appropriate to the class of target architectures. If we wish to write programs that adapt to a wide range of architectures, we must provide a means to easily insert and remove inter-processor communication.

2.4.1 Virtual Communication

One technique for adapting communication to the architecture is to specify virtual communication and then package virtual communication into physical communication. For example, Poker expects programmers to use fine-grained communication via small messages between small virtual processors. Poker then applies compiler techniques to combine several small messages into a single larger message. Combining messages reduces the number of messages, which makes message overhead commensurate with the architecture.

2.4.2 Communication via Invocation

Emerald also expects programmers to communicate at a fine grain, but via object invocation rather than explicit messages. Object invocation traditionally uses procedure implementations and Emerald takes advantage of this implementation when object reside on the same processor. When performing an invocation on a remote object, a procedure implementation will not work, and Emerald implements invocation via messages. This is an instance of the general technique known as *remote procedure call*. Emerald provided a significant improvement over earlier remote procedure calls by making the semantics of remote procedure calls identical to local procedure calls. Programmers use the same communication mechanism, the operation invocation (procedure call), at all levels in the program. The Emerald implementation introduces communication among processors only when necessary.

Programs communicate between referencing environments, not processes. The literature often thinks of communication as between processes, but this is primarily an accident of early programming languages providing a referencing environment identical to the process. When we associate communication with object invocation, we move closer to the idea of communication between referencing environments.

2.5 Summary

Early parallel programming languages provided mechanisms that explicitly controlled the parallelism, distribution and communication within programs. Programs written in these languages usually executed efficiently only on the architecture for which they were originally written. Later languages provided constructs that described potential parallelism, rather than exploited parallelism. Then, late in program development, programmers could change the exploitation of the parallelism provided by those constructs. These languages were the first step in achieving architectural adaptability. Recent languages focused on the use of data abstraction in parallel programming, particularly with respect to distribution. Data abstraction enables programmers to adapt programs to a greater range of architectures than a fixed set of parallel constructs.

3 Matroshka Model and Rationale

Everything should be made as simple as possible, but no simpler.
— Albert Einstein

This chapter describes the Matroshka (Матрёшка) model of parallel programming and its rationale. The model has three goals:

Transparency: The model should not hide significant architectural capabilities.

Uniformity: The model should provide uniform mechanisms for defining program elements, reguardless of their eventual implementation.

Efficiency: The model's mechanisms should have efficient implementations.

The Matroshka model uses a few, carefully chosen, general mechanisms for uniformly defining sequential and parallel abstractions to achieve a rich programming environment. In describing SR, Andrews *et al.* [1986] state "Thus a distributed programming language necessarily contains more mechanisms than a sequential programming language." The Matroshka model contradicts this statement; the generality of its mechanisms results in fewer mechanisms than commonly found in sequential languages.

The Matroska model supports data abstraction with *objects*, with *synchronous operation invocation* and *concurrent operation execution*. That is, operations on objects execute synchronously with respect to their invokers, and execute concurrently with respect to other operations on the object. Operations may *reply early*, which enables an operation to continue concurrently with its invoker. Unlike most object-based programming languages, Matroshka uses a *copy model* of variables and parameters. Finally, Matroshka supports control abstraction via *first-class closures*.

Matroshka is not a programming language. It leaves many issues in language design unspecified in the model, such as syntax, inheritance, static or dynamic typing, *etc.* So, the model has a wide range of possible instantiations as a programming language. To make the presentation concrete, this dissertation defines the Natasha (Натáша) programming language. Natasha is a statically typed prototype language, intended only to illustrate the concepts in this dissertation. As a prototype language, it does not provide many features that are desirable in a production quality language, such as inheritance. Appendix A contains the Natasha language definition.

3.1 Uniform Data Abstraction

We may wish to use different representations for data depending on the architecture. To change representations easily, we must abstract the data. Mechanisms for data abstraction must provide for treating a collection of variables as a single item, and provide a means to define operations on the collection. Mechanisms for data abstraction include Modula-2's modules [Wirth, 1982], Ada's packages [U. S. DoD, 1983], and CLU's clusters [Liskov *et al.*, 1977].

3.1.1 Single Data Abstraction Mechanism

Many parallel programming systems have two mechanisms for data encapsulation, one for global and parallel abstractions and one for local and sequential abstractions. This dual mechanism splits the programming environment into two qualitatively different models of interaction, introducing an artificial granularity in the programmer's specification of parallelism. A better approach is to provide a single encapsulation mechanism that applies *uniformly* to both parallel and sequential abstractions.

A language provides uniform data abstraction when all program elements, from primitive language elements to large user abstractions, use the same data abstraction mechanism, regardless of the intended concurrency within the elements. If a data abstraction mechanism is to apply uniformly to all elements, it must *only* provide abstraction. Additional semantics lead to multiple mechanisms because program elements may need to differ on the other semantics.

The presence of only one abstraction mechanism does not imply only one implementation for that mechanism. If an abstraction mechanism is to apply to all program elements, it must have implementations suitable to all element sizes and uses. Programmers may then choose the appropriate implementation late in program development.

3.1.2 Data Abstraction via Objects

The Matroshka model provides data abstraction with the *object*. Every data item within a program is an object. Each object has a state represented by the states of its component variables. Programmers may invoke *operations* that manipulate the internal state of an object by invoking operations on component objects. The invocation of an operation is the sole mechanism for changing the state of the object. Thus, operation invocation is the fundamental communication mechanism in the Matroshka model. Matroshka objects are only an encapsulation mechanism.

The object model provides natural abstraction of data, from simple integers to databases. Identical syntax and semantics for operations on such disparate objects can still have very different implementations. For example, an integer will likely have a single machine word representation. A database will likely have its representation split between volatile and non-volatile storage.

Objects also provide natural units for distribution. Distributed systems and non-uniform-memory-access multiprocessors have substantial performance differences depending on whether communication occurs within a processor or between different processors. Objects provide a natural destination for communication, and hence aid the programming system in reducing communication costs.

Objects tend to reduce the referencing environment of any one piece of code. For example, without objects programmers tend to share a common pool of variables and use them in an unstructured way. Programmers using objects tend to collect variables into objects and limit variable access to a small set of operations. This reduced referencing environment originally served to reduce programming errors. In parallel programming, a smaller referencing environment means that there are fewer potential objects with which another object may communicate. Fewer destinations for communication means that programmers and programming systems may more easily analyze the program for possible race conditions and parallel optimizations.

In summary, objects provide a single set of syntax and semantics that enables a wide variety of implementations for different objects, that provides a destination for communication, and that reduces the referencing environment.

3.1.3 Objects in Natasha

Natasha programmers define object types in terms of the set of variables the object contains, and the methods that implement operations on the object. Object type definitions have the form:

```
type-name:  object
{ var-name: ...
  var-name: ...
  method operation-name parameter: type { ...};
  method operation-name parameter: type { ...};
};
```

See section A.7 for more details.

3.1.4 Generic and Polymorphic Types

A programming language that relies heavily on abstraction should provide mechanisms for generic and polymorphic definitions. This is particularly important when defining types that manipulate collections of objects. Programmers need to define operations on the collection type that can manipulate operations on the element types. In statically typed languages, making this capability available involves generating many operations on the collection type based on the operations on the element type. For example, in defining an array of integers, we also wish to define an operation on the array that returns the *reduction* of the elements over any appropriate integer operation. This meta-operation is exactly that provided by APL [Gilman and Rose, 1976]. While important to parallel programming and architectural adaptability, generic and polymorphic type mechanisms are generally well understood and not crucial to this dissertation. As a result, this dissertation will not discuss them but will assume that production-quality languages based on the Matroshka model will provide them.

Natasha provides limited support for generic types with composite names. The compiler recognizes the generic names for certain predefined language types. The user is responsible for duplicating program text for their own generic types. We use the C preprocessor for this task. It is clumsy, but serves for the prototype.

3.1.5 Nested Object Type Definitions

For implementation expedience, Natasha does not support nested object type definition. This decision was a mistake. It prevents an object definition from obtaining access to the variables in any objects that may contain it. Several example programs have an unnatural structure because they must pass information through global variables rather than though variables in a known parent object. Production-quality programming languages based on the Matroshka model should support nested object types.

3.2 Synchronous Operation Invocation

Not only must the abstraction mechanism apply uniformly to parallel and sequential objects, the means for communicating with them must also apply uniformly. The originating object must be able to communicate without knowing how the receiving object will handle it, and *vice-versa*. Several programming systems, *e.g.* *MOD [Cook, 1980], allow both synchronous communication (procedures) and asynchronous communication (messages), but usually require both sender and receiver to agree on the form, which inhibits changing the form late in program development. These systems have non-uniform communication. The cost of providing complete flexibility in communication for an object in these systems is combinatorial in the number of communication mechanisms. In contrast, SR [Andrews *et al.*, 1988] lets programmers mix-and-match synchronous and asynchronous communication.

3.2.1 Implicit Waiting

Most computations communicate with the intent of receiving a reply. However, asynchronous message-based systems often do not recognize the concept of a reply. Programmers must explicitly wait on a message containing the reply. However, waiting on the arrival of a asynchronous message is itself a synchronous operation, so even systems based on asynchronous invocation usually supply synchronous primitive operations. Thus, message-based systems tend to be non-uniform. The complexity and non-uniformity of most message-based languages has lead to a greater concentration on synchronous, procedure-based communication in which waiting is implicit. The evolution of the asynchronous, message-based ECLU [Liskov, 1979] into the synchronous, atomic-transaction-based Argus [Liskov and Scheifler, 1983] is an example of this trend.

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 invoker. Instead, the invoker 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 used heavily in most programs, this approach requires language support for implicitly passing continuations. The continuation model provides more expressive power than is necessary for the purposes of this dissertation.

Given these considerations, the Matroshka model supports uniform communication with *synchronous operation invocation* on objects. The invoker of an operation implicitly waits on receipt of the reply value, which may be used as an argument to another invocation. This is the sole communication mechanism. The model does not provide asynchronous communication directly because it has a straightforward implementation with other concepts in the model.

As an example of synchronous operation invocation, consider the following Natasha code fragment.

```
"Hello ".print![];
"World!".print![];
```

Natasha will wait for the first invocation to reply before starting the second invocation. We can be sure that this fragment will print the string "`Hello World!`" rather than "`HeWlolorl d!`" or some other equally incomprehensible variant.

3.2.2 Invocation as Communication

The Matroshka model enables efficient use of multiple architectures by associating communication with abstraction. Since programmers will use layers of abstractions, the implementation can communicate across processor boundaries at any layer of abstraction. Because the binding of processor-to-processor communication to object invocation can occur late in program development, the model allows the programmer to tune a program to an architecture without affecting the integrity of the algorithm.

For example, consider executing a program on both a distributed, message-based system and on a shared-memory multiprocessor. On the distributed system, object invocation at higher levels of the program's abstractions would be implemented by messages across the communication network, and invocation at lower levels would be implemented by procedure calls. Expensive message traffic is reduced by using messages only at the highest levels of program. On the shared-memory multiprocessor, object invocation at all levels of abstraction would be implemented by procedure calls, except at the lowest level were the processors communicate through reads and writes on individual machine words. Processor communication occurs through fast shared memory and avoids the expense of constructing messages.

Note that not all programs that execute efficiently on a shared-memory multiprocessor will execute efficiently on a distributed system. In particular, if a program has a many small, communication intensive objects with no intervening layers of abstraction, the communication graph may be too fine-grained for efficient implementation on a distributed system.

3.2.3 Implicit Reply Addressing

When using asynchronous message-based languages, programmers that wish to wait on a reply must often pass self-references so that the receiver knows where to send the reply.

The original sender must then filter incoming messages in search of the reply value. To support waiting on replies, message-based programming languages sometimes provide complex filtering mechanisms. For example, in PLITS [Feldman, 1979] programmers must allocate a transaction key for a conversation and then explicitly wait on the reply under that transaction key.

If the programmer must send results as explicitly addressed communication, the burden on the programmer is high, both for the invoker and the invokee. The destination for the result of an operation should be implicit. In Matroshka, reply destinations are implicit. For example, Natasha methods return values to their invoker with the `reply` statement:

`reply` *expression*;

There is no naming of the invoker in the reply statement.

3.2.4 Ports

One advantage of message passing systems is that they may be connected into rich networks where the intermediaries are not necessarily known in advance. Passing references to neighboring objects allows effective construction of networks. However, if the operations must be named directly, each sender in a network must know the name of the operation of the recipient, which requires the sender and receiver to agree on an operation name *a priori*. *A priori* agreement on names implies that programmers may have to place in the communication path many intermediary objects whose sole purpose is to translate operation names. See [Scott, 1987] for further discussion. Similar problems arise with static typing of message recipients. Network construction under these circumstances will be an *ad hoc* task.

The Matroshka model provides communication independence with *ports*. A port is a first-class language entity binding an operation to an object reference. Applying an argument object to a port invokes the corresponding operation. The user of a port may need to know the type of the argument and the type of the result, but does not need to know the operation name or the type of the port's object.

In Natasha, ports are typed by the type of their arguments and results. For example, the type of port accepting an `integer` and returning a `boolean` is `port‘integer boolean’`. We specify a port with the ‘`.`’ primitive. Given a variable `foo`, with the operation `bar`, the expression `foo.bar` returns a port with typed by `bar`'s parameter and result types. The type of `foo` is not part of the port's type. We then invoke the operation corresponding to the port with the ‘`!`’ primitive. For example, `foo.bar!3` invokes the `bar` operation on the object `foo` with the parameter `3`. The ‘`.`’ and ‘`!`’ primitives have equal precedence and bind left-to-right. The result of the expression is the reply value of the operation. Natasha may evaluate the components of the expression in any order, but will evaluate them sequentially.[1]

Ports enable programmers to connect objects into rich networks, where the exact type of objects is not known in advance, even in the context of statically typed languages.

[1]We chose this approach for implementation expedience and the lack of a strong reason to do otherwise.

Ports also enable programmers to build libraries of control abstractions with fewer type dependences.

3.2.5 Single Argument and Result

Control abstractions will often need to delay the invocation of an operation, or apply an operation to many objects. In addition, control abstractions may need to combine the results of several computations. To describe general-purpose abstractions to that manipulate other operations, an intermediary must be able to handle the arguments and results of operations as single units. Message based systems naturally provide this capability by referring to messages as a whole. RPC based systems generally do not provide a mechanism that enables the programmer to refer to the set of procedure arguments. (Suitable changes would enable RPC systems to refer to the set of arguments.)

The Matroshka model simplifies programming of control abstractions by allowing exactly one parameter and one result for each operation. This means that control abstractions need only handle one argument and result combination. Passing a record as the argument achieves the effect of multiple parameters. If a language based on the Matroshka model provides *record constructors* (*e.g.* Mesa [Xerox, 1984]), this approach can be as notationally concise as a list of parameters. Those operations that do not need an argument, or have no useful result, accept or return an empty record. For example, in the Natasha statement

```
"Hello World!".print![];
```

the expression '`[]`' constructs an empty record. (We name the type of empty records with the identifier `empty`.) The statement

```
range.new![ from: 1; to: 100; ];
```

constructs a record consisting of two components and passes it to the `new` operation on the `range` object.

3.3 Copy Model of Variables and Parameters

Imperative languages present two models of variables and parameters. In the conventional model (*e.g.* Fortran and the Algol family of languages), variables contain values. We call this the *copy* model. Two distinct variables cannot refer to the same storage. Figure 3.1 illustrates the relationship between objects and variables in the copy model. Note that the two `bounds` variables contain different objects. Changing the `lower` variable in one object will not affect the value in the other. In the *reference* model, variables refer to objects (*e.g.* CLU [Liskov *et al.*, 1977] and Smalltalk [Goldberg and Robson, 1983]). Figure 3.2 illustrates the relationship between objects and variables in the reference model. Two distinct variables may refer to the same object, or storage. That is, each variable is a pointer to another object. For example, the two `bounds` variables refer to the same object, so changes to the `lower` variable is visible from the objects referred to by both `proc0` and `proc1`. The reference model introduces an "infinite regression" in its pattern of objects being composed of references to other objects. When does one

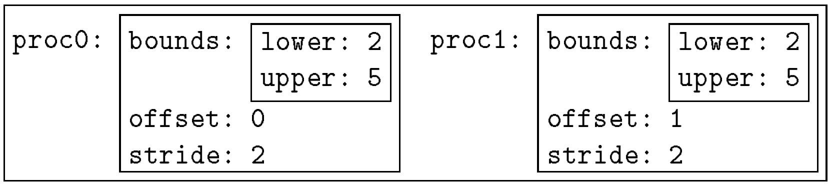

Figure 3.1: Copy Model of Variables

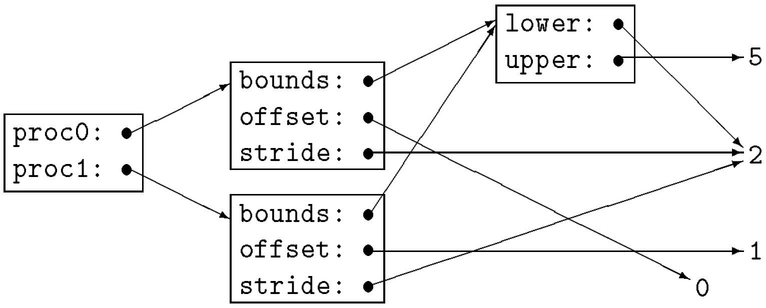

Figure 3.2: Reference Model of Variables

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.[2] The reference model can directly represent non-hierarchical, or cyclic, data structures. (On the other hand, the copy model requires the introduction of a separate pointer variable to represent non-hierarchical structures.)

Most explicitly parallel imperative languages use the reference model for concurrent abstractions, but use the copy model for parameters and local variables. The desire for a uniform encapsulation mechanism implies that a parallel language must choose one model and stick with it. The one model must subsume variables for parallel abstractions, parameters, and variables for sequential abstractions (usually local to a parallel abstraction). Table 3.1 lists combinations provided by some languages. For example, SR uses a reference model for its *resources* (which are distributable objects), but uses the copy model for local variables and parameters. Current systems with uniform encapsulation (*e.g.* Actors [Agha, 1986b] and Emerald [Black *et al.*, 1986a]) use the reference model. In contrast, Matroshka uses the copy model.[3]

[2] 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. The change in representation enables programmers to adapt to changes in access patterns.

[3] The nesting of matroshka dolls is the inspiration for the name of the Matroshka model.

	Actors	Ada	Argus	CSP	Emerald	Matroshka	SR
parallel abstractions	refer	refer	refer	refer	refer	copy	refer
sequential/local	refer	copy	refer	copy	refer	copy	copy
parameters	refer	copy	copy	copy	refer	copy	copy

Table 3.1: Combinations of Variable Models

The reference model has some attractive properties, such as fewer naming mechanisms more concise sharing, and a more straightforward implementation of polymorphic types. However, the copy model has several advantages over the reference model in the context of parallel and distributed programming. These advantages include:

- controlled aliasing, which aids the exploitation of parallelism.
- reduced object contention, because each operation receives a separate copy of its arguments.
- reduced interprocessor communication, because arguments to an operation are copied to the processor executing the operation rather than remaining remote.
- more parameter implementations (*e.g.* copy on write), which provides more opportunities for optimizing parameter passing.
- more efficient storage management, because lifetime can be associated with scope.

Real programs must deal with both values and references, so the choice of the variable model in a language represents a bias, and not an absolute choice. The bias of the copy model is more appropriate for parallel programming.

3.3.1 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 (until proven otherwise). A potential shared reference requires either that the programmer and system not attempt to operate on the two variables concurrently, or that the object explicitly control 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. This inhibits the detection and exploitation of parallelism by both programmers and compilers.

In contrast, the copy model ensures that each variable refers to a different object. Thus the programmer and the compiler are free to operate on two different variables concurrently. The copy model allows compilers to parallelize sequential code effectively.[4]

[4]Though we do not depend on sophisticated compilers for architectural adaptability, we do wish to exclude their use.

The current research effort in parallelizing sequential programs [Polychronopoulos, 1988; Allen *et al.*, 1987; Sarkar, 1990; Wolfe, 1989], may aid in further parallelizing parallel programs if the parallel program adopts a copy model of variables, though there is considerable research remaining [Sarkar and Hennessy, 1986]. This approach is likely to complement the coarser-grained parallelism that programmers provide.

3.3.2 Reduced Object Contention

In multiprocessor systems, mutable (*i.e.* value changing) objects are obvious sources of possible contention. However, many objects may be accessed in phases, where for long periods of time the program is interested in an object's value and not in tracking its changes in state. For example, in Gaussian elimination each row changes state until it becomes the pivot row, at which point further reductions need only the value of the row. To reduce contention in the reference model, the programmer has two options: to make the rows immutable or to copy the pivot row explicitly.

When 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 copy 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. The copy approach represents the copy model, but with higher run-time and programmer costs. Implicit copy under the copy model often requires less run-time support, and hence costs less.

3.3.3 Reduced Interprocessor Communication

The reference model implies heavy communication between machines as operations traverse back and forth across machine boundaries to reach the objects referenced by the parameters. This potential inefficiency on distributed systems is such that Argus, which has an internal reference model based on CLU, uses an external copy model. On the other hand, the Emerald system [Black *et al.*, 1986b] uses the reference model over a local area network. Emerald mitigates the cost of remote arguments by moving parameter objects, and the objects they refer to, from the calling machine to the called machine. This approach may limit performance if there is contention for the argument object. In contrast, the copy model moves all necessary information at the point of call. Communication occurs exactly twice, once sending the parameters, and once sending the results.

3.3.4 More Parameter Implementations

Parameters passed "by value" may be implemented by copying the argument or by passing a pointer to the argument (assuming the operation does not change the argument.) In contrast, parameters passed "by reference" as other than pointers require coherency protocols. Thus "by value" parameters are more amenable to optimization than "by reference" parameters.

This implementation flexibility is important because multiprocessors have a high degree of sharing and complicated mechanisms for sharing, and the point at which it is more efficient to pass arguments by pointer or by copy changes more often than in distributed systems. For instance, the BBN Butterfly can implement copy parameters several ways, 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 accessed infrequently, passing the objects by pointer is more efficient. For large objects that are accessed frequently, passing the objects by copy is more efficient (or by moving them, as in Emerald).

Compilers may provide a copy implementation of a "by reference" parameter when the argument is immutable, or when the programmer makes an explicit copy at the point of call. These constraints are hard to meet, so the reference model effectively discourages "by value" parameters. Hence, the reference model provides less flexibility in implementing parameter passing in comparison to the copy model. These limitations become important when porting programs among different multiprocessors.

3.3.5 Efficient Storage Management

Under the reference model, references to an object may spread freely. Because the existence of reference to an object usually implies the existence of the object, it is generally not possible to determine the lifetime of an object statically. The indeterminate lifetime of objects implies dynamic heap allocation and system-wide garbage collection. To collect garbage, 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 large or widely distributed systems, the number of possible locations for a reference is too large to be effectively searched. This issue is important because institutions invest in parallel programming for speed. Institutions are often willing to purchase performance with engineering effort. The reference model inhibits performance. On the other hand, the variables under the copy model contain objects. This enables compilers to determine of the lifetime of every object statically. Because the lifetime of the variable containing it is known, more efficient storage management is possible.

3.3.6 The Object as a Reference Parameter

One criticism of a strict copy model of parameters in algorithmic languages, such as C [Kernighan and Ritchie, 1978], is that they do not allow changing the argument (as distinct from the parameter). Programmers must pass explicit references to change data. This criticism is less valid in object-based systems because the object being operated on is implicitly passed by reference. For example, the implicit reference to an 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.

3.3.7 Explicit Reference Variables

The Matroshka model adopts the copy model of variables and parameters. Variables contain objects, they do not refer to objects. This means that operation arguments are objects, not object references. Because a strict copy model of computation can only represent hierarchical structures and not graph structures, the Matroshka model provides an explicit *object reference* capability, *i.e.* pointers. Reference variables must be dereferenced explicitly. The type of a reference depends on the type of the referent. The Matroshka model is biased towards copies rather than references.

3.3.8 Variables, Parameters, and References in Natasha

Under the Matroshka model, variables serve to capture objects for later reference. While variables contain objects, a variable name refers to an object. The variable itself contains an object, but the name refers to the object. More specifically, a variable name is a literal value for a reference to the corresponding object. Because of the copy model of variables, two different simple variables names must refer to two different objects. In Natasha, all variable declarations have an initializing expression, which implicitly defines the type of the variable.

Variable declarations have the form:

variable-name : expression ;

For example,

```
letters: 26;
```

defines the variable '`letters`' with the initial value '`26`', which is an integer, so the variable's type is integer.

Parameters have no initializing value, so they are declared with their type instead of their initial value. For example, a `boolean` parameter would have the following definition:

```
invert: boolean
```

where `boolean` is the variable name of the type object.

Natasha makes a distinction between *l-values* and *r-values*. Natasha expects references to appear as the left operand of the '`.`' primitive, and expects values to appear as the operand of the '`!`' primitive. When a value appears as the left operand of '`.`', Natasha constructs a reference to the object. When a variable name appears on the right of the '`!`' primitive, Natasha invokes the copy operation on the corresponding variable to obtain it's value. For example, the expression `foo.op!bar` is equivalent to the expression `foo.op!(bar.copy![])`. (The postfix '`@`' operator is equivalent to `.copy![]`).

For the most part, the l-value/r-value interpretation is intuitive. The exception is array indexing. The result of array indexing is the reference we want. We do not want Natasha to construct a reference to the reference. We indicate this with the '`,`' primitive, which suppresses reference creation. For example, the statement

```
A#i,.print![];
```

prints the `i`'th element of `A`. It is equivalent to the statements

```
A#(i@),.print![];
A.select!(i.copy![]),.print![];
```

Likewise, when retrieving an array element, the select operation returns a reference, which we want to dereference to obtain the value. We do so explicitly with the `copy` operation. The following statements achieve the desired effect, and are semantically equivalent.

```
b := (A#i,@);
b := (A#(i@),@);
b.assign!(A.select!(i.copy![]),.copy![]);
```

Note that expressions bind left-to-right, so the parenthesis around the expression on the right-hand side of the assignment are necessary.

3.4 Concurrent Operation Execution

Architectural adaptability depends on having much potential parallelism available, and exploiting it as appropriate. So, a parallel programming model should not interfere with the programmer's ability to express parallelism.

3.4.1 No Implicit Synchronization

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] shows that the system must be written to permit as much parallelism as possible. Otherwise, the applications will serialize 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 forces serialization on system objects.

Any limitation on the potential concurrency within the encapsulation mechanism will magnify itself at each level of abstraction. Systems that place a heavy price on abstraction mechanisms encourage programmers to avoid them, resulting in highly unstructured, unmaintainable programs. To prevent unnecessary synchronization, the Matroshka model provides *concurrent operation execution* within a single object. Thus, while invocations are synchronous with respect to their invokers, they are asynchronous with respect to other invocations on the object. The model itself does not supply synchronization. If the programmer needs to synchronize invocations, the programmer must explicitly program such synchronization using language or implementation defined object types.

3.4.2 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 often introduces unnecessary synchronization costs. 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 happens with Ada [U. S. DoD, 1983]. This allows the system to compile in-line those operations 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.

Matroshka implicitly dispatches operations for execution as soon as they are invoked. Any synchronization with other threads is provided by the programmer using object-specific synchronization mechanisms.

3.4.3 Synchronization in Natasha

Matroshka presents no mechanism for synchronization other than that implicit in an operation invocation waiting for the reply. We assume that any languages based on our model will provide some primitive synchronization mechanism(s), like atomic memory accesses, test-and-set, or higher-level synchronization primitives. Natasha provides three synchronization primitives, counting semaphores, condition variables as in Mesa [Lampson and Redell, 1980], and concurrent-read-exclusive-write (crew) locks.

3.5 Uniform Control Abstraction

Much like data abstraction, which hides the implementation of an abstract data type from users of the type, control abstraction hides the exact sequencing of operations from the user of the control construct. In parallel programming, control abstraction includes the partial order of execution. Most current imperative parallel languages lack facilities for defining control abstractions. Hilfinger [1982] provides a short history of major abstraction mechanisms in programming languages, from procedure and variable abstraction in Fortran, through data structure abstraction in Algol68 and Pascal, to data type abstraction in Alphard, CLU, and Euclid. However, Hilfinger does not mention control abstraction. Significant mechanisms for control abstraction are present in both Lisp and Smalltalk. In these languages, control abstraction is present to enhance the expressiveness of the language. In parallel languages, control abstraction is even more important because the flexibility of control more directly affects performance.

When the semantics of a construct admit either a parallel or a sequential implementation, the user of the construct need not know which implementation is used during execution. The program will execute correctly whichever implementation is used. In general, a control construct defined using control abstraction may have several different implementations, each of which exploits different sources of parallelism. Programmers can choose appropriate exploitations of parallelism for a specific use of a construct on a given architecture by selecting among the implementations. The definition of a control construct represents potential parallelism; an implementation of the construct defines the exploited parallelism. Using annotations, we can easily select implementations without changing the program and thereby achieve architectural adaptability.

The appropriate parallelization of control is generally dependent on user data structures, For example, parallel programs may need to distribute the work on a list. Since data structures are user-defined, control constructs that operation on them must also be user-defined. For example, a programmer must be able to define a construct for parallel iteration over a list, in addition to primitive control structures. To define general-purpose control constructs, programmers must be able to reference code, as well as data, indirectly. Since the determination of sequential or parallel execution of these new control constructs is architecture-dependent, programmers must be able to change their implementations late in program development. This late binding implies that control constructs apply uniformly to parallel control as well as to sequential control. The issue of programmable control abstractions is not unique to parallel systems, it is common to programming in general.

The need for defining control abstractions is greater in parallel languages than in sequential languages, because otherwise each programmer must build *ad hoc* mechanisms for creating parallelism. *Ad hoc* creation of parallelism increases the cost of developing highly parallel programs and markedly increases the cost of changing the method of parallelism, which in turn inhibits the specification of extensive parallelism and especially data parallelism. For instance, without a mechanism for control abstraction, users cannot define general purpose control constructs, such as a parallel for-loop, without explicitly collecting references to the relevant portion of the environment.

With uniform control, it is feasible to provide libraries with many different implementations of control constructs. For instance, a library could provide both parallel and sequential implementations of a tree traversal construct. Programmers may then choose the appropriate implementation late in program development.

The Matroshka model provides three basic control mechanisms: expressions based on operation invocation, the sequential execution of expression statements, and a reference to a statement sequence within a referencing environment (a closure). These mechanisms enable programmers to define and implement control abstractions. Previous sections discussed expression evaluation. We assume the reader understands sequential statement execution and will not discuss it further. The following subsections discuss closures and their implications.

3.5.1 Closures

Control constructs manipulate units of work. For instance, the body of a for-loop is the unit of work passed to the for-loop construct. In the implementation of a control construct, we must be able to handle the work as a single item, without reference to the environment in which the work was defined. However, the work itself needs access to the environment in which it was defined. For example, the body of a for-loop needs access to the variables in its context, but the implementation of the for-loop does not. The local variables of the procedure in which the loop is embedded provide the context for the loop body. A nested procedure is another instance of work within a context.

Work-within-a-context has limited use when it only appears in language-defined control constructs. A limited facility for work-within-a-context is the procedure parameters in Pascal [Jensen and Wirth, 1975]. Since Pascal procedures may be nested, programmers can wrap the work in a procedure nested within the appropriate context. Pascal has no procedure variables, so programmers are limited in their ability to define control constructs. In addition, nested procedures are cumbersome and programmers tend not to use them. Modula-2 [Wirth, 1982] and C [Kernighan and Ritchie, 1978] provide procedure variables, but restrict their context to the global scope, their use in defining control constructs. Any restrictions on the assignment and scope of work-within-a-context limits its use in abstracting control.

The power of work-within-a-context really only becomes apparent when the handle on work is a first-class programming entity that may be defined within any referencing environment and manipulated as any other data. Lisp's *lambda* [Steele, 1984] and Smalltalk's *blocks* [Goldberg and Robson, 1983] provide a means for forming closures at any point in the program, assigning them to variables, passing them to other portions of the program, and executing them from any other portion of the program. Closures are similar to passing nested procedures in Pascal, but with the added power of assignment and the notational convenience of being defined in-line.

Matroshka uses closures to support control abstraction. These closures may accept a parameter, which enables the control construct to communicate with the body of work. Natasha represents closures with the notation

`closure` *parameter*: *type* `{ ...}`

When the parameter specification is absent, Natasha infers `empty` as the parameter type.

3.5.2 Activations as Objects

An operation in execution has an activation record. In Matroshka, this activation record is itself an object. So, this *activation object* may also respond to operations. Matroshka represents *closures* as anonymous operations on activation objects. As such, closures are implicitly an object reference and operation pair, and the specification for a closure yields a port in execution. That is, the run-time value of a closure specification is a port. This port is exactly the same mechanism introduced earlier (in section 3.2.4), and may be passed outside the enclosing scope to be executed by other objects. The activation provides the non-local referencing environment of the closure, so closures maintain access to the variables defined within the scope of the closure.[5]

3.5.3 Conditional Execution

Once a language supports closures as first-class entities, the language need supply only sequence, closure, and procedure invocation as the primitive control mechanisms. Conditional and repetitive execution become user-defined operations, outside the scope of the core language definition. For example, in Smalltalk, the objects of the boolean type have an `if` method (operation, procedure) that accepts a closure to execute if the object is true. With closures, languages can rely on control abstraction and need not define special syntax for conditional and iterative execution.

Natasha provides conditional and iterative execution through passing closures to pre-defined objects. For example, `boolean` objects (with values `true` or `false`) provide an '`if`' operation that accepts a port[6] and invokes the corresponding operation if the boolean value is true. Otherwise, it does not invoke the operation. For example,

```
(current > maximum) .if! closure { maximum := current; };
```

is the classic algorithm for keeping track of a maximum value. The expression `(current > maximum)` returns a `boolean` object, which executes its `if` operation and invokes the closure only if the object is `true`. The statement `maximum := current;` executes only if the closure is invoked. The type of the closure is `port'empty empty'`. Natasha provides several other predefined control constructs and encourages programmers to define more. For example, figure 3.3 shows how use the `while` operation to copy the input stream to the output stream. Control constructs may provide arguments to the closures they invoke. For example, the predefined `range` type provides iteration over an integral range. A range object consists of an integral lower bound and an upper bound. The object's `sequfor` operation accept a port (usually a closure), and for each value in its range, applies the value to the port. Figure 3.4 shows how to use `range` and `sequfor` to print the integers from 8 through 32, inclusive.

In an imperative parallel language that supports closures, we can define a parallel-for-loop construct that accepts a range of integers and a closure to execute for each

[5]The existence of the environment is identical to the existence of the object representing the environment. Any mechanisms for determining the existence of a normal data object also apply to environments.

[6]The port is usually, but not always, a closure. The `if` operation cannot distinguish between a port derived from a closure and one derived from an operation on an object. See section 3.5.2.

```
input: endfile;              ;; a character variable for input

boolean.while!               ;; a while loop
[ cond: closure              ;; compute loop condition
  { input.read![];           ;; read input character
    reply input ~= endfile;  ;; while input is not end of file
  };
  body: closure              ;; the body for the while loop
  { input.print![];          ;; write character to output
  };
];
```

Figure 3.3: Example Copying Input to Output

```
range.new![ from: 8; to: 32; ]  ;; create a new range object
.sequfor!                       ;; do a sequential for loop
closure i: integer              ;; a closure accepting an integer
{ i.print!3;                    ;; print the integer, width >= 3
  newline.print![];             ;; on a separate line
};
```

Figure 3.4: Example Printing a Range of Integers

---

integer within the range. We can define the semantics of the construct such that the only guarantees on the ordering of iterations are that no iteration will start before the construct starts and that iterations will complete before the construct completes. We call this the `forall` construct. This programmer defined construct makes weak guarantees on the concurrency and synchronization between iterations. Given this construct, we can easily provide both a sequential implementation (like the for-loop in sequential languages) and a parallel implementation that executes all iterations in parallel. Indeed, there are many more implementations of this construct. When programmers can choose the construct in the design of their program, they can select the most appropriate implementation of the construct for their architecture at compile time without affecting the semantics of the program. For example, on a uniprocessor they would choose a sequential implementation and on a multiprocessor they would choose a parallel implementation.

3.6 Early Reply from Invocations

Many abstractions can reply to their clients long before all the associated computations are complete. A mechanism for processing an operation asynchronously to the invoker 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 invoker, otherwise programmers will tend to avoid it.

The Matroshka programming model provides for asynchronous processing between a server and its clients with an *early reply* from operation invocation. After the reply, the operation may continue processing for an arbitrary time. Early reply maintains the local state of the operation after reply, but without necessarily introducing concurrent access to activation variables. Matroshka's early reply dissolves the binding between operation reply and operation termination prevalent in current imperative languages. The invoker waits for a reply, but does not wait for termination of the operation. Since the invoker may continue after receiving the reply, this early reply provides a source of parallelism. Indeed, *early reply is the sole source of parallelism provided by the Matroshka model.* It is a sufficient mechanism for implementing other forms of parallelism, such as asynchronous invocation. The use of early reply is transparent to the invoker, allowing the implementation of an object to change according to the system's need for concurrency. This mechanism is not new [Andrews *et al.*, 1988; Liskov *et al.*, 1986; Scott, 1987], but its expressive power does not appear to be widely recognized.

3.6.1 Early Reply in Natasha

We denote the value-returning reply statement with the keyword `reply` preceding the expression. For example,

```
reply 8;
```

Each method or closure may have (and execute) only one reply. When the reply is not present, Natasha infers a reply with an empty record as the last statement of the method.

3.6.2 Noting the Partial Order of Execution

The presence of an early reply in a method definition specifies a partial order of execution, which admits parallelism. For example, given the method definition,

```
method bigger par: integer
{ s_1; ...; s_i; reply par>8; s_j; ...; s_n; };
```

the statements invoking the corresponding operation

```
...; s_x; obj.bigger!(a+4); s_y; ...
```

result in the following partial order of execution:

$$\ldots \to s_x \to \text{eval } \texttt{(a+4)} \to s_1 \to \ldots \to s_i \to \text{eval } \texttt{par>8} \begin{array}{l} \nearrow\ s_j \to \ldots \to s_n \\ \\ \searrow\ s_y \to \ldots \end{array}$$

The statements $s_j \ldots s_n$ may execute in parallel with statement s_y and its successors.

An informal description of the partial execution order associated with a control construct can sometimes get involved. To state precisely and concisely the partial execution order defined by a construct, we introduce the following notation. This notation is not part of Matroshka, nor of Natasha. We use the notation in the following chapters.

Two events in the execution of an operation (or closure) are significant, its invocation and its reply. In describing the partial order provided by a control construct, we specify the partial order among these events using a set of rules. These rules do not implement the construct or define complete semantics, they merely state the temporal relationships. We use $\downarrow$ `operation` to signify the invocation of `operation`, $\uparrow$ `operation` to signify its reply, and $\rightarrow$ to signify that the implementation of the operation must ensure that the event on the left side precedes that on the right side. We also specify universally quantified variables in brackets after the rule. Since the invocation of an operation ($\downarrow$ `operation`) must necessarily precede its reply ($\uparrow$ `operation`), we omit such rules.

For example, the sequential for-loop operation `sequfor` on a range of integers `rng` from `lower` to `upper` has the following control semantics:

$\downarrow$ `rng.sequfor!work` $\rightarrow$ $\downarrow$ `work!lower`
$\uparrow$ `work!`i $\rightarrow$ $\downarrow$ `work!`$(i+1)$ $\quad$ [i : `lower` $\leq i <$ `upper`]
$\uparrow$ `work!upper` $\rightarrow$ $\uparrow$ `rng.sequfor!work`

These rules, respectively, are: the first iteration starts after the `sequfor` starts; the current iteration replies before the next one starts; and the last iteration replies before `sequfor` replies. This set of partial orders is actually a total order — no parallelism is possible.[7]

3.7 Summary

The Matroshka model supports *uniform data abstraction* via *objects* and *uniform control abstraction* via *closures*, which enables programmers to choose data or control abstractions early in program development, while choosing their implementations late.

Unless explicitly synchronized, object *operations execute concurrently* and may *reply early*, which enables the invoker and the operation to execute concurrently. Concurrent operation execution and early reply enable programmers to represent extensive parallelism among and between objects.

Matroshka provides communication via *synchronous operation invocation*. This, coupled with a *copy model* of variables and parameters, enables communication to scale with abstraction. Uniform abstraction and scalable communication enable the programmer and compiler to exploit parallelism at many levels within a program. Because parallelism may be exploited at many levels, and be rebound among these levels easily, programs can execute efficiently on many different architectures.

[7] Unless, of course, `work` replies early. These early replies are independent of the control construct, which can only order invoke and reply events.

4 Control Abstraction

Any problem in computer science can be solved
with another level of indirection.

This chapter introduces the use of control abstraction in parallel programming. We show how to build new control constructs, which improves our ability to express parallelism, how to provide multiple implementations for control constructs, which improves our ability to exploit parallelism, and how to use control abstraction to distribute processing. Control constructs represent what we *can* do, their implementations represent what we *choose* to do.

4.1 Expressing Parallelism

Given the importance of control flow in parallel programming and the multitude of constructs proposed, it seems premature to base a language on a small, fixed set of control constructs. In addition, if we are to encourage programmers to specify all potential parallelism, we must make it easy and natural to do so; no small set of control constructs will suffice. We require a mechanism to create new control constructs that precisely express the parallelism in an algorithm. Control abstraction provides us with the necessary flexibility and extensibility.

In this section we show how to use our mechanisms for control abstraction to build well-known parallel programming constructs. The techniques we use generalize to implementing other control constructs.

4.1.1 Fork and Join

In our first example, we use closures and early reply to implement a *fork-and-join* control mechanism similar to that provided in Mesa [Lampson and Redell, 1980]. The `fork` operation starts the computation of a value, which the `join` operation later retrieves. This *fork-and-join* is similar to a Multilisp *future*, except that programmers must request values explicitly with `join`.[1] Its declaration and semantics are:

[1] Our sample definition is somewhat restrictive in that the closure argument may only return integers. We could make our definition more general using some form of generic type facility; doing so is beyond the scope of this dissertation.

```
forkjoin: object
{ method fork work: port‘empty integer’ replies empty ;
  method join replies integer ;
};
mailbox: forkjoin.new![];
```

$\downarrow$ `mailbox.fork!work` $\rightarrow$ $\downarrow$ `work`
$\uparrow$ `work` $\rightarrow$ $\uparrow$ `mailbox.join![]`

These rules state that `fork` invokes `work`, and that `join` waits for the reply from `work` before replying. The user must invoke the join after the fork replies:

$\uparrow$ `mailbox.fork!work` $\rightarrow$ $\downarrow$ `mailbox.join![]`

It is not enough to invoke `join` after invoking `fork`, one wait for the reply from `join` before invoking `join`. These partial orders permit parallel execution. However, they do not guarantee parallelism because the rules state no order between the reply from `fork` and the invocation of `work`. The additional order:

$\uparrow$ `mailbox.fork!work` $\rightarrow$ $\downarrow$ `work`

which states that `fork` must reply before invoking `work`, would guarantee concurrent execution. We clarify the reason for omitting this rule in section 4.2.

Assume a `power` operation on integers that returns the object's value raised to the power given by the argument. We can use the definition of `fork` and `join` to evaluate two invocations of the `power` operation in parallel.

```
mailbox: forkjoin.new![];
mailbox.fork! closure { reply 3.power!4; };
n: 5.power!6;
sum: n + (mailbox.join![]);
```

Figure 4.1 shows the implementation of `fork` and `join` illustrates the use of early reply and explicit synchronization to achieve parallelism. This implementation uses only the mechanisms described in chapter 3, with the addition of atomic Boolean reads and writes. Busy-waiting synchronizes the two computations. We could easily change this implementation to use semaphores for synchronization and avoid busy waiting. The last method needs some explanation. The `repeat` operation executes its parameter while the parameter returns `true`. It completes whenever the parameter returns `false`. The postfix `~` operator signifies boolean negation. The first statement of the method for `join` is equivalent to the Pascal statement:

```
while not ready do ;
```

This statement busy waits on the boolean variable `ready`.

```
forkjoin: object
{ ready: false;
  result: 0;

  method fork work: port‘empty integer’
  { ready := false;
    reply [];             ;; caller continues
    result := (work![]);
    ready := true;
  };

  method join replies integer ;
  { boolean.repeat! closure { reply ready˜ };  ;; busy wait
    reply result;
  };
};
```

Figure 4.1: Example Implementation of Fork and Join

4.1.2 Cobegin

Our next example is the `cobegin` construct, which executes two closures in parallel and replies only when both have replied.[2] Its syntax and semantics are:

```
cobegin: object
{ method two [[ a: port‘empty empty’; b: port‘empty empty’; ]]
         replies empty;
};

do: cobegin.new![];
```

$\downarrow$ `do.two![ a: work1, b: work2 ]` $\rightarrow$ $\downarrow$ `work1`
$\downarrow$ `do.two![ a: work1, b: work2 ]` $\rightarrow$ $\downarrow$ `work2`
$\uparrow$ `work1` $\rightarrow$ $\uparrow$ `do.two![ a: work1, b: work2 ]`
$\uparrow$ `work2` $\rightarrow$ $\uparrow$ `do.two![ a: work1, b: work2 ]`

These orders permit but do not guarantee parallel execution. The orders that guarantee concurrent execution are:

$\downarrow$ `work1` $\rightarrow$ $\uparrow$ `work2`
$\downarrow$ `work2` $\rightarrow$ $\uparrow$ `work1`

[2]We could provide a more general n argument cobegin given a language that allows lists as arguments (*e.g.* Lisp).

These rules state that cobegin.two must invoke both closures before waiting on the replies.

Given the above definition, we can use this statement to implement the parallel evaluation of integer powers from the previous example.

```
n: 0;
m: 0;
do.two![ a: closure { n := (3.power!4); };
         b: closure { m := (5.power!6); }; ];
sum: n + m;
```

We use a valueless version of our previous definition of forkjoin and closures to build an implementation of cobegin.

```
cobegin: object
{ method two [[ a: port‘empty empty’; b: port‘empty empty’; ]]
  { mailbox: forkjoin.new![];
    mailbox.fork!work1;
    work2![];
    mailbox.join![];
  };
};
```

4.1.3 Forall

In our next example we define an parallel iterator over a range of integers, analogous to a parallel for loop or a CLU iterator [Liskov *et al.*, 1977].[3] Its syntax and semantics are:

```
range: object [[ from: integer; to: integer; ]]
{ method forall work: port‘integer empty’ replies empty;
};
```

$\downarrow$ rng.forall!work $\rightarrow$ $\downarrow$ work!i $\quad [i :$ from $\leq i \leq$ to$]$

$\downarrow$ work!i $\rightarrow$ $\downarrow$ work!$(i+1)$ $\quad [i :$ from $\leq i <$ to$]$

$\uparrow$ work (i) $\rightarrow$ $\uparrow$ rng.forall!work $\quad [i :$ from $\leq i \leq$ to$]$

These rules state, respectively, that: the forall starts before any iteration; iterations start in ascending order;[4] and all iterations reply before forall does. Again, we omit the rule that guarantees parallelism:

$\downarrow$ work!i $\rightarrow$ $\uparrow$ work!j $\quad [i, j :$ from $\leq i \leq$ to $\wedge$ from $\leq j \leq$ to$]$

which says that the implementation would have to start all iterations before waiting on the reply of any iteration.

Figure 4.2 shows the use cobegin and recursion to build a parallel *divide-and-conquer* implementation of forall that uses a binary tree to start all instances of work.

[3] Unlike CLU, our emphasis is on the separation of semantics and implementation for general control constructs, rather than the ability to iterate over the values of any abstract type. In addition, we generalize CLU iterators from sequential execution to parallel execution.

[4] This rule is useful primarily when using forall to implement other control constructs.

```
range: object [[ from: integer; to: integer; ]]
{ method forall work: port‘integer empty’
  { (from = to) .if! closure { work!from; };
    (from < to) .if! closure
    { middle: (from + to) / 2;
      cobegin.two!
      [ a: closure
        { range.new![ from: from; to: middle; ].forall!work; };
        b: closure
        { range.new![ from: middle + 1; to: to; ].forall!work; };
      ];
    };
  };
};
```

Figure 4.2: Example Implementation of Forall

This implementation executes each iteration of `forall` in parallel, and therefore would only be appropriate when the granularity of parallelism supported by the architecture is well matched to the granularity of each iteration. Otherwise, it would be better to use an alternative parallel implementation that creates fewer tasks than iterations, where each task executes several iterations. The degree of parallelism provided by this alternative implementation may change easily. However, the degree of parallelism cannot be selected using annotations for operation implementations alone because the degree is a quantitative attribute. This is in contrast to a qualitative change in implementation. We can use *quantitative* annotations to indicate the desired grain. For example, an implementation of `forall` that grouped iterations (named with the `_GROUPED` annotation), could accept a `GRAIN` annotation with an integer value. This annotation can select the grain of parallelism.

These examples show the power of control abstraction when used to define parallel control flow mechanisms. Using closures and early reply we can represent many different forms of parallelism. In particular, we used closures, early reply, and a synchronization variable to implement `forkjoin`. We then used `forkjoin` to implement `cobegin`, and `cobegin` with recursion to implement `forall`.

4.2 Exploiting Parallelism

Our approach to adapting the exploitation of parallelism to different architectures relies on the programmer specifying lots of potential parallelism and then implementing the appropriate subset. The programmer does so by using constructs that represent potential parallelism, and then selecting the appropriate implementations. The *algorithm* determines the control constructs used to represent potential parallelism; the *architecture* determines the implementations used to exploit parallelism.

4.2.1 Multiple Implementations

Data operations often have multiple implementations. For example, matrix addition has sequential, vector, and parallel implementations, each appropriate to different architectures. We can extend this approach to control constructs as well. Control abstraction permits multiple implementations for a given control construct. These implementations can exploit differing sources of parallelism, subject to the partial order constraints of the construct. In effect, the definition of a control construct represents potential parallelism; the implementation defines the exploited parallelism.

Our rules for each of the control constructs in section 4.1 deliberately left the partial orders underspecified to admit either a parallel or sequential implementation. We complete the example constructs in section 4.1 by providing alternative implementations here. To distinguish each implementation, we annotate it with a descriptive identifier that follows the operation identifier. We assume programmers will annotate each implementation of a control construct with a name that describes the degree of parallelism exploited by the implementation. For example, our parallel divide-and-conquer implementation of `forall` from the previous section would be annotated as follows:

```
method forall_DIVIDED ...
```

whereas the alternative parallel implementation that groups iterations together for execution would be annotated this way:

```
method forall_GROUPED ...
```

As an example of implementation flexibility, consider a sequential implementation of `forkjoin` that computes the result of the `join` operation first, and then continues.

```
forkjoin_SEQUENTIAL: object
{ result: 0;

  method fork work: port‘empty empty’
  { result := (work![]); };  ;; caller waits for work to finish

  method join
  { reply result; };
};
```

Using this sequential implementation of `forkjoin` within the implementation of `cobegin` produces a sequential implementation of `cobegin`. Alternatively, we could change the implementation of `cobegin` to execute the two statements in sequence without the use of `forkjoin`.

```
cobegin: object_SEQUENTIAL
{ method two [[ a: port'empty empty';
                b: port'empty empty'; ]]
  { a![]; b![]; };
};
```

Although either approach results in a sequential implementation of `cobegin`, changing the implementation of `cobegin` has two advantages: the implementation of `cobegin` would no longer require an implementation of `forkjoin` and we would avoid the overhead of invoking the `fork` and `join` operations.

Similarly, we can build a sequential implementation of `forall` either by using an embedded sequential implementation of `cobegin` or by changing the implementation of `forall` to use the sequential `sequfor` construct. Once again there is an advantage to changing the implementation of `forall` — the `sequfor` construct has a particularly efficient implementation based on machine instructions.

```
range: object [[ from: integer; to: integer; ]]
{ method forall_SEQUENTIAL work: port'integer empty'
  { self.sequfor!work; };
};
```

4.2.2 Selecting Implementations

Once we have multiple implementations for a given control construct, some using varying amounts of parallelism, we can control the amount of parallelism we exploit during execution by selecting appropriate implementations at the point of use. One simple technique for selecting implementations is program annotations. Each use of a construct can select an appropriate implementation by placing the corresponding annotation after the operation identifier in its invocation.[5,6] For example,

```
3.power_PARALLEL!4
```

computes 3^4 with a parallel implementation of `power`.

A wide range of choices for exploiting parallelism are possible by choosing different implementations of a few predefined constructs (such as `forkjoin`, `cobegin` and `forall`). When the library of predefined implementations does not provide enough architectural adaptability, a new implementation may be necessary. However, separating the semantics of use from the implementation of a control mechanism significantly simplifies the task of exploiting a different subset of the potential parallelism.

In figure 4.3, we illustrate the use of annotations to select a particular parallelization for Quicksort. We consider two potential sources of parallelism. When the array is partitioned, the search for an element in the bottom half of the array that belongs in

[5] A reasonable set of default annotations will reduce the coding burden on the programmer. In particular, we recommend that the default implementation be sequential.

[6] Smart compilers could choose these annotations. The techniques for the automatic selection of different implementations for sequential data structures [Low, 1976] may apply to choosing implementations for control constructs. We do not assume such a compiler.

```
sortable_array: object
{ sorting: array‘SIZE integer’.new! closure { reply 0; };

  method quicksort_COARSE [[ lower: integer; upper: integer; ]]
  { lower < upper .if! closure
    { rising: lower;
      falling: upper;
      key: sorting#lower,@;  ;; i.e. sorting[lower] as an r-value
      boolean.while!
      [ cond: closure
        { cobegin_SEQUENTIAL.new![].two!
          [ a: closure { boolean.repeat! closure
                          { rising :=+ 1;
                            reply key >= (sorting#rising,@);
                          };
                        };
            b: closure { boolean.repeat! closure
                          { falling :=- 1;
                            reply key < (sorting#falling,@);
                          };
                        };
          ];
          reply rising <= falling;
        };
        body: closure { temp: sorting#rising,@;
                        sorting#rising, := (sorting#falling,@);
                        sorting#falling, := temp;
                      };
      ];
      sorting#lower, := sorting#falling,@;
      sorting#falling, := key;
      cobegin_PARALLEL.new![].two!
      [ a: closure
        { self.quicksort_COARSE![ lower: lower;
                                  upper: falling; ]; };
        b: closure
        { self.quicksort_COARSE![ lower: falling+1;
                                  upper: upper; ]; };
      ];
    };
  };
};
```

Figure 4.3: Example Annotated Quicksort

the top half can occur in parallel with a similar search that takes place in the top half. Similarly, the two recursive calls to Quicksort on each half of the array can occur in parallel.

In this particular implementation we chose to exploit the coarse-grain parallelism available during the recursive calls (using the `_PARALLEL` annotation to select the parallel implementation of the second `cobegin`) and chose not to exploit the finer-grain parallelism available during partition. We could experiment with fine-grain parallelism by simply changing the `_SEQUENTIAL` annotation to select the parallel implementation of the first `cobegin`. We annotate the resulting implementation of quicksort with `_COARSE`.

Current parallelizing compilers could probably find the fine grain parallelism automatically (there are no overlapping writes to variables), even though this parallelism may not be useful on many multiprocessors. The more important source of parallelism available in the recursive calls would be much more difficult, if not impossible, to find automatically, because it must prove a partition on the data.

The control constructs of section 4.1 have several possible implementations. We may adapt many parallel programs simply by choosing to use different implementations of these constructs on different architectures.

4.3 Distributing Processing

Our approach to distribution relies on data abstraction for data distribution and control abstraction for process distribution. In this section we concentrate on process distribution via control abstraction.

4.3.1 Distributed Implementations of Control Constructs

We adopt the approach of Par [Coffin and Andrews, 1989] and Chameleon [Alverson and Notkin, 1991] in providing control construct implementations that distribute computations. For example, we add additional implementations of the `forall` control construct to provide distribution. Given the parallel loop:

```
range.new![ from: 0; to: 100; ].forall!
closure i: integer { ... };
```

we can distribute the computation by selecting a distributed implementations of `forall`. For example,

```
range.new![ from: 0; to: 100; ].forall_MODULAR!
closure i: integer { ... };
```

specifies the implementation that distributes processes in a modular fashion. That is iteration i executes on processor $i \bmod p$ where p is the number of processors. We expect programming languages based on Matroshka to provide implementations of predefined control constructs corresponding to several alternate distributions.

Programmers can provide alternate process distributions by defining new implementations of control constructs. For example,

```
range: object [[ from: integer; to: integer; ]]
{ method forall_PAIRED_MODULAR work: port`integer empty'
  { range.new![ from: 0; to: (from+to)/2; ].forall_MODULAR!
    closure index: integer
    { work!(from+(index*2));
      work!(from+(index*2)+1);
    };
  };
};
```

defines a new implementation of `forall` that distributes *pairs* of iterations in a modular fashion, rather than single iterations.

Programmers can define new control constructs, which describe their algorithms more precisely. For example, we can iterate over even integers in parallel with the following code.

```
range.new![ from: 0; to: whatever; ].forall!
closure index: integer
{ (index%2 = 0).if! closure { ..... }; };
```

This code throws away half the parallelism it generates. We can define a new control construct, `foreven`, that iterates over even integers. We use it like this:

```
range.new![ from: 0; to: whatever; ].foreven!
closure index: integer { ..... };
```

This example does not specify more parallelism than it uses. We can implement `foreven` as follows.

```
range: object [[ from: integer; to: integer; ]]
{ method foreven_MODULAR work: port`integer empty'
  { range.new![ from: 0; to: (from+to)/2; ].forall_MODULAR!
    closure index: integer { work!(from+(index*2)); };
  };
};
```

This example illustrates how control abstraction can achieve precise distribution of processing.

4.3.2 Combining Data and Control Abstraction

Data and control abstraction reach their full potential when combined, that is when data abstractions export control operations. For example, assume we have a distributed implementation of a tree. We can define an *iterator* for the tree. An iterator is a control construct that applies work to each element of a data structure. Iterators are generally useful in data abstraction [Liskov *et al.*, 1977] because they help separate the access to a data structure from the implementation of that structure. In parallel programming, iterators help to link data distribution with process distribution. When data distribution and process distribution are distinct, programmers must explicitly coordinate the two. This is difficult for non-regular computations that determine data distribution dynamically. On the other hand, iterators can manage process distribution by shipping each task to the processor with the corresponding element.

5 Extended Examples

Example is always more efficacious than precept.
— Samuel Johnson, 1759

This chapter presents two extended examples, Gaussian elimination and subgraph isomorphism. These examples are not complete applications; application programmers can expect to find more potential parallelism than these examples provide. On the other hand, these examples represent non-trivial computations, such as might be found at the core of parallel applications.

5.1 Gaussian Elimination

We will use Gaussian elimination[1] as an example of using control abstraction for architectural adaptability. Gaussian elimination is a well-known algorithm, has nontrivial synchronization constraints, and admits several different exploitations of parallelism. Our goal is to create a single source program that represents these different exploitations, each of which can be selected by an appropriate choice of annotations, and thereby duplicate previous extensive experience in the development and tuning of parallel Gaussian elimination on the BBN Butterfly [Crowther *et al.*, 1985; Thomas, 1985; LeBlanc, 1986; LeBlanc, 1988].

In solving a set of linear equations using Gaussian elimination, we first compute an upper triangular matrix from the coefficient matrix M, producing a modified vector of unknowns, which we then determine using back-substitution. Since back-substitution is a small percentage of the total time required to solve the equations, it was not done in any of the earlier experiments, and we will not consider it here. We concentrate on computing the upper triangular matrix by eliminating (zeroing) entries in the lower triangle (those entries below the diagonal), as illustrated in figure 5.1. To eliminate an entry $M_{i,j}$, we replace row M_i with $M_i - M_j \frac{M_{i,j}}{M_{j,j}}$, where M_j is known as the pivot row. However, we cannot eliminate $M_{i,j}$ until after row M_j is stable, *i.e.*, $M_{j,k} = 0, \forall k < j$. In addition, all previous entries in row i must already be eliminated, *i.e.*, $M_{i,k} = 0, \forall k < j$.

[1] We choose pivot equations in index order; numerically robust programs choose pivot equations based on the data. We use the fragile algorithm for presentation and historical reasons.

$$
\begin{array}{cccccc}
? & ? & ? & ? & ? & ? \\
0 & ? & ? & ? & ? & ? \\
0 & 0 & ? & ? & ? & ? \\
0 & 0 & 0 & ? & ? & ? \\
0 & 0 & 0 & 0 & ? & ? \\
0 & 0 & 0 & 0 & 0 & ?
\end{array}
$$

Figure 5.1: Gaussian Element Elimination Goal

These two synchronization constraints limit the amount of parallelism that we can expect to achieve.

We present this example as a sequence of programs derived from the standard sequential algorithm, reflecting earlier experiences with this application. Later, in chapter 6, we propose a method that avoids the intermediate steps in this sequence and proceeds directly to the final form.

5.1.1 The First Cut

Our first attempt is based on the standard (non-pivoting) sequential algorithm for upper triangulation.

```
system: array`SIZE array`SIZE float''.new!.....;
range.new![ from: 0; to: SIZE-2; ].sequfor!
closure pivot: integer
{ range.new![ from: pivot+1; to: SIZE-1; ].sequfor!
  closure reduce: integer
  { fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
    range.new![ from: pivot; to: SIZE-1; ].sequfor!
    closure variable: integer
    { system#reduce,#variable, :=- (system#pivot,#variable,*fraction);
    };
  };
};
```

The inner loop eliminates a single entry in the matrix; the middle loop eliminates an entire column (below the diagonal); and the outer loop eliminates the entire lower triangle.

The straightforward parallel implementation of this algorithm parallelizes the two inner loops with `forall`.[2] Section 4.1 showed that the `forall` construct has both a parallel and sequential implementation. By using annotations to select a parallel implementation for both loops, we can create an extremely fine-grain parallel implementation.

```
system: array`SIZE array`SIZE float''.new!.....;
range.new![ from: 0; to: SIZE-2; ].sequfor!
closure pivot: integer
{ range.new![ from: pivot+1; to: SIZE-1; ].forall_DIVIDED!
  closure reduce: integer
  { fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
    range.new![ from: pivot; to: SIZE-1; ].forall_DIVIDED!
    closure variable: integer
    { system#reduce,#variable :=- (system#pivot,#variable,*fraction);
    };
  };
};
```

Vector processors could exploit the parallelism in the inner loop by invoking vector instructions, rather than using the parallel implementation of `forall`. On a vector processor we would expect our compiler to recognize a `_VECTOR` annotation and produce vector instructions for the innermost loop.[3] To port the program to a vector multiprocessor, such as the Alliant FX, we use both a parallel implementation for the outer `forall` and a vector implementation for the inner `forall`.

The Butterfly lacks vector instructions, and cannot profitably exploit the parallelism in the inner loop. Therefore, we can select an implementation that does not attempt to exploit fine-grain parallelism by choosing the `_SEQUENTIAL` annotation for the inner loop. The Butterfly can exploit the parallelism in the middle loop, so we choose the `_DIVIDED` annotation for the middle loop. This was precisely the first program developed in earlier work [LeBlanc, 1988].

The execution speed of the sequential and parallel annotations on the middle loop of the Natasha program on the Butterfly appear in figure 5.2.

5.1.2 Distribution

The initial parallel performance of our program on the Butterfly is not good. In reviewing the program, we note that there is no indication of data distribution. In a NUMA machine, such as the Butterfly, we must distribute data and processing to obtain efficient execution.

[2] Iterations of the outermost loop cannot execute in parallel because of the data flow constraint that an equation cannot be used as a pivot until it has been reduced completely.

[3] We claim no particular advantage over vectorizing compilers here, however this example does show that control abstraction can represent fine-grain parallelism explicitly.

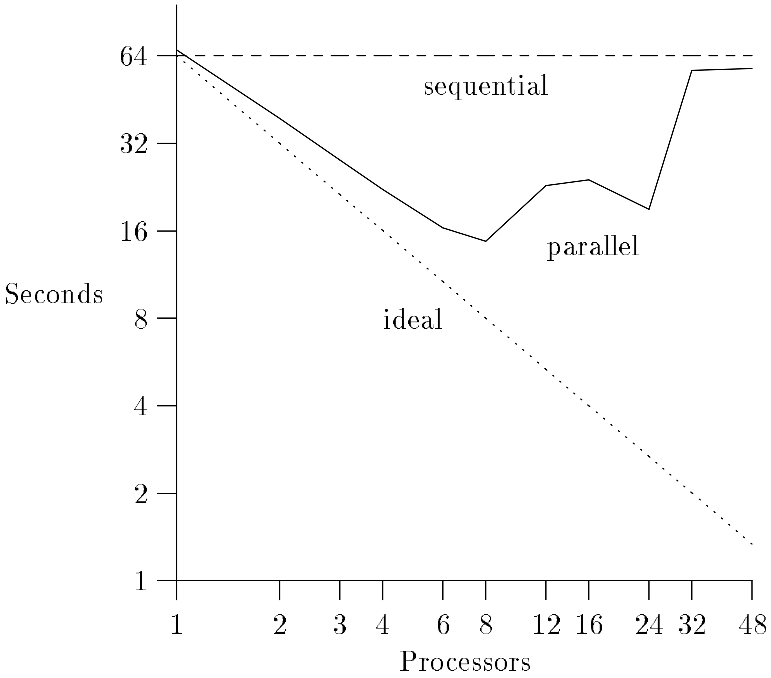

Figure 5.2: Performance of First Gaussian Program

The obvious approach for data distribution in Gaussian elimination is to distribute the equations equally among processors. Given n equations and p processors, there are two simple ways to distribute equations. The first distribution assigns equations $i\lceil n/p\rceil, i\lceil n/p\rceil+1, \ldots, (i+1)\lceil n/p\rceil-1$ to processor i (the *divided* distribution), assuming zero based indexing. The second distribution assigns equations $i, p+i, 2p+i \ldots$ to processor i (the *modular* distribution). The data alone do not appear to decide between the two distributions; we should look to process distribution.

The obvious approach for process distribution is also to distribute reductions equally among processors. We should use the same distribution strategy that we use for data, so that data and processing are co-located. For processing, the distribution method does matter. If we use the divided distribution, we may have excessive waiting at the start of the program while the first few pivot equations are reduced, and at the end of the program while the last processor finishes up the last few reductions. The modular distribution avoids both these problems by distributing both the first few equations and the last few equations among the processors. Given that our process distribution favors modular distributions, we should select the corresponding data distribution. The resulting program is:

```
system: array_MODULAR`SIZE array`SIZE float''.new!.....;
range.new![ from: 0; to: SIZE-2; ].sequfor!
closure pivot: integer
{ range.new![ from: pivot+1; to: SIZE-1; ].forall_MODULAR!
  closure reduce: integer
  { fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
    range.new![ from: pivot; to: SIZE-1; ].forall_SEQUENTIAL!
    closure variable: integer
    { system#reduce,#variable, :=- (system#pivot,#variable,*fraction);
    };
  };
};
```

Note that the program differs only in the annotations used.

The performance of the `_DIVIDED` and `_MODULAR` annotations appears in figure 5.3. As expected, the `_MODULAR` distribution performs better.

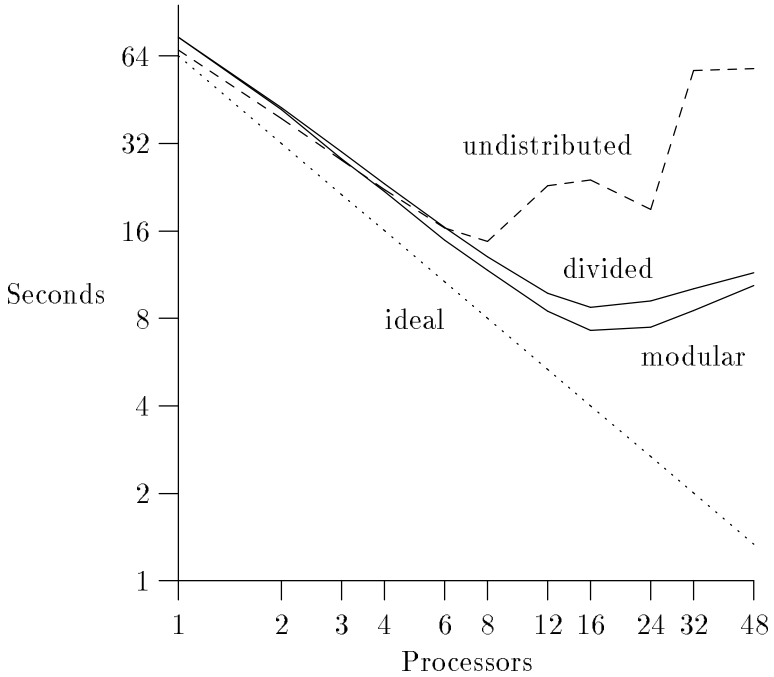

Figure 5.3: Performance of Distributed Gaussian

5.1.3 Improving Parallelism

The Gaussian program's performance is not as good as it could be. To improve the performance, we will concentrate on the order in which elements are eliminated. The annotation of the inner loop does not affect the order of eliminations, so we will not consider it in this analysis. The sequential annotation of the middle loop results in the order of eliminations shown in figure 5.4. The *total order* of eliminations implies

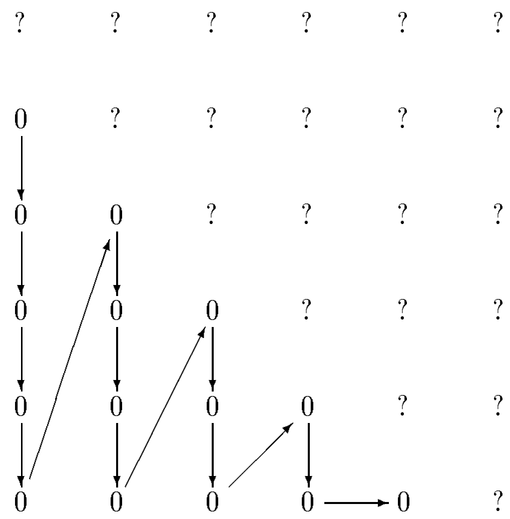

Figure 5.4: Sequential Gaussian Element Elimination

sequential elimination, and vice-versa.

When we use the `_MODULAR` annotation on the middle loop, we obtain the *partial* order of eliminations shown in figure 5.5. The resulting program exhibits a series of phases separated by the selection of a pivot. Experimentation with this parallelization of Gaussian elimination highlighted the time processors spent waiting for other processors to complete each phase. These empirical results led to the development of an implementation based on the synchronization constraints for the problem.

The original sequential algorithm contains implicit synchronization constraints that caused us to serialize the outermost loop. The data flow constraints for the algorithm are that pivot equations must be applied to a given equation in order, and an equation must be reduced completely before it can be used as a pivot. In our notation, the constraints are:

$$\uparrow i \text{ reduce } j \rightarrow \downarrow k \text{ reduce } j \qquad [i,j,k : 1 \leq i < j \leq \text{size} \wedge i < k \leq \text{size}]$$
$$\uparrow i \text{ reduce } j \rightarrow \downarrow j \text{ reduce } k \qquad [i,j,k : 1 \leq i < j \leq \text{size} \wedge j < k \leq \text{size}]$$

The constrains also appear in figure 5.6. We can enforce these constraints with explicit synchronization, resulting in the following program. We use blocking condition variables with wait and signal operations for synchronization.

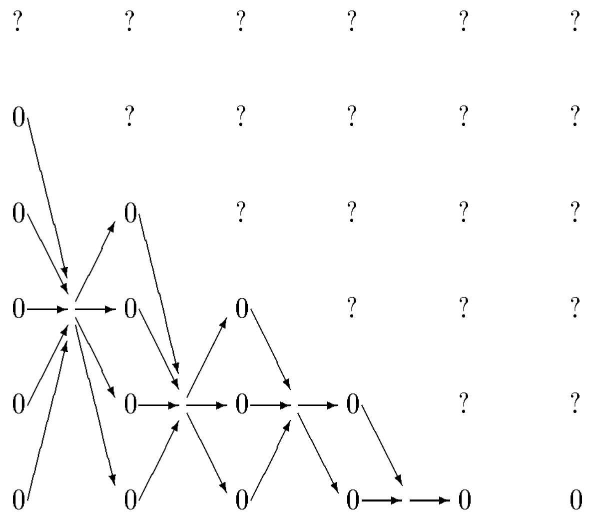

Figure 5.5: Phased Gaussian Element Elimination

```
system: array‘SIZE array‘SIZE float’’;
done: array‘SIZE condition’;
done#1,.signal![];
range.new![ from: 1; to: SIZE-1; ].forall_DIVIDED!
closure reduce: integer
{ range.new![ from: 0; to: reduce-1; ].sequfor!
  closure pivot: integer
  { done#pivot,.wait![];
    fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
    range.new![ from: pivot; to: SIZE-1; ].forall!
    closure variable: integer
    { system#reduce,#variable :=- (system#pivot,#variable,*fraction);
    };
  };
  done#reduce.signal![];
};
```

Since this new version of the program has fewer constraints on parallelism, we expect it may execute faster. Figure 5.7 shows that the fully parallel version does execute faster, particularly with more processors.

Note that we cannot derive this particular version of the program from our previous versions by selecting an appropriate combination of implementation choices for the `forall` construct. In addition, we cannot select the use of explicit synchronization in this new program in tandem with the parallelism we plan to exploit, since explicit synchronization is embedded in the body of the loop. The fault, however, lies not in our approach, but in our failure to use the full power of control abstraction. In particular,

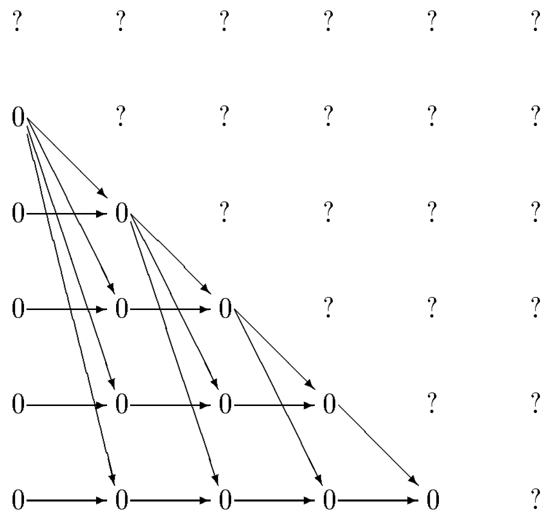

Figure 5.6: Fully Parallel Gaussian Element Elimination

we did not capture the order in which we select pivot and reduction equation pairs in a single control construct.

5.1.4 A New Control Construct

We can define a control construct, `forpairs`, that takes two parameters: the number of equations in the system, and the work for each pivot and reduction pair, which is to reduce a single equation given a pivot. The construct encapsulates all parallelism and synchronization in selecting pairs of pivot and reduction equations. We encapsulate the reduction within a closure; its parameters are the indices of the pivot and reduction equations. The `forpairs` construct invokes the closure with the appropriate pairings, while maintaining the synchronization necessary for correct execution. Its syntax and semantics are:

```
pair: [[ pivot: integer; reduce: integer; ]];

object: triangulate size: integer
{ method forpairs work: port‘pair empty’ replies empty;
};
t: trangulate.new!size;

↓t.forpairs!work
 → ↓work![ pivot: i; reduce: j; ]                    [i,j : 1 ≤ i < j ≤ size]
↑work![ pivot: i; reduce: j; ]                      [i,j,k : 1 ≤ i < j ≤ size
 → ↓work![ pivot: k; reduce: j; ]                                ∧ i < k ≤ size]
↑work![ pivot: i; reduce: j; ]                      [i,j,k : 1 ≤ i < j ≤ size
 → ↓work![ pivot: j; reduce: k; ]                                ∧ i < k ≤ size]
```

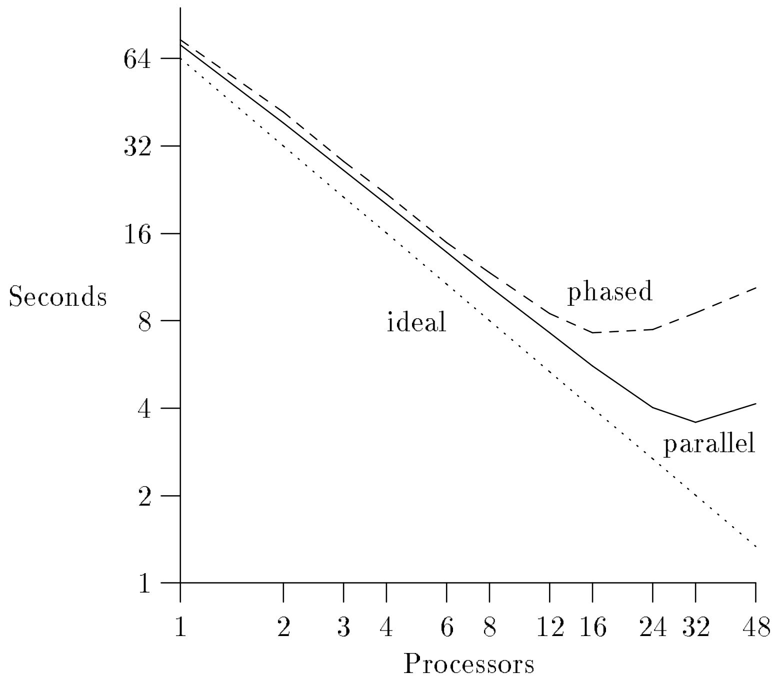

Figure 5.7: Performance of Phased and Fully Parallel Gaussian

This construct has several implementations, corresponding to the different exploitations of potential parallelism discussed above. A sequential implementation of `forpairs` is:

```
method forpairs_SEQUENTIAL work: port‘pair empty’
{ range.new![ from: 0; to: SIZE-2; ].sequfor!
  closure pivot: integer
  { range.new![ from: pivot+1; to: SIZE-1; ].forall_SEQUENTIAL!
    closure reduce: integer
    { work![ pivot: pivot; reduce: reduce; ];
    };
  };
};
```

By substituting `forall_DIVIDED` for `forall_SEQUENTIAL` we get `forpairs_PHASED`, which exploits the same parallelism as the earlier phased version of the program. In addition, we can also substitute `forall_GROUPED` for `forall_SEQUENTIAL` to obtain a `forpairs_PHASED_GROUPED`.

We exploit the more extensive parallelism based on the problem's synchronization constraints with the following implementation:

```
method forpairs_SYNCHED work: port‘pair empty’
{ done: array‘SIZE condition’;
  done#1,.signal![];
  range.new![ from: 1; to: SIZE-1; ].forall_DIVIDED!
  closure reduce: integer
  { range.new![ from: 0; to: reduce-1; ].sequfor!
    closure pivot: integer
    { done#pivot,.wait![];
      work![ pivot: pivot; reduce: reduce; ];
    };
    done#reduce.signal![];
  };
};
```

This implementation admits more parallelism than `forpairs_PHASED`, but may have higher execution overhead because of the need to accommodate synchronization. As earlier, we can substitute `forall_GROUPED` for `forall_DIVIDED` to obtain a `forpairs_SYNCHED_GROUPED`.

When rewritten to use `forpairs`, the fully parallel code to form the upper triangular matrix looks like this:

```
system: array‘SIZE array‘SIZE float’’;
triangluate.new!SIZE.forpairs_SYNCHED!
closure [[ pivot: integer; reduce: integer; ]]
{ fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
  range.new![ from: pivot; to: SIZE-1; ].forall_DIVIDED!
  closure variable: integer
  { system#reduce,#variable :=- (system#pivot,#variable, * fraction);
  };
};
```

By selecting an appropriate implementation of `forpairs` and the `forall` construct embedded in its body, we can describe all the previous parallelizations of this problem. Programmers can select twenty different implementations of this program by varying the two annotations to select a divide-and-conquer, grouped, sequential, or vector implementation of `forall`, and a synchronized divide-and-conquer, synchronized grouped, phased divide-and-conquer, phased grouped, or sequential implementation of `forpairs`. Our experience has shown that `forpairs_SYNCHED_GROUPED` and `forall_SEQUENTIAL` is the most efficient implementation on the Butterfly. The `forpairs_SYNCHED_GROUPED` and `forall_VECTOR` annotations are most appropriate for the Alliant. This same program has been ported to a Sun workstation by selecting `forpairs_SEQUENTIAL` and `forall_SEQUENTIAL`. The key to the adaptability in our solution is the introduction of an algorithm-specific control construct.

5.1.5 Distribution Revisited

After introducing an additional parallelization of Gaussian elimination, we should reconsider the distribution of processing and data. Reconsidering distribution ensures that we obtain maximum performance. Figure 5.8 shows that the `_DIVIDED` distribution

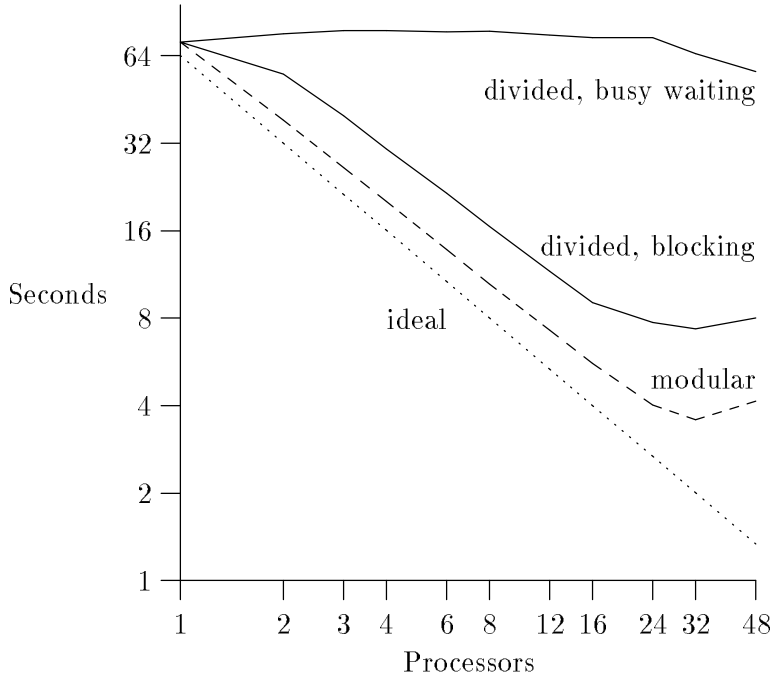

Figure 5.8: Performance of Fully Parallel Gaussian Distributions

performs much worse than the `_MODULAR` distribution. This poor performance results from the processors attempting to work against the synchronization constraints of the algorithm. This mismatch between distribution and synchronization is particularly noticeable when using busy waiting, where the program is effectively serialized.

5.1.6 Communication

When we examine the communication pattern in the Gaussian example thus far, we find that there is much fine-grain communication during a reduction. This communication arises when retrieving individual elements from the read-shared pivot row.

In regular computations such as Gaussian elimination, we can predict when read-sharing will occur and rely on mapping operations [Coffin and Andrews, 1989] to replicate equations. When each equation stabilizes, we can call a mapping operation that instructs the array that it is now appropriate to replicate the equation. These mapping operations are a complex form of annotation.

Another approach to solving the read-sharing problem is to pass the pivot equation as a parameter to the reduction equation. To do this, we recognize that an equation is itself an important data abstraction. The work we pass to the `forpairs` control construct now consists primarily of the invocation of a reduction operation on an equation.

```
system: array‘SIZE equation’;
triangluate.new!SIZE.forpairs_SYNCHED!
closure [[ pivot: integer; reduce: integer; ]]
{ system#reduce,.reduction![ pivot: pivot; value: system#pivot,@; ];
};
```

where equations have the implementation

```
equation: object store: array‘SIZE integer’
{
  method reduction [[ pivot: integer; value: equation; ]]
  { fraction: store#pivot, / (value#pivot,@);
    range.new![ from: pivot; to: SIZE-1; ].forall_SEQUENTIAL!
    closure variable: integer
    { store#variable, :=- (value#variable, * fraction);
    };
  };
};
```

Figure 5.9 shows the performance improvement on the Butterfly associated with abstracting equations. The figure understates the performance difference because the

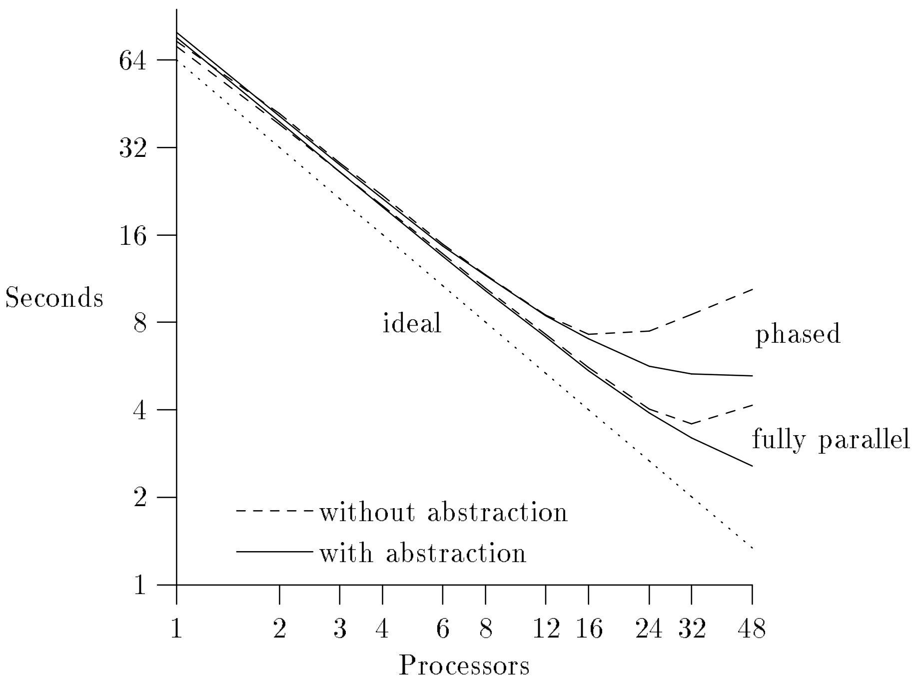

Figure 5.9: Performance of Gaussian With Data Abstraction

prototype Natasha implementation introduces an artificially high computation to communication ratio in the non-abstract version, but does not do so in the abstract version. (See section 7.3.3.)

5.2 Subgraph Isomorphism

This section highlights the interaction of data abstraction and control abstraction. In particular, we show that data abstractions with embedded control abstractions are a powerful and adaptable representation of potential parallelism. Our example is subgraph isomorphism. The problem is to find the set of isomorphisms from a small graph to subgraphs of a larger graph. We present a generalized form of the algorithm developed for the 1986 DARPA parallel architecture benchmark [Costanzo *et al.*, 1986], which is based on Ullman's sequential tree-search algorithm [Ullman, 1976]. The algorithm has four grains of parallelism, however the benchmark program only exploited one grain. Without a method and language to support architectural adaptability, there was not enough time available during the benchmark to write the different programs necessary to exploit the different grains.

A graph isomorphism is a mapping from each vertex in one graph to a unique vertex in the second, such that if two vertices are connected in the first graph then their corresponding vertices in the second graph are also connected. In subgraph isomorphism, the first graph is smaller than the second graph and we ask for an isomorphism from the first graph to a subgraph of the second graph.

Our algorithm for finding isomorphisms postulates a mapping from one vertex in the small graph (a small vertex) to a vertex in the large graph (a large vertex). This mapping constrains the possible mappings for other small vertices. We then postulate a mapping for the next small vertex, and constrain mappings based on that postulate. Because each small vertex we choose may have several possible mappings, we must search each possibility. This search takes the form of a tree, where nodes at level i correspond to postulated mappings for small vertex i. The mappings at levels 1 through $i-1$ constrain the possible mappings at level i.

Each node in the tree must represent the remaining possible mappings for each small vertex. At the root of the tree, each small vertex may map to any large vertex. The root's children have a single mapping for the first small vertex, and then several possible mappings for the remaining small vertices. Tree nodes that have no possible mapping for at least one small vertex are invalid isomorphisms, and we may prune these nodes from the search tree. The leaves of the tree will have at most one mapping for each small vertex. Leaves with exactly one mapping for each small vertex represent complete isomorphisms.

Relative to the search time, initializing the search takes little time. So, we will not discuss the initialization except to note that some static constraints may eliminate possible mappings in the root node.

Representation

In our representation, each vertex has an integer label, from 1 to the maximum number of vertices. We represent each graph by an array, where each element of the array corresponds to a vertex and contains the set of integer labels for the vertex's immediate neighbors.

```
graph‘SIZE’: object
{ store: array‘SIZE set‘SIZE integer’’;
};
small_neighbors: graph‘SMALL’.new![];
large_neighbors: graph‘LARGE’.new![];
```

Small vertex `1` connects to the small vertices in `small_neighbors#1`.

We represent tree nodes with an array of sets. Each element of the array corresponds to a small vertex and the set contains the integer labels of large vertices to which the small vertex might map.

```
tree_node‘SMALL LARGE’: object
{ store: array‘SMALL set‘LARGE integer’’;
};
node: tree_node‘SMALL LARGE’.new![];
```

Small vertex `1` may map to any element of `node#1`.

Searching Possibilities

The coarsest grain of parallelism arises when searching among the various possibilities for a given small vertex. We call the small vertex under consideration the *focus*, and we call the current possible mapping the *image*. Given a set of possibilities in the set `possible#focus`, we need to examine each postulated mapping. Using a language with the typical fixed control constructs, we would write:

```
method search
{ range.new![ from: 0; to: maximum_large; ].forall!
  closure image: integer
  { possible#focus,.member!image .if! closure
    { self@.examine!image;
    };
  };
};
```

When selecting a parallel implementation of `forall`, we must pay the overhead of starting each task. Because most possible mappings will be near empty, the `if` condition is usually false, and most tasks will immediately finish. This immediate ending of tasks represents a substantial amount of wasted effort.

The problem with the above code is that we wish to iterate over the elements of the set, but the `forall` forces us to iterate over the representation for the set and then test for membership. A better approach is to combine data abstraction and control abstraction and define a parallel iterator for sets, generalizing the CLU [Liskov *et al.*, 1977] iterators.[4,5] Iterators enable us to state precisely that the parallelism is over the elements present, and not over potential elements. We define a `forall_elements` operation that executes a closure (or operation) for each element of the set.

```
set‘integer’: object
{ method forall_elements work: port‘integer empty’ replies empty;
};
```

$$\downarrow \texttt{members.forall\_elements!work} \rightarrow \downarrow \texttt{work!}i \qquad [i : i \in \texttt{members}]$$
$$\uparrow \texttt{work!}i \rightarrow \uparrow \texttt{members.forall\_elements!work} \qquad [i : i \in \texttt{members}]$$

Iterators are also useful in the distribution of processes with data.

Given the `forall_elements` operation, we rewrite the `search` operation as:

```
method search
{ node#current_small,.forall_elements!
  closure image: integer
  { self@.examine!image;
  };
};
```

This representation is clearer and potentially more efficient. Building the iterator requires a mechanism to define control abstractions that interact with data abstractions. The closure mechanism serves this need.

This grain of parallelism in searching tree nodes is coarse, suitable for multiprocessors and distributed systems.

Examining a Mapping

The next task is to examine a single proposed mapping and propagate the constraints of that mapping. The first task is to enforce the minimal constraints — the focus vertex may map to no large vertex other than the image vertex, and no other small vertex may map to the image vertex. Next, we check to see if the incomplete isomorphism is a leaf in the search. If so, we report the isomorphism,[6] otherwise we apply better constraints.

[4] Iterators are a limited form of control abstraction intended to support data abstraction. With iterators, the user of an abstraction can apply an operation to all the elements of an abstract data type without knowing the representation of the type.

[5] Generators are data operations that generate a sequence of elements on demand. Programmers often call generators from within loops to implement iteration. But, the generator is not an iterator and is not a control abstraction.

[6] The constraint filters are not complete. They may leave some invalid isomorphisms at the leaves of the search tree. A separate check will eliminate these before they are reported.

```
method examine postulate: integer
{ image := postulate;
  self.minimal_constraints![];
  focus = maximum_small .ifelse!
  [ then: closure { self.report_possible_isomorphism![]; };
    Else: closure { self.constrain![ ]; };
  ];
};
```

We use two non-trivial constraints, vertex connectivity and vertex distance, to filter possible mappings. Because these filters only remove elements from the sets of possible mappings, we may execute them in parallel, which requires atomic element removal. The filters may leave some map sets empty, in which case no isomorphism is possible for that node. If we have a valid node, we can choose the next vertex, and search its possibilities.

```
method constrain
{ cobegin.new![].two!
  [ a: closure { self.distance_filter![ ]; };
    b: closure { self.connect_filter![ ]; };
  ];
  self.no_empty_mapping![].if! closure
  { focus :=+ 1;
    self.search![ ];
  };
};
```

At most, this routine offers two-way parallelism. This parallelism is usually not enough, alone, to exploit modern multiprocessors effectively. However, it can supplement other forms of parallelism by doubling the number of processes, which often increases the ability of execution systems to balance computational load. Because both filters change the node, a shared-memory architecture is likely to be more effective. On the other hand, when the filters execute sequentially, the second filter need not examine mappings removed by the first filter, which reduces the amount of computation. The programmer must decide when exploiting parallelism here is appropriate and when it is not.

Distance Filter

Two small vertices separated by a distance x cannot map to two large vertices separated by a distance $y > x$.[7] We rely on two precomputed arrays, `small_distance` and `large_distance`, to retrieve distance information. Using the current focus vertex and its postulated image as one vertex of each pair, we successively choose each small vertex as the second small vertex and remove those possible mappings with an inconsistent distance. Using `forall_elements` the operation is:

[7] Two small vertices can map to large vertices separated by a distance $y < x$ because the isomorphism may ignore edges in the large graph that shorten the distance.

```
method distance_filter
{ range.new![ from: 0; to: maximum_small-1; ].forall!
  closure other_small: integer
  { possible#other_small,.forall_elements! closure other_large: integer
    { small_distance#focus,#other_small,
            < (large_distance#image,#other_large,@) .if!
      closure { possible#other_small.remove_element!other_large; };
    };
  };
};
```

As in the search operation, the if condition quickly ends many potential tasks. Because we cannot evaluate the condition in terms of the members of the set alone, we cannot adopt the earlier solution and fold the test into a simple iterator. However, we can define a *conditional iterator.* Conditional iterators accept a condition to test elements as well as the work to perform on each element if it passes the test. This approach enables us to evaluate the conditions sequentially, avoiding the overhead of a parallel task for quick computations, and then create a parallel task for each element that passes the test. The conditional iterator for integer sets is:

```
method forall_elems_cond
[[ test: port‘integer boolean’; work: port‘integer’; ]]
replies empty;
```

```
↓ members.forall_elems_cond!
  [ test: ...; work: ...; ] → ↓ test!i                  [i : i ∈ members]
↑ test!i → ↓ work!i                           [i : i ∈ members ∧ test!i]
↑ work!i → ↑ members.forall_elems_cond!
            [ test: ...; work: ...; ]         [i : i ∈ members ∧ test!i]
↑ test!i → ↑ members.forall_elems_cond!
            [ test: ...; work: ...; ]       [i : i ∈ members ∧ ¬ test!i]
```

This definition leaves room for several different implementations.

Given forall_elems_cond, the distance filter becomes:

```
method distance_filter
{ range.new![ from: 0; to: maximum_small; ].forall!
  closure other_small: integer
  { possible#focus,.forall_elems_cond!
    [ test: closure other_large: integer
            { reply small_distance#focus,#other_small,
                         < large_distance#image,#other_large,@; };
      work: closure other_large: integer
            { self#other_small,.remove_element!other_large; };
    ];
  };
};
```

The conditional iterator is strictly more expressive than a simple iterator. (A constant *true* condition yields the semantics of the simple iterator.) The implementation of the conditional iterator can exploit parallelism in the work, and not among conditions, which was not possible with the simple iterator.

Note that in the code above, the body of the `forall_elems_cond` acts only on the set used in the `forall_elems_cond`. We ask the `forall_elems_cond` to create potential parallelism, then ask `remove_element` to synchronize so that element removal is atomic. We can eliminate this inconsistency by recognizing that we are removing elements that meet a condition, and use an operation representing exactly that action. We define a `remove_elements_cond` operation that for each element of the set asks a closure if it should remove the element.

```
method remove_element_cond test: port‘integer boolean’ replies empty;
```

$\downarrow$ `members.remove_element_cond!test` $\rightarrow$ $\downarrow$ `test!`i $\qquad [i : i \in$ `members` $]$

$\uparrow$ `test!`i $\rightarrow$ $\uparrow$ `members.remove_element_cond!test` $\qquad [i : i \in$ `members` $]$

Its implementation must synchronize with other operations on the set.

Our final version of `distance_filter` expresses our intent precisely, while leaving much latitude in the possible implementations of `remove_element_cond`.

```
method distance_filter
{ range.new![ from: 0; to: maximum_small-1; ].forall!
   closure other_small: integer
   { possible#other_small,.remove_element_cond!
      closure other_large: integer
      { reply small_distance#focus,#other_small,
                 < (large_distance#image,#other_large,@);
      };
   };
};
```

The potential sources of parallelism are in the `forall` (medium grain), and in `forall_elems_cond` or `remove_element_cond` (fine grain). The former is appropriate to shared memory multiprocessors and the latter is appropriate to vector and SIMD machines.

Connectivity Filter

Given a postulated mapping, the neighbors of the focus vertex can only map to neighbors of the image vertex. Again, we can use `forall_elements` in iterating over the neighbors. We can also remove possible mappings for the neighbors in parallel. The resulting mapping is the intersection of the possible mappings and the neighbors of the current large vertex. With the benefit of experience gained above, we can move directly to an operation for set intersection and assignment.

```
method connect_filter [[ at: integer; for: integer; ]]
{ small_neighbors#,focus,.forall_elements!
  closure other_small: integer
  { node#other_small,.assign_intersection!
        (large_neighbors#image,@);
  };
};
```

We can leave the degree of exploited parallelism to the implementation of the set intersection. Given an appropriate implementation of sets, vector instructions can implement the intersection. A second potential source of parallelism, appropriate for shared memory multiprocessors, arises in `forall_elements`.

Control abstraction is a powerful tool for defining representation-independent operations on data. For instance, we can implement `assign_intersection` with `remove_element_cond`.

```
method assign_intersection others: set‘integer’
{ self.remove_element_cond!
  closure member: integer
  { reply others.element_of_set!member~;
  };
};
```

Given such a tool for defining operations, we may be tempted to define data abstractions that provide minimal sets of operations and rely on general control abstraction to implement more extensive data operations. Unfortunately, when we rely on general control abstraction to implement data operations, we lose the ability to take advantage of the representation of data in exploiting parallelism. For example, it is difficult to derive an implementation of set intersection based on *and*ing bit strings from the above definition of `assign_intersection`. If data abstractions export a wide variety of operations, programmers of implementations of these abstractions can improve performance by taking advantage of the representation.

Control abstraction encourages data representation-independent programming, which *users* of abstractions desire for architectural adaptability. *Designers* of abstractions must be careful to include many operations, so that *implementors* of abstractions can take significant advantage of the representation.

We identified several sources of parallelism in our algorithm. They are appropriate to distributed, multiprocessor, and uniprocessor machines. Programmers need only choose the appropriate annotation when adapting the program to a given machine. For example:

the implementation of operation	may annotate the invocation of	with any of the annotations
`search`	`forall_elements`	`_SEQUENTIAL _GROUPED _DIVIDED`
`constrain`	`cobegin`	`_SEQUENTIAL _PARALLEL`
`distance_filter`	`forall`	`_SEQUENTIAL _GROUPED _DIVIDED`
	`remove_elem_cond`	`_SEQUENTIAL _VECTOR`
`connect_filter`	`forall_elements`	`_SEQUENTIAL _GROUPED _DIVIDED`
	`assign_intersection`	`_SEQUENTIAL _VECTOR`

Selecting combinations of these annotations provides us with 216 possible implementations of subgraph isomorphism. We expect a dozen to make sense for current machines. This example uses iterators, conditional iterators, and conditional data operations, to show how data and control abstraction interact to provide powerful mechanisms for representing and exploiting parallelism.

6 Programming Method

There never has been, nor will there ever be, any programming language in which it is the least bit difficult to write bad code.
— Lawrence Flon

Abstraction reduces the cost of any program changes that may arise while debugging, porting, and enhancing programs. This chapter presents a method for using control abstraction in parallel programs to achieve architectural adaptability. Programmers are generally aware of the benefits and costs of *data* abstraction, but not of *control* abstraction. Just as the introduction of data abstraction requires a change in programming method, so does the introduction of control abstraction.

6.1 Abstract Early and Often

Abstracting early is a good principle in sequential programming because it delays commitment [Thimbleby, 1988], which localizes the program's assumptions and reduces the effort needed to change a program. However, when programming sequentially, we often do not use abstractions because there is a simple, natural, and obvious best implementation. The best implementation is usually obvious because most sequential machines share the same von Neumann 'type architecture'. In contrast, there are several common type architectures [Snyder, 1986] for parallel machines. The performance of a given exploitation of parallelism may vary widely among these type architectures. Abstraction helps adapt programs among different type architectures. Parallel programmers should resist implementing prematurely, and rely on data and control abstraction.

In developing a program using control abstraction, the programmer needs to identify the places where the algorithm organizes and schedules 'units of work'. The programmer should encapsulate each of these 'organize and schedule' activities in a control construct. For instance, a key control abstraction in Gaussian elimination is "select the pivot and reduction equations". Its corresponding unit of work is "reduce an equation". So, we should explicitly represent the "select" control abstraction with a control construct.

Adaptable programs will use control abstractions that minimize the restrictions on statement sequencing (*i.e.* maximize potential parallelism). This leaves maximum freedom to choose an efficient implementation.

Likewise, an equation in Gaussian Elimination is a useful data abstraction and "reduce an equation" is an abstract operation on equations. The equation abstraction gives us another potential site for communication. When appropriate, we can communicate from one processor to another at the invocation of "reduce an equation", rather than at lower levels in the code. More data abstractions give us more potential communication.

Where appropriate, programmers should use data abstractions that provide control abstractions to manipulate the data. For example, we should program in terms of sets (data abstraction) and parallel iteration over sets (control abstraction), rather than bit vectors (data representation) and parallel scanning of bit vectors (representation-dependent control). The resultant program will be both easier to understand and easier to adapt to other architectures.

6.2 Use Precise Control Constructs

When the control constructs we use to specify parallelism do not *precisely* express the parallelism appropriate to an algorithm, we must introduce explicit synchronization to restrict excessive parallelism or we must accept less parallelism than the algorithm permits. In this section, we show how control abstraction can enable simultaneous selection of parallelism and control synchronization, as well as accommodate data dependence.

6.2.1 Embed Synchronization

The presence of parallelism in a program generally implies the presence of synchronization. When we introduce parallelism, we must also introduce synchronization. Ideally, we select synchronization with the same mechanism that exploits parallelism. Parallel programs exhibit two types of synchronization: *data synchronization* ensures consistent access to data by independent threads of control; *control synchronization* coordinates between threads created to work together in parallel. In particular, synchronization that supports a data dependence is control synchronization. As in Multilisp [Halstead, 1985], we assume that data synchronization is embedded in data abstractions.

Explicit synchronization needed to restrict excessive parallelism must be inserted or removed depending on the choices made to exploit parallelism. This process can be error-prone and can make adapting programs to different architectures difficult. Therefore, when explicit synchronization is needed to implement control synchronization, it should appear only in the implementation of control constructs, and never in the body of work passed to a control construct. If, in the development of a program, it becomes necessary to introduce synchronization into the body of work, the control construct should be redesigned to embed the synchronization.

When using a control construct that provides more synchronization or serialization than needed, we abandon potential parallelism. Constructs that maximize potential parallelism leave more room for exploitation of parallelism and enhance our ability to adapt to new architectures. We should use control constructs that provide the maximum potential parallelism allowed by the algorithm.

We should choose control constructs that express precisely the parallelism and synchronization that the algorithm requires, neither more nor less. When selecting an exploitation of parallelism in such a construct, we implicitly select the appropriate synchronization. If an implementation of a construct exploits no parallelism, it needs no synchronization, and need not pay the overhead.

Embedding synchronization in a construct limits its applicability, so we must be careful to select a construct appropriate to the problem at hand. When choosing an existing control construct that does not provide the necessary synchronization in its implementation, we must insert explicit synchronization into the work to be performed. Unfortunately, this commits us to a specific exploitation of parallelism that cannot be changed with an annotation. The resulting program is more difficult to tune or port. Rather than use an inappropriate control construct and additional explicit synchronization, the preferred approach is to build a new construct that encapsulates the correct synchronization.

Gaussian Elimination in the Uniform System

An interesting example of the use of a control construct with insufficient synchronization arose in previous work with Gaussian elimination. An early version of the program developed at BBN [Thomas, 1985] used the Uniform System parallel programming library [Thomas, 1986]. The Uniform System provides a globally shared memory and a set of predefined task generators. Each generator accepts a pointer to a procedure and executes the procedure in parallel for each value produced by the generator. Thus, generators are a limited form of control abstraction. The Uniform System provides generators for manipulating arrays and matrices, including `GenOnHalfArray`, which generates the indices for the lower triangular portion of a matrix. The Uniform System implementation of Gaussian elimination used this generator. It's Natasha representation and partial orders are:

```
pair: [[ index1: integer; index2: integer; ]];

triangulate: object size: integer;
{ method GenOnHalfArray work: port'pair empty' replies empty;
};

t: triangulate.new!SIZE;

↓ t.GenOnHalfArray!work
  → ↓ work![ index1: i; index2: j ]                  [i, j : 1 ≤ i < j ≤ size]
↑ work![ index1: i; index2: j ]
  → ↑ t.GenOnHalfArray!work                          [i, j : 1 ≤ i < j ≤ size]
```

This generator provides the parallelism of our `forpairs` construct, but without the synchronization constraints. As a result, the Uniform System program included explicit synchronization within the body of the work.[1] Gaussian elimination using `GenOnHalfArray` looks like this:

[1]The actual program used more efficient synchronization than is shown here, but this version accurately represents the control flow and is consistent with our earlier examples.

```
system: array‘SIZE array‘SIZE float’’;
pivot_done: array‘SIZE condition’;
element_done: array‘SIZE array‘SIZE condition’’;
pivot_done#1,.signal![];
trianglulate.new!SIZE.GenOnHalfArray_DIVIDED!
closure reduce: integer
{ pivot_done#pivot,.wait![];
  pivot > 1 .if! closure
  { element_done#reduce,#(pivot-1),.wait![];
  };
  fraction: system#reduce,#pivot, / (system#pivot,#pivot,@);
  range.new![ from: pivot; to: SIZE-1; ].forall_DIVIDED!
  closure variable: integer
  { system#reduce,#variable,
             :=- (system#pivot,#variable, * fraction);
  };
  element_done#reduce,#pivot.signal![];
  ( pivot = (reduce-1)).if! closure
  { pivot_done#reduce,.signal![];
  };
};
```

This implementation uses explicit synchronization to provide the serialization implicit in the `sequfor` loop in `forpairs_SYNCHED`. (See section 5.1.4.) Given the limited facilities for creating new generators in the Uniform System, and the existence of `GenOnHalfArray`, this implementation was a reasonable one. Nevertheless, a more efficient implementation would have been possible had the correct control construct been available or easily created. With control abstraction, we can build constructs that contain the necessary synchronization.

6.2.2 Explicit Versus Implicit Synchronization

In the implementation of a control construct, we often have a choice between relying on the synchronization implicit in other control constructs or using explicit synchronization. There is no single resolution of this choice for all cases. For example, the synchronization implicit in the outer loop of our phased implementation of Gaussian upper triangulation unnecessarily limits the amount of parallelism in the program. On the other hand, some of the explicit synchronization used in the Uniform System program is both expensive and unnecessary. The `forpairs_SYNCHED` implementation is a balanced combination of explicit and implicit synchronization. It uses explicit synchronization to remove the limit on parallelism imposed by the phased implementation. It also uses a `sequfor` loop to serialize the application of pivots to a single equation, in place of explicit synchronization in the Uniform System program.

6.2.3 Expose Data Dependence

In Gaussian elimination, we were able to concentrate solely on the partial order rules to derive a new control construct and embed synchronization within the construct (section 5.1.4). We may not always be able to do so. Occasionally, the natural expression of control and its work places a data dependence deep within the body of a loop, rather than at the beginning or end. For example, consider a sequential loop of the form:

```
range.new![ from: 1; to: N; ].sequfor!
closure i: integer
{ statement list 1; a#(i+1), := (a#i,@); statement list 2;
};
```

This loop has a loop-carried data dependence between iteration i and iteration $i+1$. We cannot use `forall` to specify parallelism because we would violate the dependence. One possible approach is to insert explicit synchronization around the statements containing the data dependence. Unfortunately, the presence of synchronization within the body of the loop would then be separate from the implementation of the loop, which is where we choose whether to exploit parallelism. If we follow our previous advice and avoid explicit synchronization, this dependence forces us to choose a control construct that provides more synchronization than the algorithm actually requires. The solution to this dilemma is to break the loop body into separate bodies, exposing the data dependence, and then use a control construct that handles the multiple bodies.

We create a construct that accepts the loop in three pieces, corresponding to the statements that can execute in parallel before and after the data dependence, and the statements containing the data dependence. This more complex construct is also more precise, which gives us more flexibility in exploiting parallelism.

```
method forall3 [[ head: port‘integer empty’;
                  body: port‘integer empty’;
                  tail: port‘integer empty’; ]]
replies empty;

rng: range.new![ from: lower; to: upper; ];

↓ rng.forall3![ head: ...; body: ...; tail: ...; ]
 → ↓ head!i                                          [i : lower ≤ i ≤ head]
↑ head!i → ↓ body!i                                  [i : lower ≤ i ≤ head]
↑ body!i → ↓ tail!i                                  [i : lower ≤ i ≤ head]
↑ body!i → ↓ body!(i + 1)                            [i : lower ≤ i < head]
↑ tail!i →
↑ rng.forall3![ head: ...; body: ...; tail: ...; ]   [i : lower ≤ i ≤ head]
```

The implementation must execute $\texttt{head}_i$ before $\texttt{body}_i$ before $\texttt{tail}_i$ and execute $\texttt{body}_i$ before $\texttt{body}_{i+1}$. Using this control abstraction, we can rewrite the original loop as follows:

```
range.new![ from: 1; to: N; ].forall3!
[ head: closure i: integer { statement list 1; };
  body: closure i: integer { a#(i+1), := (a#i,@); };
  tail: closure i: integer { statement list 2; };
];
```

This control construct admits a parallel implementation wherein the `head`'s and `tail`'s all execute in parallel.

```
method forall3_DIVIDED
[[ head: integer; body: integer; tail: integer; ]]
{ size: (from+1)-to+1;
  blocking: array`size semaphore';
  blocking#lower,.signal![];
  self.forall_DIVIDED!
  closure i: integer
  { head!i; blocking#i,.wait![];
    body!i; blocking#(i+1),.signal![];
    tail!i;
  };
};
```

An alternative implementation that avoids the use of explicit synchronization and results in a slightly different parallelization is as follows:

```
method forall3_PHASED
[[ head: integer; body: integer; tail: integer; ]]
{ self.forall_DIVIDED!head;
  self.sequfor!body;        ;; always sequential
  self.forall_DIVIDED!tail;
};
```

The `_DIVIDED` implementation avoids phases and admits more parallelism, but because it uses blocking synchronization primitives, may be less efficient. The programmer can decide if the benefit of the extra parallelism is worth its cost.

We must balance the programming cost of splitting bodies of work against the likely possible architectures that may exploit the newly exposed parallelism. This balance depends on the synchronization constraints within the construct and the likely number and size of units of work for the construct — small units of work with complex synchronization constraints are unlikely to have efficient implementations on current architectures. This observation applies to Gaussian elimination. In particular, synchronization constraints between pivot and reduction terms (rather than equations) are possible, but synchronizing each multiplication and subtraction pair introduces unacceptable overhead on most current architectures.

6.3 Reuse Code

Parallel programming is hard, so programmers should build on each other's work where possible. A library of well-debugged data and control abstractions is the programmer's most effective productivity tool. If the programmer needs a reasonably common control construct, it may appear in a library of constructs and their implementations. However, some control constructs will be algorithm-specific; no library will contain implementations for those constructs. The programmer must design and implement the construct. However, the programmer need only code implementations *as needed* for the *architecture at hand*, and need not code implementations for all architectures or possible exploitations of parallelism. The set of implementations will expand during program tuning and porting. Each implementation remains available for use later. In contrast, without control abstraction programmers tend to abandon previous exploitations of parallelism in the search for the best exploitation for a given architecture. A program's investment in architectural adaptability is primarily in the constructs it uses, and secondarily in the set of implementations for those constructs. Changing a construct is a serious undertaking; using another implementation of a construct is not.

Because the ability of a programmer to tune the program depends on the availability of several implementations for many control abstractions, the presence of a library of general-purpose control abstractions will directly affect the viability of a programming language based on the Matroshka model.

6.4 Experiment with Annotations

After developing a program using control abstraction, the programmer must annotate each use of a control construct with the desired implementation. Initially, programmers simply make their best guesses, or leave the choice to defaults or the compiler. Later, programmers must refine their annotations. In sequential programming, the code sections critical to performance, and the effect of optimizations on them, may not be at all obvious, and are often counter-intuitive [Bentley, 1982]. The critical code sections are even more unpredictable in parallel programming. Experimental methods and program analysis tools [Fowler *et al.*, 1988; Mellor-Crummey, 1989; LeBlanc *et al.*, 1990] will help parallel programmers determine the most efficient exploitation of parallelism. When poor performance relates back to a control construct, the programmer can easily choose an alternate implementation (using more or less parallelism and synchronization) by changing the annotation to select an alternate implementation. The programmer may then measure the effect of the new annotation on program performance.

7 Natasha Implementation

For which of you, intending to build a tower, sitteth not down first, and counteth the cost, whether he have sufficient to finish it?
— The Bible, *St. Luke*

We showed the importance of abstraction in parallel programming, and how to exploit different grains of parallelism by selecting an appropriate implementation for each abstraction. Although descriptive power is an important property, programmers use parallelism to improve performance. Any programming language that uses closures and operation invocation to implement the most basic control mechanisms might appear to sacrifice performance for expressibility. With an appropriate combination of language and compiler, however, user-defined control constructs can be as efficient as language-defined constructs. We support this claim with an implementation and its analysis. First, we describe our implementation of Natasha. Second, we describe straightforward, locally applicable optimizations that reduce the execution cost of Natasha mechanisms. Third, we compare the execution speed of Natasha programs against equivalent C programs.

7.1 Compiler and Library Organization

The Natasha implementation is composed of a compiler generating C code for the GNU C compiler [Stallman, 1989], and a run-time library. The Natasha compiler takes advantage of several GNU extensions to the C language. The run-time library assumes a shared-memory environment, but is otherwise mostly machine independent. The machine dependent parts include support for the Sun-3 under the Unix operating system, the BBN Butterfly Parallel Processor [BBN, 1985a] under Chrysalis [BBN, 1985b], the Butterfly Plus [BBN, 1987] under Platinum [Cox and Fowler, 1989], the Butterfly Plus under Psyche [Scott *et al.*, 1990], and the Alliant FX [Alliant, 1987] under Concentrix (a Unix derivative) [Alliant, 1989]. An efficient implementation of Natasha requires cooperation between the compiler, runtime library, and host operating system.

The Natasha compiler has the following passes:

Preprocess: The compiler runs all programs through the C preprocessor first. This feature enables programmers to avoid editing program source when changing annotations.

Parse: The parse phase builds a parse tree. It uses the LEX scanner generator and the yacc parser generator. The parse rules do l-value and r-value interpretation (see section 3.3.8) and recognize late replies (see section 7.2.1).

Bind: The bind phase builds a semantic tree from the parse tree by binding types to parse tree elements. This phase builds the symbol table and checks types for consistency. This phase does port in-lining and closure in-lining for predefined control operations by examining the syntax tree for known patterns.

Generate: The code generation phase selects and writes GNU C source as an intermediate program that implements the Natasha program. Both the bind and generate phases are written in C++.

Compile: The Natasha compiler invokes the GNU C compiler (version 1.35) on the intermediate program and links in the Natasha run-time library.

The machine-independent part of the run-time library implements the predefined types, task invocation, task scheduling, processor management, and program initialization. The machine-dependent part of the run-time library implements atomic locks and counters, coroutine switch, system memory allocation, terminal input/output, process creation (one per processor), and address space initialization.

7.2 Optimizing Natasha Mechanisms

This section presents several optimization techniques that are well within the bounds of current compiler technology. The Natasha compiler uses most of them, though not always in a general way. We expect a production compiler for other languages based on Matroshka to use nearly all these optimizations.

7.2.1 Operation Invocation

The presence of an object model need not imply poor execution performance. While Natasha programs can express inherently expensive operations, operations rarely use their full generality. By implementing each invocation with only the generality necessary to execute properly, we can substantially reduce the cost of operation invocation. This section describes several optimizations that successively cast away more generality and thereby improve the execution performance of Natasha programs. With these optimizations, operation invocation can be as efficient as code written in-line in conventional algorithmic languages. The approach relies on providing several different implementations for operation invocation, each appropriate to a different class of operation.

Invocations as Procedure Calls: Since an invocation may execute concurrently with its invoker after executing its reply, a conservative implementation of invocation provides a separate thread of control for each invocation. This approach is prohibitively expensive. We can reduce this cost by noting that operations that have no statements after the reply have no opportunity for parallelism and have a partial order identical to regular procedures. We can therefore implement these operations as regular procedures. Even though invocations are frequent, the vast majority have valid implementations as procedure calls.

Delayed Replies: In those cases where an operation replies early, it is often safe to delay the reply until the invocation completes. This delay allows us to exploit the procedure implementation once again. We can safely delay a reply if no statement following the reply requires resources (such as synchronization variables) that statements following the invocation release. This situation is common and is the case in all our examples. We cannot expect the compiler to always determine whether to delay a reply, so we use two different forms of reply. One indicates that the compiler may delay an early reply, and the other indicates that the compiler may not.[1]

In-line Ports: The use of Natasha ports hides the invoked object and the exact method from the invoker. While this aids program and communication independence, it means that simple implementations must call method procedures indirectly. We eliminate most of this cost by noting that programs invoke most ports immediately after forming them. By eliding the port formation, the compiler can generate direct calls to the method procedures. This optimization provides little benefit by itself, however, it is a necessary step in achieving the following optimization.

In-line Methods: Even if we are able to avoid creating a new thread of control for each operation invocation, we may still pay the price of a procedure call for each invocation. We can reduce overhead even further by statically identifying the implementation of operations, which makes it possible to use in-line substitution. Natasha identifies implementations through static typing; other languages could identify implementations through type analysis.

7.2.2 Task Execution

Eliminating tasks will substantially reduce sequential execution time, but leaves no room for parallelism. We must, at some point, pay the price for task creation so that parts of the program may execute in parallel. The Matroshka model encourages a large number of short-lived parallel tasks, so task efficiency is important. The Natasha compiler does several optimizations that improve the performance of tasks.

Migrate Tasks: The primary task optimization is to execute the task on the processor that contains the corresponding object. This means that interprocessor communication occurs only at invocation and reply.

Eliding Local Tasks: Sometimes it is most efficient to invoke an operation as a remote task when the object is remote, and to invoke the operation as a procedure when

[1] On uniprocessors, early replies that are not delayed force a multi-tasking implementation.

the object is local. A simple test for object locality makes this operation inexpensive. Emerald [Jul *et al.*, 1988] relies heavily on this type of local/nonlocal determination.

Avoid Stack Allocation: In general, tasks may wait on other tasks or synchronization variables. To avoid wasting processors, we block these tasks and schedule another task for execution. This requires allocating a separate stack for each task. When a task will not block, because it accesses no synchronization variables and creates no tasks, we can avoid allocating a separate stack and use the scheduler's stack to execute the task. The Lynx implementation uses this technique [Scott, 1986; Scott, 1987].

This optimization is currently available only for the `_quick` implementations of range and array iterators. A general implementation of non-blocking tasks is feasible, but would require re-writing the runtime system.

Non-Preemptive Execution: The Natasha run-time library reduces the synchronization requirements within the runtime by executing tasks until they block or complete.

Last-In-First-Out Scheduling: In executing a program based on our model, we may think of a tree of parallel tasks where each reply generates a branch in the tree. Normal FIFO scheduling strategies will traverse this tree of tasks in a breadth-first manner. In a breadth-first execution, the number of active nodes grows quickly. Their representation will quickly consume the entire storage of almost any machine.

The typical solution to this problem is to use a LIFO scheduling queue [Halstead, 1990], which encourages a depth-first execution. The number of active nodes is small, with $O(\log n)$ simultaneous activations. This approach is preferable to the normal FIFO scheduling, which corresponds to a breadth-first evaluation, which has $O(n)$ simultaneous activations.

Stealing Work: When a processor has an empty scheduling queue, it takes tasks from other processors' queues. In contrast to Concert Multilisp and Mul-T [Halstead, 1990], we take tasks *least* recently enqueued rather than *most* recently enqueued. Taking the least recently enqueued tasks least disturbs of the locality of processors. (This issue is important in maintaining $O(\log n)$ activations, and so the strategy applies to UMA multiprocessors in addition to NUMA multiprocessors.)

7.2.3 Control Operations

Control abstraction has a high potential cost. This section provides several optimizations that reduce this cost. In Natasha programs that are structurally similar to programs in conventional algorithmic languages, these optimizations lead to control operations with similar costs.

In-Line Closures: In Natasha, closures are operations on activation objects. As operations, closures are amenable to all the optimizations that apply to normal operations. By doing source-level in-lining, and propagating closure parameters to their use within the definition of a control operations, compilers may substitute closures in-line. In-line substitution is especially important for the efficient execution of sequential control constructs. When the compiler can determine the implementation of a construct statically, it can replace the invocation with the implementation, and propagate the closure parameter through to its use. Using this technique, we can convert control constructs using late replies into equivalent machine branch instructions. The Natasha compiler currently does not do this optimization. It is the compiler's greatest weakness.

Stack Allocation of Closures: Closures in Smalltalk and Lisp require that their environments remain in existence for the lifetime of the closure. The standard implementation of closures uses heap allocation for all operation activations that contain closures. Since the cost of dynamic allocation can be substantial, the widespread use of closures could have severe performance implications.

There are at least three language-dependent approaches to reducing the cost of closure environments. The first is to analyze the program to determine if a closure is used after normal termination of its environment. If not, the compiler may allocate the environment on an activation stack [Kranz *et al.*, 1986]. The second approach restricts the assignment of closures, like Algol68 reference variables, such that the environment is guaranteed to exist.[2] The third approach, which we used in our implementation for expedience, defines programs that invoke a closure after its environment has terminated as erroneous.[3] Each of these approaches enables stack allocation for closures, significantly reducing the overhead associated with their use.

Direct Scheduler Access: Note that the presence of an implementation for a control construct, such as `forall`, using our mechanisms does not imply that a programming system must use the implementation. In particular, implementations of `forall` are most efficient when they can directly manipulate scheduler queues. We expect that programming systems will provide implementations of common control constructs that are integrated with the scheduler. The Natasha implementation provides special scheduling queues for `forall` constructs.

Task Generators: Control operations, such as `forall`, may often start many tasks. The obvious implementation of a sequential loop creating the appropriate number of tasks involves $O(n)$ work on setting up the loop. An alternate approach places a *task generator* on the scheduling queue. The process of grabbing a task involves generating another task from the description on the queue. This distributes the task creation overhead among several processors.

[2] Algol68 does not permit assignment of references to variables outside the local scope, thus ensuring that the lifetime of a variable containing a reference is no greater than the lifetime of the referent.

[3] As in Ada [U. S. DoD, 1983], we use the term erroneous to indicate incorrect programs that the compiler and runtime need not detect as incorrect.

7.2.4 Data Access

The Natasha prototype compiler provides no sophisticated optimizations on data access. Three optimizations that could be done are:[4]

Distribute Data: Implementations of Matroshka must provide facilities for building distributed implementations of data abstractions. Natasha provides several distributed arrays, which programmers may use to build other distributed types.

Replicate Data: High performance implementations of Matroshka should also provide mechanisms to replicate data easily. Natasha does not currently provide mechanisms to aid the replication of data, though programmers could do so manually.

Replicate Pointers to Data: Because each object (and record), once constructed, always contains the same set of objects, the implementation may implement the record as a list of pointers and copy the list freely.

7.3 Performance Evaluation

The Natasha implementation is not production-quality. It does not implement several optimizations that would be present in production implementations of Matroshka and are necessary to perform competitively with existing algorithmic languages. To show that a Natasha implementation could be competitive, we describe the weaknesses in the prototype implementation, manually apply the missing optimizations to two example programs, and compare the manually optimized programs to equivalent C programs.

7.3.1 C as an Intermediate Language

The GNU extensions to the C language eased the code generation task considerably. Even so, the C language has significant weaknesses when used as a back end. It introduces several structural inefficiencies in the generated code.

Expressions: The Natasha compiler translates Natasha expressions directly to C expressions. Because of Natasha's port model, the Natasha compiler generates many references to the variable `a` with the expression `*(&(a))`.[5] While this expression is semantically equivalent to the simplest expression, `a`, it has the side effect of forcing the variable to reside in main memory, rather than in registers. This increases the number of main memory references, which decreases the program performance considerably. Another effect of the expression `*(&(a))` is that it inhibits some forms of constant expression evaluation.

[4] The operating system can also help manage data access [Bolosky *et al.*, 1989; Cox and Fowler, 1989; Bolosky *et al.*, 1991].

[5] While it is possible to perform this optimization with a simple post-processor, the optimization would be ineffective if done in isolation. See section 7.3.2 and figure 7.1.

Record Constructors: Natasha relies on record constructors to build argument lists. The compiler uses GCC's struct constructors to represent record constructors and build ports. Unfortunately, the GCC compiler insists on building such records in main memory, rather than in registers. This also reduces performance substantially.

Nested Scopes: Natasha provides nested procedures, primarily as closures. However, C does not provide nested procedures, so the compiler cannot directly translate Natasha closures to C procedures. The Natasha compiler represents closure nesting by explicitly linking together structs containing Natasha activation variables. GCC always allocates structs in memory, so all Natasha variables will be allocated in memory even though many should be promoted to registers. This also reduces performance substantially.

Procedure In-Lining: The Natasha compiler takes advantage of GNU's C compiler `inline` pragma on procedures. This pragma instructs the compiler to splice the assembly code of the procedure into that of the caller, which removes the procedure prolog and epilog code. While effective at reducing the cost of procedure calls, assembly in-lining is not nearly as effective as source in-lining. Because of difference in performance between source and assembly in-lining, Natasha does source in-lining for predefined operations. For implementation expedience, it does not do source in-lining for user operations.

No Coroutines: The C language provides neither coroutines nor access to the stack. In building Natasha tasks, it became necessary to call an assembly procedure to implement parameter passing to coroutines and switching to coroutines. The resulting task invocation then consists of calls to

- the manager procedure (possibly in-line),
- the general invocation procedure (placing arguments on the caller's stack),
- the task allocation procedure,
- the parameter copy procedure,
- the block copy procedure (placing arguments on the callee's stack),
- the coroutine switch procedure,
- the reply procedure, and
- the task deallocation procedure.

There are at least two excess procedure calls and one parameter copy that would not be necessary if the implementation language directly supported coroutines.

7.3.2 Sequential Execution

Natasha programs take more than twice the time to execute than equivalent C programs. In an effort to determine if the inadequacy of C as an intermediate language was the primary component of the difference in execution speed, we wrote a sequential program using only control constructs that are directly represented in C. The program makes four million calls to a random number generator. We then manually edited the C code generated (after the C preprocessor) with four simple optimizations. They are:

1. convert `*(&(var))` to `var`,
2. represent Natasha activation variables as C activation variables (instead of as members of a struct within the activation),
3. convert `*((var = expr), &var)` to `expr`,[6] and
4. represent Natasha global variables as C global variables.

The resulting execution times appear in figure 7.1. These manual optimizations bring Natasha execution times to within 2% of comparable C programs. We expect that a production compiler for Natasha would be competitive with an optimizing C compiler.

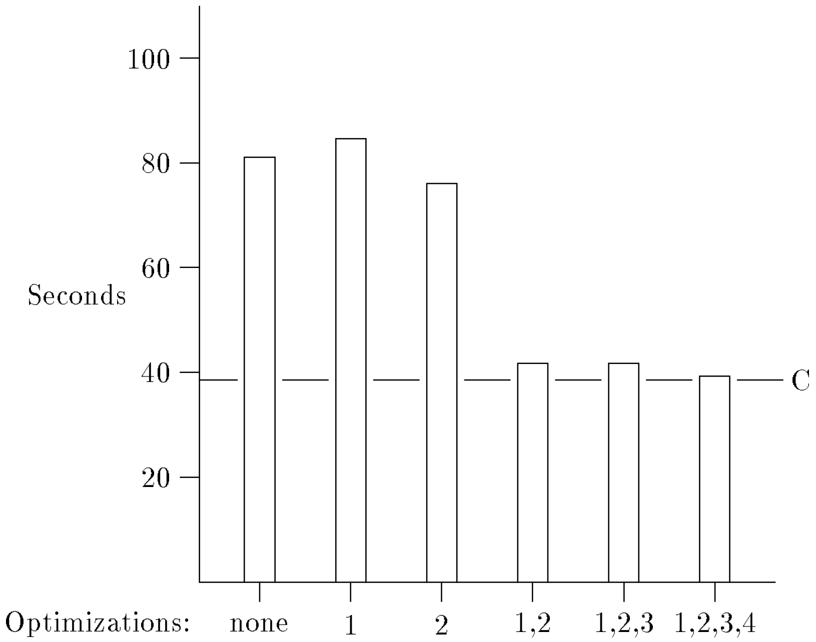

Figure 7.1: Manual Optimization of Sequential Natasha

[6]This expression is an artifact of the mismatch between C operators being define on values and Natasha operators being defined on objects.

7.3.3 Parallel Execution

Given that the sequential execution of Natasha programs can be as fast as C programs, we tested the Natasha program for Gaussian elimination against an existing hand-tuned C program [LeBlanc, 1988]. We use the same set of problem and program restrictions. Figure 7.2 shows the unoptimized Natasha program executes much slower than the C program. The first step in hand optimizing the Natasha program is to apply the opti-

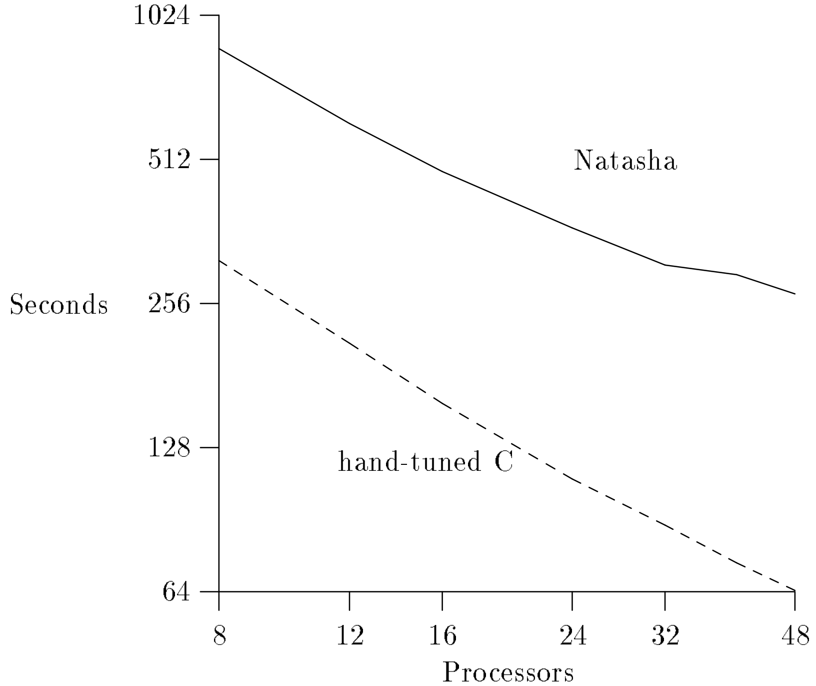

Figure 7.2: Initial Performance of Parallel Natasha

mizations of section 7.3.2 to the inner loop, then apply induction variable elimination to the inner loop. The resulting execution times appear in figure 7.3. In this and future figures, the dotted lines show the original Natasha and hand-tuned C execution times. These optimizations help Natasha considerably for fewer processors, but provide no aid for many processors. The Natasha program is communication limited for large numbers of processors. The prototype implementation introduces two unnecessary copies of the pivot row, one copying from the parameter to the activation record, and the second in constructing the argument list to the reduction. Removing these redundant copies results in the times shown in figure 7.4. Two optimizations remain. The first in-lines closures used in the `forpairs` control construct. The second copies only the non-zero portion of the pivot row. This latter optimization requires a language that can manipulate sub-arrays. Figure 7.5 shows the resulting times. We expect production implementations of Natasha to use all these optimizations and therefore perform competitively with hand-tuned C programs. We do not expect Natasha programs to outperform hand-tuned C programs. The Matroshka model will provide a performance advantage only indirectly by making the tuning of parallel programs easier.

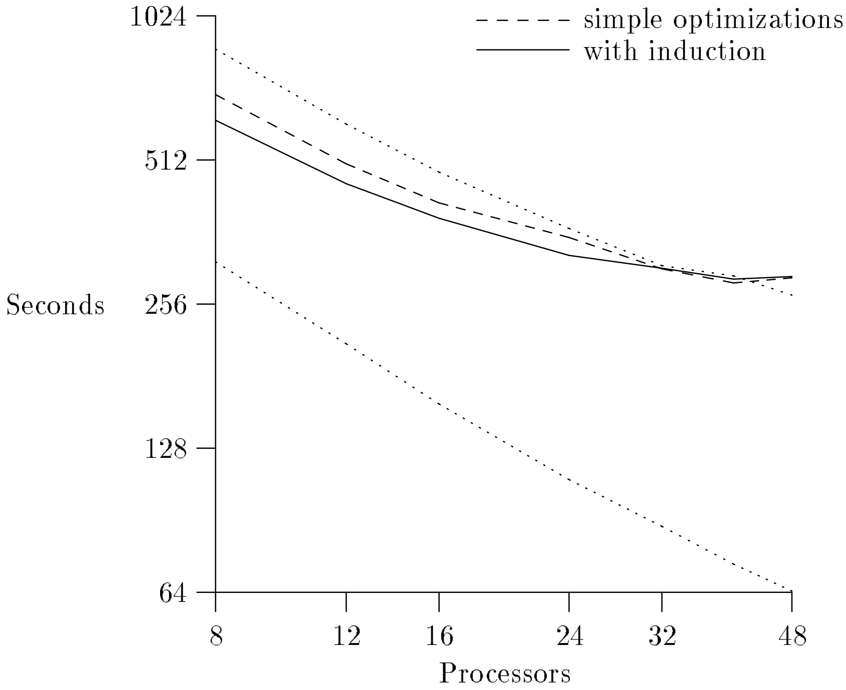

Figure 7.3: Performance With Inner-Loop Optimization

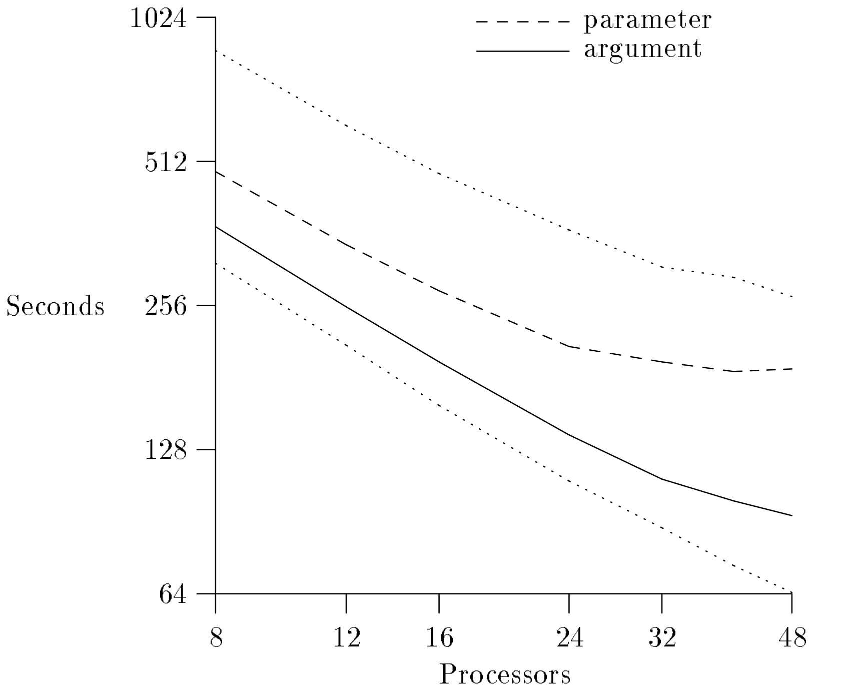

Figure 7.4: Performance Without Redundant Copies

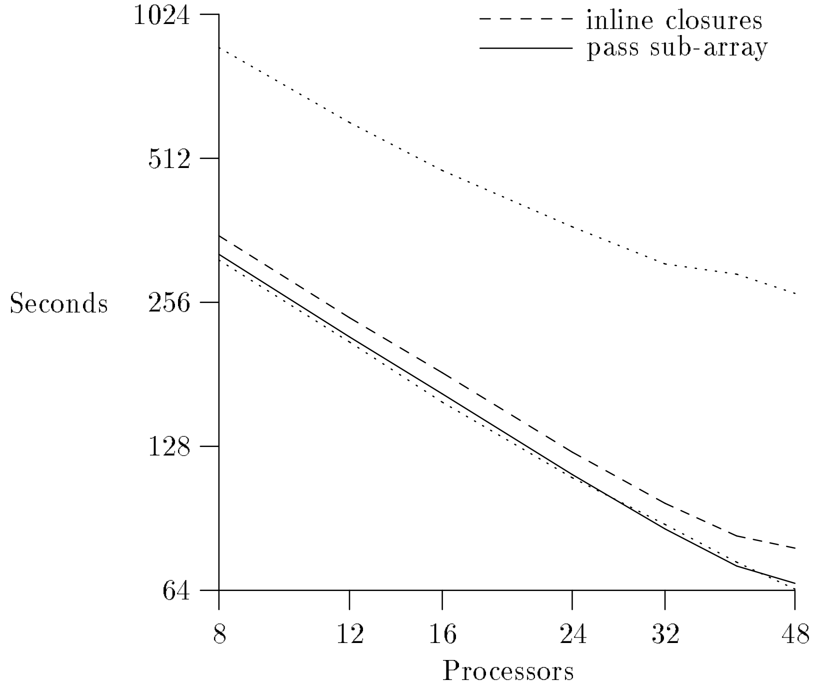

Figure 7.5: Performance With Final Optimizations

8 Conclusions

Though Patience be a tired mare, yet she will plod. There must be conclusions.
— William Shakespeare, Henry V*, 1600*

To implement a parallel algorithm, programmers must identify and exploit parallelism, distribute data and processing, and choose communication. The appropriate implementation of an algorithm depends heavily on the target architecture. Since parallel architectures vary widely, implementations of an algorithm will also vary widely.

Architectural adaptability is the ease with which programmers can tune or port a program to a different architecture. When programmers embed assumptions about the architecture in the text of programs, the programs are difficult to adapt. We can avoid unwanted assumptions in programs by separating the specification of an algorithm from its realization on a given architecture. This separation of use from implementation is an instance of abstraction. Early work in parallel programming languages provided a set of predefined abstractions that provided the separation of specification from realization. Unfortunately, early languages provided no mechanism for the user to continue the process of abstraction to represent an application-specific separation. Recent work emphasizes the use of data abstraction to separate distribution from the rest of the program.

8.1 Contributions

We extend the use of abstraction in parallel programming for architectural adaptability to control abstraction. Control abstraction provides several benefits in the construction of programs that adapt well to different architectures.

- With control abstraction, programmers are not limited to a fixed set of control constructs. Users can create new constructs that express arbitrary partial orders of invocations and store them in a library for use by others. We presented a model of parallel programming based on a small set of primitive mechanisms for control abstraction and showed how the model can directly implement common parallel control constructs. With the ability to define algorithm-specific control constructs, we can more precisely represent the potential parallelism within an algorithm.

- Each control construct can have multiple implementations, each of which exploits a different subset of the potential parallelism defined by the construct. Selecting different implementations of a construct at the points of use exploits different sources of parallelism within a program. By embedding synchronization in the implementation of control constructs, separate from the program logic, programmers select parallelism and synchronization simultaneously.

- When combining data abstraction with control abstraction, distributed data structures can co-locate processing with the corresponding data. In this way, programmers select distribution once when selecting the data implementation, rather than twice — with the data structure and separately with the control structure.

- By associating communication with operation invocation, each operation invocation is a source of potential interprocessor communication. By selecting among different implementations, we realize interprocessor communication at the appropriate points in the program.

We presented a method for using abstraction to achieve architectural adaptability in explicitly parallel programs. In developing adaptable programs, programmers must specify the algorithm in terms of data and control abstractions. Programmers adapt parallel programs by selecting an implementation for each use of an abstraction, and then experimentally measuring the effect. Programmers can choose among existing implementations of an abstraction or build new implementations as needed. The set of implementations will expand during program tuning and porting, leaving different exploitations documented within the source.

This dissertation shows that control abstraction is an effective means for achieving architectural adaptability in explicitly parallel imperative programs. In particular, control abstraction is the fundamental form of abstraction needed for architectural adaptability. In addition, we presented several optimizations that provide an efficient implementation of data and control abstraction. Based on our experience, we believe the benefits and reasonable cost of control abstraction argue for its inclusion in explicitly parallel programming languages.

8.2 Future Work

While the benefits of control abstraction for architectural adaptability are clear, concerns of applicability to full production environments remain. Future work in this area must address these concerns.

The prototype implementation described served adequately for this dissertation. However, production use of control abstraction for architectural adaptability will require production implementations of Matroshka. Issues remaining include efficient task management, data replication, and data distribution.

The cost of programming with control abstraction is not clear. To gain further understanding of the costs and benefits involved in the heavy use of control abstraction in parallel programming, we need to experiment with full production applications.

The generic and polymorphic type support needed for control abstraction appears to be greater than that for data abstraction alone. The extent of the support needed is not clear.

The programming method presented in this dissertation relies on a library of data and control abstraction. We do not yet know how to build comprehensive libraries of data and control abstractions.

These open issues concern programming-in-the-large. As such, the effort necessary to address them properly is significant.

Bibliography

[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.

[Albert *et al.*, 1988] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr., "Compiling Fortran 8x Array Features for the Connection Machine Computer System," In *Proceedings of the ACM SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages and Systems*, pages 42–56, July 1988, reprinted in ACM SIGPLAN Notices 23(9), September 1988.

[Allen *et al.*, 1987] Randy Allen, David Callahan, and Ken Kennedy, "Automatic Decomposition of Scientific Programs for Parallel Execution," In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[Alliant, 1987] Alliant Computer Systems Corporation, One Monarch Drive, Littleton, Massachusetts 01460, *ALLIANT FX/Series Product Summary*, June 1987.

[Alliant, 1989] Alliant Computer Systems Corporation, One Monarch Drive, Littleton, Massachusetts 01460, *Concentrix System Reference: System Calls and C Library Routines*, February 1989.

[Alverson, 1990] Gail A. Alverson, "Abstraction for Effectively Portable Shared Memory Parallel Programs," Technical Report 90-10-09, Department of Computer Science, University of Washington, October 1990, Ph.D. Dissertation.

[Alverson and Notkin, 1991] Gail A. Alverson and David Notkin, "Abstracting Data-Representation and Partition-Scheduling in Parallel Progams," In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 138–151, Tokyo, Japan, April 1991.

[Andrews *et al.*, 1988] Gregory R. Andrews, Ronald A. Olsson, Michael H. Coffin, Irving J. P. Elshoff, Kelvin Nilsen, Titus Purdin, and G. Townsend, "An Overview of the SR Language and Implementation," *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[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.

[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.

[BBN, 1987] BBN Laboratories, Cambridge, Massachusetts, *Inside the Butterfly Plus*, October 1987.

[Bentley, 1982] Jon Louis Bentley, *Writing Efficient Programs*, Software Series. Prentice-Hall Inc., 1982.

[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.

[Bolosky *et al.*, 1989] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott, "Simple But Effective Techniques for NUMA Memory Management," In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona, December 1989, reprinted in Operating Systems Review 23(5), December 1989.

[Bolosky *et al.*, 1991] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox, "NUMA Policies and Their Relation to Memory Architecture," In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, California, April 1991.

[Brinch Hansen, 1975] Per Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, SE-1:199–207, June 1975.

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

[Budd, 1984] Timothy A. Budd, "An APL Compiler for a Vector Processor," *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[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.

[Burks *et al.*, 1946] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Report prepared for the U. S. Army Ordnance Department, 1946, Reprinted in [**?**, pages 92–119].

[Burton, 1984] F. Warren Burton, "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs," *ACM Transactions on Programming Languages and Systems*, 6(2):159–174, April 1984.

[Coffin, 1989] Michael H. Coffin, "Par: A Language for Architecture-Independent Parallel Programming," Technical Report 89–18, Department of Computer Science, University of Arizona, September 1989.

[Coffin, 1990] Michael H. Coffin, "An Approach to Architecture-Independent Parallel Programming," April 1990, seminar presented at the University of Rochester.

[Coffin and Andrews, 1989] Michael H. Coffin and Gregory R. Andrews, "Towards Architecture-Independent Parallel Programming," Technical Report 89–21a, Department of Computer Science, University of Arizona, September 1989.

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

[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.

[Cox and Fowler, 1989] Alan L. Cox and Robert J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32–44, Litchfield Park, AZ, December 1989, reprinted in Operating Systems Review 23(5), December 1989.

[Crowther *et al.*, 1985] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor," In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–540, August 1985.

[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.

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

[Fowler *et al.*, 1988] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, May 1988.

[Gilman and Rose, 1976] Leonard Gilman and Allen J. Rose, *APL: An Interactive Approach*, John Wiley & Sons, New York, second edition, 1976.

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

[Goldman *et al.*, 1990] Ron Goldman, Richard P. Gabriel, and Carol Sexton, "Qlisp: An Interim Report," In Takayasu Ito and Robert H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems*, number 441 in Lecture Notes in Computer Science, pages 161–181. Springer-Verlag, 1990, the Proceedings of the US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989.

[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, 1985] Robert H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Halstead, 1990] Robert H. Halstead, Jr., "New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools," In Takayasu Ito and Robert H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990, the Proceedings of the US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989.

[Harrison (Alverson) and Notkin, 1990] Gail Harrison (Alverson) and David Notkin, "Effective Parallel Portability," Technical Report 89-09-08 Revised, Department of Computer Science and Engineering, University of Washington, January 1990.

[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.

[Hilfinger, 1982] Paul N. Hilfinger, *Abstraction Mechanisms And Language Design*, ACM Distinguished Dissertation. MIT Press, 1982.

[Hudak, 1986] Paul Hudak, "Para-Functional Programming," *Computer*, 19(8):60–70, August 1986.

[Hudak, 1988] Paul Hudak, "Exploring Parafunctional Programming: Separating the What from the How," *IEEE Software*, 5(1):54–61, January 1988.

[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.

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

[Jul *et al.*, 1988] Eric Jul *et al.*, "Fine-Grained Mobility in the Emerald System," *IEEE Transactions on Software Engineering*, 6(1), February 1988.

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

[Kranz *et al.*, 1986] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams, "ORBIT: An Optimizing Compiler for Scheme," In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986, in SIGPLAN Notices 21(7), July 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.

[LeBlanc, 1986] 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, 1988] Thomas J. LeBlanc, "Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor," In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, number 13 in IMA Volumes in Mathematics and its Applications, pages 145–163. Springer-Verlag, 1988.

[LeBlanc *et al.*, 1990] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, 9(2):203–217, June 1990.

[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.*, 1986] Barbara H. Liskov, Maurice P. Herlihy, and Lucy Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 150–159, January 1986.

[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.

[Low, 1976] James R. Low, *Automatic Coding: Choice of Data Structures*, Number 16 in Interdisciplinary Systems Research. Birkhäuser Verlag, Basel and Stuttgart, 1976.

[Mellor-Crummey, 1989] John M. Mellor-Crummey, "Debugging and Analysis of Large-Scale Parallel Programs," Technical Report 312, Department of Computer Science, University of Rochester, September 1989, Ph.D. Dissertation.

[Miller and Stout, 1989] Russ Miller and Quentin F. Stout, "An Introduction to the Portable Parallel Programming Language Seymor," In *Proceedings of the Thirteenth Annual International Computer Software and Applications Conference*, pages 94–101. IEEE Computer Society, September 1989.

[Murtagh, 1983] T. P. Murtagh, *A Data Abstraction Language for Concurrent Programming*, PhD thesis, Cornell University, Computer Science Department, January 1983.

[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.

[Polychronopoulos, 1988] Constantine D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[Quiroz, 1991] César A. Quiroz, "Systematic Detection of Parallelism in Ordinary Programs," Technical Report 351, Department of Computer Science, University of Rochester, May 1991, Ph.D. Dissertation.

[Sabot, 1988] Gary Wayne Sabot, *The Paralation Model: Architecture-Independent Parallel Programming*, MIT Press, 1988.

[Sarkar, 1990] Vivek Sarkar, "PTRAN — The IBM Parallel Translation System," Technical Report RC-70566, Research Division, IBM T. J. Watson Research Center, Yorktown Heights, New York, May 1990.

[Sarkar and Hennessy, 1986] Vivek Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 17–26, July 1986.

[Scott, 1986] Michael L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 242–249, August 1986.

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

[Scott *et al.*, 1990] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh, "Multi-Model Parallel Programming in Psyche," In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, Seattle, WA, March 1990, reprinted in ACM SIGPLAN Notices 25(3).

[Snyder, 1984] Lawrence Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, 17(7):27–36, July 1984.

[Snyder, 1986] Lawrence Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential," Technical Report 86–03–04, Department of Computer Science, University of Washington, Seattle, Washington, March 1986.

[Stallman, 1989] Richard M. Stallman, *Internals of GNU CC*, April 1989.

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

[Steele and Hillis, 1986] Guy L. Steele, Jr. and W. D. Hillis, "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing," In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, August 1986.

[Strom *et al.*, 1991] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini, *Hermes: A Language for Distributed Computing*, Prentice-Hall Inc., 1991.

[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.

[Stroustrup, 1986] Bjarne Stroustrup, "An Overview of C++," *ACM SIGPLAN Notices*, 21(10):7–18, October 1986.

[Thimbleby, 1988] Harold Thimbleby, "Delaying Commitment," *IEEE Software*, 5(3):78–86, May 1988.

[Thomas, 1985] R. Thomas, "Using the Butterfly to Solve Simultaneous Linear Equations," Butterfly Working Group Note 4, BBN Laboratories, March 1985.

[Thomas, 1986] R. Thomas, "The Uniform System Approach to Programming the Butterfly Parallel Processor," BBN Report No. 6149, BBN Advanced Computers Inc., June 1986.

[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.

[Ullman, 1976] J. R. Ullman, "An Algorithm for Subgraph Isomorphism," *Journal of the ACM*, 23:31–42, 1976.

[Wirth, 1982] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin, second edition, 1982.

[Wolfe, 1989] Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

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

A Natasha Prototype Language

Better is the enemy of good enough.[1]

Natasha (Ната́ша) is a *proof-of-concept* language for the Matroshka (Матрёшка) parallel programming model. Natasha is *not* a development quality language. Since the primary reason for parallel programming is faster problem solution time, efficiency is a significant concern. To support efficient execution, Natasha is a statically typed language. This enables Natasha implementations to obtain performance comparable to Pascal, Fortran, and C. Some Natasha mechanisms have several different implementations, each appropriate to different parallelism, distribution, and communication. Natasha is an object-based language, but is not an object-oriented language because it provides no inheritance. The Matroshka mechanisms for parallelism are orthogonal to inheritance, so a prototype language need not contain inheritance.

A.1 Syntax

This section contains the formal definition of the Natasha syntax. The syntax notation uses $\equiv$ for non-terminal definition, $\{\ \}$ for repetitions of zero or more, $[\ ]$ for optional items, and $|$ for alternatives. Within the definition of a non-terminal, text in 'roman font' represents non-terminal symbols, text in '`typewriter font`' represents terminal symbols, and text in '*italic font*' is descriptive commentary.

A.1.1 Tokens

Natasha programs are composed of a sequence of lexical tokens, each composed of a sequence of characters. In reading the program text, the parsing algorithm will extend the length of its current token if it can. Otherwise, it will consider the token complete and start another token. That is, tokens will be as long as possible. Except within quoted strings, a space or line break will force the parser to start a new token.

Natasha programs are first passed through the C preprocessor. This means programmers should ensure that any '`#`' character at the beginning of a line is intended for the preprocessor.

[1] This appears to be an adaptation of Voltaire's "*Le mieux est l'ennemi du bien [The best is the enemy of the good]*", 1764.

Comments start with the characters ‘;;’ and extend to the end of the line. Comments are equivalent to white space.

Natasha tokens fall into two categories, simple and complex. The simple tokens each consist of a single string. Table A.1 lists these tokens. The complex tokens are strings

token	purpose	token	purpose
:	declare variable	;	terminate statement
.	construct port	!	invoke port
,	suppress making reference	,,	suppress making copy
[and]	delimit record constructor	[[and]]	delimit record type
(and)	delimit expression	{ and }	delimit statements
‘ and ’	delimit sub-names		

Table A.1: Simple Tokens

of characters from certain restricted classes. Table A.2 lists the character classes, and

digit ≡ 0–9
letter ≡ A–Z | a–z
alnum ≡ digit | letter | _ | $
operch ≡ - | @ | # | % | ^ | & | * | + | ~ | = | < | > | ? | / | : | |
nql ≡ *any character except* " *or newline*

Table A.2: Character Classes

table A.3 lists the complex token definitions. For example, a letter followed by a sequence

character ≡ \ *any-character-except-newline*
identifier ≡ letter { alnum }
integer ≡ digit { digit }
operator ≡ operch { operch }
string ≡ " { nql } " { " { nql } " }
pragma ≡ $ alnum { alnum }

Table A.3: Complex Tokens

of letters, numbers, and underscores forms an identifier. Some identifiers are reserved to the syntax or predefined as literals. Table A.4 lists them. Natasha provides minimal support for generic types via a composite names. A name consists of an identifier optionally followed by a left single quote ‘‘’, a sequence of names or integers, and a right single quote ‘’’. The names within single quotes are called *subnames*.

Table A.5 presents the Natasha grammar. The elements of this grammar are described later.

identifier	purpose
`closure`	declare closure
`cast`	reply casting to type of object
`false`	Boolean literal
`forward`	forward declaration of object methods
`method`	declare object operation methods
`object`	declare object types
`operator`	declare operator for an operation
`reply`	provide reply value for operation
`true`	Boolean literal

Table A.4: Reserved Identifiers

```
name         ≡ identifier [ ‘ subname { subname } ’ ]
subname      ≡ integer | name
primary      ≡ boolean | character | integer | string | closure | record
               | ( expression ) | name [ ,, ]
reference    ≡ name | expression [ , ]
expression   ≡ primary | expression ! primary | reference . identifier
               | reference operator object | reference operator

variable     ≡ name : expression ;
record       ≡ [ { variable } ]
field        ≡ identifier : name ; | identifier : rec-type ;
rec-type     ≡ [[ { field } ]]

execution    ≡ expression ;
reply        ≡ reply expression ; | cast expression ;
statements   ≡ { variable | execution }
parameter    ≡ | identifier : name | rec-type
body         ≡ parameter { { pragma } statements [ reply statements ] }
closure      ≡ closure body

method       ≡ method identifier body
               | method identifier parameter forward result-type-name ;
oper-def     ≡ operator binary-operator . identifier ! ;
               | operator unary-operator . identifier ! [ ] ;
obj-type     ≡ object identifier parameter
               | { { pragma } { variable | execution | method | operdef } } ;
type-decl    ≡ name : rec-type ; | name : obj-type ;
program      ≡ { type-decl | execution }
```

Table A.5: Grammar

A.2 Types

Natasha relies on *type objects* to define the type of objects. Type objects usually have a '`new`' operation, which returns instances of common programming types, such as integers. There is no way to name the type objects for type objects. Natasha recognizes certain generic types that are inherent to the language. In addition, it predefines some common simple types, and some synchronization types.

A.2.1 Inherent Types

The Natasha language does not support user-defined generic types. However, the language does recognize some generic types inherent to the structure of the language. Programmers name these types with the composite name mechanism.

Records compose objects into a larger object. The field identifiers and types together define the type of the record. They are structurally equivalent. Section A.4 presents the means for naming record types. There is one name for each structurally distinct record type. Natasha predefines the name '`empty`' to refer to the type object for an empty record, *i.e.* one with no components. This is the "empty" or "bottom" type of the language.[2]

References are pointers to objects of a given type. Name reference types with '`refer`'*type*-name`''`.

Ports provide the target for operation invocation. The types of the operation's parameter and result define the type of the port. Note that the type of the operation's object does not contribute to the port's type. Name port types with '`port`'*parameter-type*-name *result-type*-name`''`.

Arrays provide for fixed-length homogeneous lists. The number and type of the elements defines the array's type. Name array types with '`array`'integer-*number-elements element-type*-name`''`. The annotations `_DIVIDED` and `_MODULAR` specify distribution of elements among processors.

Table A.6 lists the operations on inherent types. Table A.7 lists the operations on their types.

A.2.2 Simple Types

Natasha provides literal values for four predefined simple types: '`boolean`', '`integer`', '`character`', and '`string`'. The literals for booleans are '`true`' and '`false`'. A sequence of one or more digits forms an integer. The range of integers is from −2147483648 to 2147483647. The result of two operations applied simultaneously to the same object are undefined, that is, operations are *not* atomic. A backslash '`\`' followed immediately by another character, except a newline, is the literal for that character. Natasha also predefines the variables `newline`, which initially contains the newline character, and

[2]There is no "top" type.

object type	name	operator	parameter type	result type
record	`copy`	`@`	`empty`	*record*
record	`assign`	`:=`	*record*	`empty`
`refer`‘name’	`copy`	`@`	`empty`	`refer`‘name’
`refer`‘name’	`assign`	`:=`	`refer`‘name’	`empty`
`port`‘param result’	`copy`	`@`	`empty`	`port`‘param result’
`port`‘param result’	`assign`	`:=`	`port`‘param result’	`empty`
`array`‘integer elem’	`copy`	`@`	`empty`	`array`‘integer elem’
`array`‘integer elem’	`assign`	`:=`	`array`‘integer elem’	`empty`
`array`‘integer elem’	`select`[a]	`#`	`integer`	`refer`‘elem’
`array`‘integer elem’	`forall`[b]		`port‘integer empty’`	`empty`
`array`‘integer elem’	`sequfor`[c]		`port‘integer empty’`	`empty`

[a]Returns a reference to the element, not a copy.

[b]A parallel iterator for the array. The annotations are `_SEQUENTIAL` (the default) and `_PARALLEL`. The additional annotation `_QUICK` specifies the non-blocking version of `_PARALLEL`. The argument port must not block.

[c]A sequential iterator for the array.

Table A.6: Operations on Inherent Objects

object type	name	operator	parameter type	result type
record-type	`copy`	`@`	`empty`	*record-type*
reference-type	`copy`	`@`	`empty`	*reference-type*
port-type	`copy`	`@`	`empty`	*port-type*
array-type	`copy`	`@`	`empty`	*array-type*
array-type	`new`[a]		`port‘integer` name’	`array`‘integer name’

[a]The array sequentially initializes its elements with the values returned from invoking the port with indices in the range $0 \ldots \text{integer} - 1$.

Table A.7: Operations on Inherent Type Objects `endfile`, which initially contains a character signaling the end of a file. A string is composed of a sequence of any characters, except the double quote and newline, enclosed by double quotes. The presence of a literal in the program yields a new object for each elaboration of the source.

The Natasha language predefines additional types that receive no special treatment in the language syntax or semantics, but for which the compiler or runtime library has special knowledge or implementations. The '`range`' type provides for iteration over an integer range.

Table A.8 lists the operations on `boolean` and `integer` objects. Table A.9 lists the operations on `character`, `string`, and `range` objects. Table A.10 lists the operations on their types objects. Natasha predefines the `boolean`, `integer`, `character`, and `range` variables to refer to these objects.

A.2.3 Synchronization Types

Since the Matroshka model does not provide synchronization between operations, the Natasha language must add mechanisms to synchronize. The '`semaphore`', '`condition`', and '`crew`' types provide synchronization. Operations on these objects are atomic. Note that there are no copy or assign operations on synchronization objects.

The Natasha implementation provides a print lock so that Natasha programs and the implementation can print without interleaving their output. The implementation predefines the `printlock` variable which contains a print lock object. There is no way to name the print lock type object. Operations on `printlock` are atomic.

Table A.11 lists the operations on synchronization objects. Table A.12 lists the operations on their types objects. Natasha predefines the `semaphore`, `condition`, and `crew` variables to refer to these objects.

A.3 Variables

Variables serve to capture objects for later use. Variables contain objects, that is, the variables adhere to the value or copy model. A variable name refers to an object within the least containing scope that defines that variable. More specifically, a variable name is a literal value for a reference to the corresponding object. Because of the copy model of variables, two different simple variables names must refer to two different objects. All variable declarations have an initializing expression, which implicitly defines the type of the variable.

Variable declarations have the form:

variable-name : expression ;

For example,

```
letters: 26;
```

defines the variable '`letters`' with the initial value '`26`', which is an integer, so the variable's type is `integer`.

object type	name	operator	parameter type	result type
boolean	copy	@	empty	boolean
boolean	assign	:=	boolean	empty
boolean	and	&	boolean	boolean
boolean	or	|	boolean	boolean
boolean	not	~	empty	boolean
boolean	if[a]		port‘empty empty’	empty
boolean	ifelse[b]		[[then:port‘empty empty’; Else:port‘empty empty’;]]	empty
boolean	print[c]		empty	empty
boolean	read[d]		empty	empty
integer	copy	@	empty	integer
integer	assign	:=	integer	empty
integer	assign_add[e]	:=+	integer	empty
integer	assign_subtract[f]	:=-	integer	empty
integer	add	+	integer	integer
integer	subtract	-	integer	integer
integer	multiply	*	integer	integer
integer	divide	/	integer	integer
integer	modulo	%	integer	integer
integer	negate	-	empty	integer
integer	absolute		empty	integer
integer	equal	=	integer	boolean
integer	not_equal	~=	integer	boolean
integer	greater	>	integer	boolean
integer	greater_equal	>=	integer	boolean
integer	less	<	integer	boolean
integer	less_equal	<	integer	boolean
integer	print		integer[g]	empty
integer	read[h]		empty	empty

[a] Invoke the argument port if the object is true.

[b] Invoke the then port if the object is true, otherwise invoke the Else port.

[c] Print T or F.

[d] Read T or F and set the object.

[e] Assign after add.

[f] Assign after subtract.

[g] The argument is the minimum field width.

[h] Read and set the object.

Table A.8: Operations on Boolean and Integer Objects

object type	name	operator	parameter type	result type
character	copy	@	empty	character
character	assign	:=	character	empty
character	equal	=	character	boolean
character	not_equal	~=	character	boolean
character	greater	>	character	boolean
character	greater_equal	>=	character	boolean
character	less	<	character	boolean
character	less_equal	<=	character	boolean
character	print		empty	empty
character	read[a]		empty	empty
string	copy	@	empty	string
string	assign	:=	string	empty
string	print		empty	empty
range	copy	@	empty	range
range	assign	:=	range	empty
range	sequfor[b]		port‘integer empty’	empty
range	forall[c]		port‘integer empty’	empty

[a] Read and set the object.

[b] Invoke the port sequentially for each integer in the range.

[c] Invoke the port for each integer in the range. The annotations are `_SEQUENTIAL` (the default), `_PARALLEL` (parallel with no distribution), `_DIVIDED` (parallel with divided distribution), and `_MODULAR` (parallel with modular distribution). The additional annotation `_QUICK` specifies a non-blocking implementation for non-sequential implementations. The argument port must not block.

Table A.9: Operations on Character, String, and Range Objects

object type	name	operator	parameter type	result type
boolean-type	copy	@	empty	*boolean-type*
boolean-type	while[a]		[[cond:port‘empty boolean’; body:port‘empty empty’;]]	empty
boolean-type	repeat[b]		port‘empty empty’	empty
integer-type	copy	@	empty	*integer-type*
character-type	copy	@	empty	*character-type*
character-type	new[c]		integer	character
string-type	copy	@	empty	*string-type*
range-type	copy	@	empty	*range-type*
range-type	new[d]		[[from:integer; to:integer;]]	empty

[a] Invoke the `cond` port and if it returns true, invoke the `body` port and repeat.

[b] Invoke the argument port and repeat if it returns true.

[c] Returns a character with the given ASCII code.

[d] Create a range inclusive of `from` and `to`.

Table A.10: Operations on Simple Type Objects

object type	name	operator	parameter type	result type
`semaphore`	`signal`		`empty`	`empty`
`semaphore`	`wait`		`empty`	`empty`
`condition`	`signal`		`empty`	`empty`
`condition`	`wait`		`empty`	`empty`
`crew`	`start_read`[a]		`empty`	`empty`
`crew`	`end_read`		`empty`	`empty`
`crew`	`start_write`[b]		`empty`	`empty`
`crew`	`end_write`		`empty`	`empty`
printlock	`signal`		`empty`	`empty`
printlock	`wait`		`empty`	`empty`

[a] Completes when no writer is active.

[b] Completes when no reader or writer is active.

Table A.11: Operations on Synchronization Objects

object type	name	operator	parameter type	result type
semaphore-type	`copy`	`@`	`empty`	*semaphore-type*
semaphore-type	`new`		`integer`[a]	`semaphore`
condition-type	`copy`	`@`	`empty`	*condition-type*
condition-type	`new`		`empty`	`condition`
crew-type	`copy`	`@`	`empty`	*crew-type*
crew-type	`new`		`empty`	`crew`

[a] The number of initial signals.

Table A.12: Operations on Synchronization Type Objects

A.4 Records

A record collects several objects into a single object. A set of variable declarations surrounded by brackets '`[`' and '`]`' specifies a record constructor. The expressions within the variable declarations of a record constructor are evaluated within the scope visible to the record constructor. For example,

```
[ from: 3; to: 8; ]
```

when executed, constructs a record with an integer '`from`' component containing '`3`' and the '`to`' component containing '`8`'.

A record need not contain any components. The expression '`[]`' denotes these empty records.

The form for specifying a record type is similar to that of a record constructor. A sequence of component specifiers surrounded by double brackets '`[[`' and '`]]`' specifies a record type. Each component specifier consists of a name followed by a colon, a type name or another record type specifier, and a semi-colon. For example,

```
[[ from: integer; to: integer; ]]
```

specifies the record type for the above record constructor.

The name '`empty`' is equivalent to the record specification '`[[ ]]`'. This name is needed for compound names because record specifiers cannot appear in names. Programmers may define names for their own record types by defining a variable with the record type as its value. For example,

```
bounds: [[ from: integer; to: integer; ]];
```

A.5 Expressions

Expressions produce free-floating objects, or values. These values exist for the lifetime of their use within enclosing expressions. Each literal in an expression provides a new instance of the corresponding type for each execution of the expression.

Expressions are composed of two major primitives: the make-port primitive '`.`' forms a port, and the apply '`!`' primitive invokes an operation by applying an parameter to a port. Both '`.`' and '`!`' are left associative and have equal priority. Expressions are evaluated sequentially. The components of an expression are *not* evaluated concurrently.

A.5.1 Make Port

The make-port primitive '`.`' takes an object reference on its left and an operation identifier on its right and returns a port object. The operation identifier and the type of the object reference uniquely determine the object to invoke.

If the left operand of make-port is an identifier, a reference to the corresponding variable in the least enclosing scope becomes part of the port. For example, '`foo.print`' makes a port from the '`print`' operation and the object reference produced by the '`foo`' identifier.

If the left operand of make-port is an expression, a reference to the expression's result becomes part of the port. For example, '`(3+4).print`' makes a port from the '`print`' operation and a object reference dynamicly created to refer to the result of the expression '`(3+4)`'.

In the case that an expression returns a reference, the automatic reference formation for expressions would normally create a indirect reference, which is not normally desired. This automatic creation of a reference can be suppressed with the '`,`' primitive, which follows the expression and precedes the '`.`'. For example, given that the expression '`a#3`' returns a reference to the third element of an array, the expression '`a#3,.print`' make a port from that reference and the '`print`' operation.

A.5.2 Apply Argument

The apply primitive '`!`' takes a expression yielding port on the left, takes an argument object on the right, applies the argument to the port, and returns the result of the corresponding operation invocation. For example, '`a.add!3`' applies the argument '`3`' to the port '`a.add`' and returns the result.

If an identifier appears on either the right or left of the '`!`' primitive, the '`copy`' operation is implicitly applied to the corresponding variable and the result is used in the intended invocation. The '`,`' primitive following one of these identifiers suppresses the '`copy`' operation and returns a reference to the variable. The left operand may be a general expression, but the right operand must be either a literal or a parenthesized expression.

In the event that an operation requires no arguments, the convention is to pass the empty record '`[]`' as an argument.

A.5.3 Operators

The general form of expression, involving make-port primitives, operation identifiers, and apply primitives can be somewhat verbose. With each operation, programmers may define an operator to abbreviate the three make-port, operation, and apply tokens with a single operator token. Operator tokens consist of one or more characters from the set '`@#%^&*+-=:<>?/|~`', excluding some specific sequences reserved to the syntax.

For example, the predefined type '`integer`' defines the '`+`' operator for the '`add`' operation. So, the expression '`a+5`' is equivalent to the expression '`a.add!5`'.

The conventions for operator definitions are: '`@`' for '`copy`' and '`:=`' for '`assign`'.

A.6 Closures

Closures specify a sequence of statements to be executed within the context of an object, a method, or another closure. The syntax consists of the '`closure`' keyword, a parameter definition, a left brace '`{`', a sequence of pragmas, a sequence of statements, and a closing right brace '`}`'. The closure pragmas are the same as the method pragmas.

Closures take a single parameter and returns a single result. A *parameter declaration* provides a means for referring to the parameters of closures, methods, and object types.

The normal parameter declaration consists of a name, a colon, and a type name. For example, '`addend: integer`' specifies that the parameter is named '`addend`' and that it is an integer. The parameter captures the argument to an invocation, much as a variable captures the result of an expression. The parameter object may then be used as a variable.

Natasha only supports a single parameter, so programmers that desire multiple parameters must collect them in a record. Natasha provides a *record constructor* for building records at the point of use. This notation in the context of invocations is as concise as a list of named parameters. For convenience, the name for a record parameter may be dropped from its definition. In this case, the names of the record's components are directly accessible from the new scope. This enables the programmer to avoid naming the record itself and provides direct access the record components similarly to variables. For example, when using the parameter declaration

```
[[ was: integer; now: integer; ]]
```

Some operations need no parameter. In this case, the parameter may be left unspecified and the compiler infers the '`empty`' record type. Note that the empty record must still be specified when invoking such an operation.

The statements of a closure include variable declarations, expressions to execute, and a single '`reply`' statement. These statements execute sequentially, in textual order. The '`reply`' statement computes the reply value and indicates its type. Statements may occur in any order, but variables must be declared before used. The absence of a '`reply`' is equivalent to a '`reply [];`' as the last statement in a closure.

The specification for a closure yields a port in execution. That is, the run-time value of a closure specification is a port.

A.7 Object Types

An object type provides a mechanism for producing user-defined objects. An object type is itself an object. For every object of a given type, there exists a type object. The purpose of type objects is to provide a means to declare types at compile time and to create objects at run time. Type objects generally respond to the '`new`' operation (with an appropriate parameter) and generate a new object. While this approach can require considerable compiler support, the prototype language restricts the occurrence of object types to the outermost scope so that they may be easily compiled.

The object type definition consists of the '`object`' keyword, a parameter declaration, a left brace '`{`', a sequence of statements, and a closing right brace '`}`' The statements are variable declarations, method declarations, operator definitions, and expression to execute. These statements may appear in any order, but variables and methods must be declared before used. The variables of the object type define the components of the type.

Invoking the `new` operation on the type object produces an object of that type. The parameter to the `new` operation must match the parameter of the object type. The parts of the object type execute in the order of their specification. The result of the `new` operation is the produced object after all of its type object parts have executed. For example,

```
object willy: integer { nilly: false; };
```

defines an object taking an integer parameter. The object will then contain two variables, the integer '`willy`' and the boolean '`nilly`'. However, this object is not useful because it defines no methods for operations.

The definition of an object may provides a single *method* for handling each operation. Methods describe the sequence of statements to execute when the named operation is invoked. They take a single parameter and returns a single result. The method definition syntax consists of the '`method`' keyword, the operation identifier, a parameter definition, a left brace '`{`', a sequence of pragmas, a sequence of statements, and a closing right brace '`}`'.

The pragmas control the implementation of methods. The pragmas are:

`$task` — Create a task executing the method on the processor containing the object.

`$ltask` — Create a task executing the method on the processor invoking the operation.

`$migrate` — If the object is on the processor invoking the operation, execute the method as a procedure; otherwise, the object is on a remote processor, and create a task executing the method on the processor containing the object.

`$call` — Execute the method as a procedure, using remote memory references if necessary.

`$inline` — implement the method as procedures with `gcc`'s `inline` pragma, using remote memory references if necessary.

Method statements include variable declarations, expressions to execute, and a single `reply` statement. Statements may occur in any order, but variables must be declared before used. The `reply` statement indicates the reply value. The type of the expression in the reply statement determines the type of the operation result.

An object type is not available within its definition, which would normally imply that no recursive definitions are possible. However, Natasha provides minimal support for recursive definitions in three ways:

1. An object may refer to the pseudo-variable `self`. The compiler evaluates the type of `self` only after completing the elaboration of the object definition.
2. Methods may be declared before they are defined. Forward declarations have the form:

   `method` identifier parameter `forward` *result-type*-name

3. A second type of reply enables operations to return an object of the type being defined. The `cast` statement converts a record with the same components as the object type into an instance of the object. The resulting object is the operation's reply value. The compiler does not check that the record components and object variables match, so the programmer must ensure that they do.

Figure A.1 provides an example of a recursive operation. It contains an object type that computes factorials, the creation of an instance of that type, and the computation of 4!.

```
factorial: object  ;; the factorial object has no parameter
{ ;; the forward declaration
  method evaluate argument: integer forward integer;

  ;; the actual method for the evaluate operation
  method evaluate argument: integer
  { result: 1;  ;; result is 1 in the base case
    argument ~= 0 .if! closure
    { result := (self.evaluate!(argument - 1) * argument);
    };
    reply result;
  };
};

instance: factorial.new![];
finally: instance.evaluate!4;
```

Figure A.1: Example Computing Factorials Recursively

An operator definition provides an abbreviation for operation specification. It comes in two forms,

`operator` *binary*-operator . identifier `!` ;
`operator` *unary*-operator . identifier `!` `[ ]` ;

for binary and unary operators, respectively. The user may define an arbitrary number of operators, chosen from strings of the characters '`@#%^&*+-=:<>?/|~`'. Each forms an abbreviation for a sequence '.identifier`!`' (binary) or '.identifier`![]`' (unary). A single operator may have both a unary and a binary interpretation, the parser determines which form is used. The appearance of an operator is semantically equivalent to its syntactic replacement by its definition. Operator definitions apply only to the object types which contain them.