Parallel Programming with Control Abstraction

Lawrence A. Crowl   Thomas J. LeBlanc*

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 93-60-15

June 1993

Abstract

Parallel programming involves finding the potential parallelism in an application and mapping it to the architecture at hand. Since a typical application has more potential parallelism than any single architecture can exploit effectively, we usually program the parallelism that the available control constructs express easily and that the given architecture exploits efficiently. This approach produces programs that exhibit much less parallelism than exists in the application, and whose performance depends critically on the underlying hardware and software.

In this paper we argue for an alternative approach based on *control abstraction.* Control abstraction is the process by which programmers define new control constructs, specifying constraints on statement ordering separately from an implementation of that ordering. 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 exploiting a different subset of the parallelism admitted by the construct. By selecting an implementation for each control construct using annotations, we can vary the parallelism in a program to best exploit the underlying hardware without otherwise changing the source code. This approach produces programs that exhibit most of the potential parallelism in an algorithm, and whose performance can be tuned simply by choosing among the various implementations for the control constructs in use. We use several example applications to illustrate the use of control abstraction in parallel programming and performance tuning, and describe our implementation of a prototype programming language based on these ideas on the BBN Butterfly.

This work was supported by the National Science Foundation under research grant CDA-8822724, and the Office of Naval Research under research contract N00014-92-J-1801 (jointly funded by ARPA, HPCC Software Science and Technology program, ARPA Order No. 8930). The Government has certain rights in this material.

*Computer Science Department, University of Rochester, Rochester, New York, 14627-0226

Contents

1 Introduction

Applications generally contain more potential parallelism than any one machine can effectively exploit. Although an application may have an efficient realization on a wide range of architectures, including vector processors, bus-based shared-memory multiprocessors, distributed-shared-memory machines, and distributed-memory multicomputers, each class of architecture may exploit a different subset of the parallelism inherent in the algorithm. When we write a program, we typically limit consideration to the parallelism in the algorithm that a given machine can effectively exploit, and ignore any other potential parallelism. While this approach may result in an efficient implementation of the algorithm on a given machine, the program is difficult to tune or port to different architectures, 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.

Architectural adaptability is the ease with which programmers can tune or port a program to a different architecture. Many sequential programs adapt easily to a new architecture because the source code embeds few assumptions about the underlying machine. Parallel programs, on the other hand, often contain embedded assumptions about the overhead of process management, and the cost of communication and synchronization. When an architecture violates any of these assumptions, the program must be restructured to avoid serious performance degradation or to exploit alternative sources of parallelism. This restructuring can be complex, because the underlying assumptions are rarely explicit, and the ramifications of each assumption are difficult to discern.

In this paper we address one aspect of architectural adaptability for parallel programs: the ease with which programmers can select the parallelism in an algorithm appropriate for a given machine. This particular aspect of adaptability is important because we often cannot predict the most efficient parallelization for a given architecture in advance, and a significant change in parallelization may require a drastic change in source code.

Our approach to adaptability requires that a program specify the potential parallelism in an algorithm that architectures of interest might exploit. Only a subset of the potential parallelism is realized on a given architecture, but including other potential parallelism in the source code facilitates performance tuning and porting to other architectures.

We are interested in supporting this approach in explicitly-parallel imperative programs. These programs use control flow constructs, such as `fork`, `cobegin`, and parallel `for`, to introduce parallel execution. Since the expression of parallelism in these languages is fundamentally an issue of control flow, the control constructs provided by the language can either help or hinder attempts to express and exploit parallelism.

Given the importance of control flow in parallel programming, 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. What is required is 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.

With control abstraction, programmers can build new control constructs beyond those a language may provide. Each programmer-defined control construct accepts as a parameter a body of code to execute and its execution environment — a closure. The implementation of the construct specifies an ordering among different executions of the body of code. For example, using control abstraction we can define a `forall` construct that accepts a range of integers and a body of code to execute for each integer in the range. The semantics of `forall` could be that iteration $i+1$ may not proceed until iteration i ends, thereby requiring sequential execution. Alternatively, the semantics might allow iteration $i+1$ to overlap or even precede iteration i, admitting parallel execution. Using control abstraction, the programmer specifies the exact semantics of the construct, as well as the implementations.

Much like data abstraction, which hides the implementations 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. When the semantics of a construct, such as `forall`, admit either a parallel or sequential implementation, the user of the construct need not know which implementation is actually 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 of control constructs (and hence the parallelism to be exploited) without changing the meaning of the program, and thereby achieve architectural adaptability.

1.1 Related Work

In creating a parallel program, the programmer must decide what parallelism to exploit, how to map that parallelism to processors, how to distribute data among processors, and how to communicate between parallel tasks. Researchers have proposed several techniques that address each of these problems; in this paper we focus on the first problem, specifying and exploiting parallelism. Our goal is to expose all of the potentially useful sources of parallelism in the source code of a program, while allowing the user to select the parallelism to exploit in the implementation. Our approach is compatible with techniques developed by others to address mapping [Hudak, 1986; Snyder, 1984], distribution [Coffin and Andrews, 1989; Coffin, 1992; Alverson and Notkin, 1992], and communication [Black *et al.*, 1987].

Parallel Function Evaluation

Functional programs have no side effects, so expressions may be evaluated in any order. As a consequence, parallelism in functional programs is implicit, in that expressions can be evaluated in parallel. There are two sources of parallelism in function evaluation: parallel evaluation of multiple arguments to a function and evaluation of a function in parallel with its caller (the *promise* or *future*). 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 by selecting either lazy or eager (and potentially parallel) evaluation. For example, Multilisp [Halstead, 1985] (and Qlisp [Goldman *et al.*, 1990]) provides the function `pcall` for parallel argument evaluation, and `future` for parallel expression evaluation. 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`.

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 can 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 restrict parallelism to a particular set of operations on data structures.

The Paralation model [Sabot, 1988] and 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, which represent a limited use of control abstraction, are not a general solution to the problem of specifying parallelism in explicitly-parallel programs, since they define parallelism solely in terms of a particular data structure.

Fixed Control Constructs

Explicitly parallel languages typically provide a limited set of parallel control constructs that programmers use to simultaneously represent and exploit 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.

The Par language [Coffin and Andrews, 1989; Coffin, 1990; Coffin, 1992] (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 specify 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. 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`. There is a temptation 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.

Control Abstraction

Hilfinger [1982] provides a short history of major abstraction mechanisms in programming languages, with an emphasis on procedure and data abstraction. This history does not mention control abstraction, although the mechanisms for control abstraction are present in Lisp. Control abstraction has been used in several sequential languages to support data abstraction. For example, CLU iterators [Liskov *et al.*, 1977] (or generators) are a limited form of control abstraction that allows the user of an abstract type to operate on the elements of the type without knowing the underlying representation. In CLU, and other languages designed to support data abstraction, control abstraction plays a secondary role to the specification and representation of abstract data types.

Given that parallelism is a form of control flow, control abstraction is particularly important for parallel programming. Yet, to our knowledge, only those parallel programming languages that inherit control abstraction from a parent sequential language support it. Thus, even though Multilisp and Paralation use Lisp closures in the implementation of the parallel programming constructs presented to users, there is little or no recognition of the benefits of user-defined control abstractions as a parallel programming tool.

User-Defined Control Constructs for Parallel Programming

BBN's Uniform System [Thomas and Crowther, 1988] represents one approach to user-defined control constructs for parallel programming. The Uniform System provides a global shared memory and a general-purpose task activation routine, called `ActivateGen`. This routine takes as parameters a task generation procedure and a work procedure. The task generation procedure typically consists of a loop that generates parameters for the work procedure; idle processors invoke the task generation procedure to get work. Thus, generators are a form of control abstraction. The Uniform System provides built-in generators for manipulating arrays and matrices, but allows customized generators to be implemented by calling `ActivateGen` directly.

Chameleon [Alverson, 1990; Alverson and Notkin, 1992] extends this form of control abstraction by separating the partitioning and scheduling policy from task generation. Thus, a task generator in Chameleon might specify that a work procedure is to be applied to all elements in a two-dimensional array, but the assignment of work to processors is specified separately in a partition-scheduler policy object (ps-object). By selecting among multiple ps-objects for a single task generator, one can easily vary the amount of work assigned to each processor and control the assignment of tasks to processors. In addition, a ps-object can embed both affinity scheduling (executing a task on a processor whose local memory contains the required data) and software caching (loading the required data into local memory before execution begins).

Chameleon's representation of parallelism is based on C++ functions, so the programmer must explicitly pass the environment of a task as a parameter to the task. In addition, the dynamic nature of binding under Chameleon introduces overhead on every task that precludes the use of tasks for fine-grain parallelism.

Both the Uniform System and Chameleon are runtime libraries, not programming languages, and therefore have similar limitations. Both systems have separate operations for data representation and scheduling, but provide no explicit link to ensure compatible implementations. Both systems use a run-to-completion execution model for tasks, which requires that all synchronization use busy-waiting. Both systems were designed for numeric problems, so the data distribution strategies are primarily intended for use on arrays or matrices, and the control abstractions are limited to various forms of loops.

The primary focus of both Par and Chameleon is on the use of data abstraction and schedulers to hide data and processing distributions that may vary across architectures. Like CLU, Par and Chameleon provide the minimum control mechanisms needed to support data abstraction and distribution; our approach to specifying parallelism via control abstraction is complementary to their approach to specifying data distribution via data abstraction.

1.2 Overview of the Paper

In the following section we present a notation for describing constraints on control flow in the implementation of a control construct, and introduce a small set of mechanisms for programming with control abstraction. We use these mechanisms and notation to define some common constructs for parallel programming, and present several implementations for each construct. In section 3 we use a number of concrete example applications to demonstrate the power of control abstraction in parallel programming, and to show some of the effects on programming methodology that result from the liberal use of control abstraction. In section 4 we illustrate the role of control abstraction in performance tuning by porting a parallel program among seven different shared-memory multiprocessors. This example not only illustrates the importance of multiple parallelizations for a single application, it also demonstrates the effectiveness of control abstraction for tuning the performance of parallel programs. In section 5 we describe our BBN Butterfly implementation of Natasha, a prototype parallel programming language that supports control abstraction, and argue that parallel programs based on control abstraction can achieve execution efficiency comparable to that of conventional programming languages. Finally, in section 6, we summarize our experiences and present our conclusions.

2 Control Abstraction

In this section we present a notation for describing the temporal relations of a parallel control construct, and introduce a small set of mechanisms for parallel control. Using these mechanisms and our notation, we define an interface and implementation for three common control constructs, and show that the implementations meet the specifications in the interfaces. Finally, we show alternative implementations for these control constructs, each of which exploits a different subset of the parallelism admitted by the construct.

2.1 Temporal Relations for Control Constructs

A control construct defines an order of execution for a set of compound statements. A sequential control construct, such as `if` and `while`, defines a *total order* on the execution of the constituent statements. In contrast, a parallel control construct, such as `forall` and `cobegin`, defines a *partial order* of execution; the implementation of the construct need only execute the compound statements in an order consistent with that partial order.

Events and the Precedes Relation

Control constructs execute a compound statement as a single, indivisible unit. That is, once a compound statement begins execution, it will make progress towards completion unless it contains explicit synchronization. In addition, a control construct may be used in many different contexts, and therefore the implementation of a construct cannot in general exploit knowledge of the internal structure of the compound statements it executes. Given these two facts, a control construct can only impose a partial order of execution between compound statements in terms of two *events* that take place during the execution of a compound statement: the control transfer from the control construct to a compound statement, and the corresponding return. We will use $\downarrow$ *stmt* to denote the transfer of control to a compound statement, and $\uparrow$ *stmt* to denote its return.

In defining a control construct, we must be precise about the allowable execution orderings. We will use the *precedes* relation to describe the partial order for parallel control constructs. Given two events, a and b, the notation $a \rightarrow b$ states that a must precede b. That is, in all possible executions of the program, event a occurs before event b. The precedes relation is transitive, but not symmetric.

The transfer to a compound statement always precedes its return; that is $\forall$ *stmt,* $\downarrow stmt \rightarrow \uparrow stmt$. In addition, a control construct must begin execution before it can execute any statements. That is, if *stmt* is passed as a parameter to the control construct *cc*, then $\downarrow cc \rightarrow \downarrow stmt$. This precedence constraint is implicit in every control construct, so we use the convention that the absence of any such rule for a given statement signifies that the statement is not executed at all. That is, a control construct executes a statement passed as a parameter *if and only if* the statement's execution is present in the precedence rules.

Anti-Precedes and Concurrent Relations

Any construct defined using only the precedes relation has a valid sequential implementation corresponding to a topological sort of the precedence relation. This sequential implementation may not always be appropriate however, especially when a program uses explicit synchronization. For example, if the compound statements representing the iterations of a parallel `forall` construct contain explicit synchronization, then one iteration may block awaiting the completion of a later iteration. A sequential implementation would cause deadlock in this case. To avoid this problem, we introduce the *anti-precedes* relation. When a control construct specifies that a anti-precedes b ($a \not\rightarrow b$), then in no implementation of the construct may $a \rightarrow b$ be true. In other words, b cannot wait (even indirectly) on a, although a can wait on b. Note that $a \not\rightarrow b$ is *not* equivalent to $\neg(a \rightarrow b)$. Also, while $a \rightarrow b \Rightarrow b \not\rightarrow a$, the converse is not true. That is, $b \not\rightarrow a \not\Rightarrow a \rightarrow b$ (i.e., a not waiting on b does not imply a preceding b). The anti-precedes relation is neither symmetric nor transitive.

For notational convenience, we also define the *concurrent* relation. When a control construct specifies that two events a and b are concurrent, $a \parallel b$, then $a \not\rightarrow b \wedge b \not\rightarrow a$. That is, in no implementation of the construct may either $a \rightarrow b$ or $b \rightarrow a$ be true. To satisfy this relation, the implementation of the control construct must use concurrency, either true parallelism or quasi-concurrent execution based on coroutines. The concurrent relation is symmetric, but not transitive.

Defining New Control Constructs

To create a new control construct, we must define the syntax used to invoke the construct, specify the precedence constraints that apply to every implementation of the construct, and provide at least one implementation of the construct that meets those precedence constraints. The syntactic description of a construct defines the parameters it expects, including any compound statements to be executed by the control construct. The precedence constraints are specified using the relations defined above. The implementation uses the primitive mechanisms of the language, and any previously-defined control abstractions, to specify an ordering on the execution of actions taken by the control construct.

Much like preconditions, postconditions, and invariants, our precedence notation is not a required part of the source code for a program. The precedence relations are used to define the execution flow of a control construct, and to reason about the correctness of an implementation. Although these relations are not required during compilation, we envision future programming systems that use explicit representations of this information to aid the programmer in writing parallel programs.

Finite Progress Assumption

The temporal relations used in the definition of a control construct only define a precedence relation between execution events. They do not indicate how quickly an event will occur, or whether it will occur at all. In order to make practical statements about whether or not control constructs terminate, we assume that all concurrent statements make finite progress. One practical implication of this assumption is that programs cannot use busy-waiting synchronization with non-preemptive scheduling without knowledge of the specific scheduling policy in use. Programs can use busy-waiting synchronization with a fair preemptive scheduler.

2.2 Primitive Mechanisms for Control Abstraction

We now introduce a small set of primitive mechanisms for parallel programming with control abstraction, and define the control flow of these mechanisms using our notation for precedence relations. The mechanism are: *statement sequencing*, *operation invocation*, *first-class closures*, *conditional execution*, *early reply*, and *condition variables*. These mechanisms are key components of Matroshka, an explicitly-parallel imperative programming model, and are incorporated into Natasha, a programming language based on the Matroshka model.[1] With these mechanisms, programmers can build a wide variety of control constructs to represent the parallelism in an application.

Statement Sequencing

A sequence of statements defines a total order on statement executions. Notationally, we separate statements by a semicolon. For example, $\mathtt{s}_1$; $\mathtt{s}_2$ means that $\downarrow \mathtt{s}_1 \rightarrow \uparrow \mathtt{s}_1 \rightarrow \downarrow \mathtt{s}_2 \rightarrow \uparrow \mathtt{s}_2$. (In future examples, we use the shorthand notation $\rightarrow$ *stmt* $\rightarrow$ in place of $\rightarrow \downarrow$ *stmt* $\rightarrow \uparrow$ *stmt* $\rightarrow$.)

Operation Invocation

Operations are recursive procedures (or methods) that accept parameters and return results. Operation invocation is synchronous with respect to the caller; that is, the caller waits for the result before proceeding. As in nearly all imperative programming languages, we require that argument evaluation precede operation invocation; that is, the invocation `f(g())` implies `g` $\rightarrow$ `f`. This

[1] A complete description of the Matroshka programming model and the Natasha programming language is beyond the scope of this paper; see [Crowl, 1991] for additional details.

requirement results in sequential evaluation of expressions, without limiting the potential for parallelism in control flow.

In our programming model, all interprocess communication is implemented by passing parameters to operations and returning results from them. Although our focus is on shared-memory multiprocessors, an implementation of this programming model on distributed-memory machines is possible, given an implementation of operation invocation based on remote procedure call or message passing.

We apply the operation invocation mechanism uniformly to both programmer-defined operations and language-defined primitive operations. For presentation purposes, we use a procedural notation for operation invocation, and a conventional infix notation for expressions.

First-Class Closures

General control abstraction requires a mechanism for encapsulating and manipulating a sequence of statements passed to a control construct. These statements must have access to the environment in which the control construct is used. Like Lisp, Smalltalk, and their derivatives, we use first-class *closures* to capture code and its environment. Closures capture their environment at the point of elaboration, and may reference (and change) variables in that environment, even though those variables may not be visible in the environment in which the closure is eventually called.[2]

Like procedures, closures may accept parameters and return results. Closures are also reusable, in that a program may invoke a closure many times. Each invocation produces a separate activation, and there is no implicit synchronization between activations.

Closures are, in essence, the in-line definition of a nested operation (procedure). Operations are simply named closures. Both operations and closures may be passed as arguments for later invocation.

In our notation, the definition of a closure consists of a parameter list within parentheses followed by a sequence of statements within braces. One of these statements may be the `reply` statement containing a return value expression. (A `reply` statement without an expression simply returns control.)

Using this notation, a closure that accepts an integer parameter and returns twice its value would be written as follows:

```
( arg: integer ){ reply 2*arg }
```

We can call a closure at the point of definition as follows:

```
( arg: integer ){ reply 2*arg }( 4 )
```

The first pair of parentheses defines the parameter type, the braces define the body of the closure, and the second pair of parentheses invoke the closure with an integer argument. This example is somewhat atypical; we normally name a closure and invoke it using that name, e.g. `twice( 4 )`.

[2]Closures may access variables in the surrounding environment using addresses in a shared-memory system or using messages in a distributed-memory system.

The invocation of a closure precedes the execution of the first statement in the closure. In addition, the evaluation of the reply value precedes the return from the closure. For example, assume that C has the closure definition

$\texttt{(arg: integer)\{ s}_{C_1}\texttt{; ...; s}_{C_i}\texttt{; reply f() \}}$

and that C is invoked by the following program fragment:

$\texttt{s}_1\texttt{; } C\texttt{(g()); s}_2\texttt{();}$

This program fragment results in the following partial order of execution:

$\texttt{s}_1 \rightarrow \texttt{g} \rightarrow \downarrow C \rightarrow \texttt{s}_{C_1} \rightarrow \cdots \rightarrow \texttt{s}_{C_i} \rightarrow \texttt{f} \rightarrow \uparrow C \rightarrow \texttt{s}_2$

As an example of the use of closures in the definition of a control construct, consider a `for` construct that iterates over a range of integers. The construct takes three parameters: an integer lower bound, an integer upper bound, and a closure (corresponding to the loop body) that accepts an integer parameter. The syntax for this construct is defined as follows:

```
define for( lower, upper: integer; body: closure( iteration: integer ) )
```

An example of its use is:

```
for( 1, 10, ( i: integer ){ print i } )
```

In our examples we use two notational shortcuts. First, when a closure takes no parameters, we omit the parameter list. Second, if a valueless reply is the last statement in a closure, we omit the reply statement.

Conditional Execution

For conditional execution we adopt the approach of Smalltalk [Goldberg and Robson, 1983] and depend on a *Boolean* type and built-in `if` operation that conditionally executes a closure. The syntax for this construct is defined as follows:

```
define if( cond: boolean; then_body: closure() )
```

The precedence constraints on the implementations are:

$\downarrow \texttt{if(} \textit{true}\texttt{, then\_body)} \rightarrow \texttt{then\_body} \rightarrow \uparrow \texttt{if}$
$\downarrow \texttt{if(} \textit{false}\texttt{, then\_body)} \rightarrow \uparrow \texttt{if}$

The first precedence relation states that an invocation of `if` with a conditional expression that evaluates to *true* precedes the execution of `then_body`, which represents the body of the `if` statement. The second precedence rule states that an invocation of `if` with a *false* condition precedes the return from the `if` operation. The absence of a precedence rule for invoking `then_body` when the condition is `false` means that it is not executed under those circumstances.

We invoke the `if` statement using a procedural notation:

```
if( y>0, { z := x/y } )
```

Given the `if` operation, we can build many other common control constructs for conditional execution. For example, we can define an implementation for an `ifelse` construct as follows:

```
implement ifelse( cond: boolean; then_body, else_body: closure() )
    { if( cond, then_body ); if( not cond, else_body ) }
```

Similarly, the `while` construct has the following recursive implementation:

```
implement while( cond: closure(): boolean; body: closure() )
    { if( cond(), { body(); while( cond, body ) } ) }
```

The `repeat` construct, which repeatedly executes a boolean function (or closure) until it produces *false* as a result, has the following implementation:

```
implement repeat( func: closure(): boolean )
    { if( func(), { repeat( func() ) } ) }
```

We will use each of these constructs in our example programs.

Early Reply

When an invocation of an operation (or closure) returns a result, it may continue executing in parallel with the caller. This mechanism, called *early reply*, is the sole source of parallelism in Matroshka.[3] This mechanism is not new (for example, see [Andrews *et al.*, 1988; Liskov *et al.*, 1986; Scott, 1987]), but its expressive power does not appear to be widely recognized.

An early reply defines a partial order of execution, which admits a parallel implementation. For example, assume that C has the closure definition

$\texttt{(arg: integer) \{ } \mathtt{s}_{C_1}\texttt{; ...; } \mathtt{s}_{C_i}\texttt{; reply f(); } \mathtt{s}_{C_j}\texttt{; ...; } \mathtt{s}_{C_k} \texttt{ \}}$

and that C is invoked by the following program fragment:

$\mathtt{s}_1\texttt{; } C\texttt{(g()); } \mathtt{s}_2$

This program fragment results in the following partial order of execution:

$$\begin{array}{lll} \mathtt{s}_1 \rightarrow \mathtt{g} \rightarrow \downarrow C \rightarrow \mathtt{s}_{C_1} \rightarrow \cdots \rightarrow \mathtt{s}_{C_i} \rightarrow \mathtt{f} \rightarrow \uparrow C & \rightarrow & \mathtt{s}_2 \\ & \searrow & \mathtt{s}_{C_j} \rightarrow \cdots \rightarrow \mathtt{s}_{C_k} \end{array}$$

The statements $\mathtt{s}_{C_j}$ **through** $\mathtt{s}_{C_k}$ may execute in parallel with statement $\mathtt{s}_2$ and its successors. That is, $\downarrow \mathtt{s}_{C_j} \not\rightarrow \downarrow \mathtt{s}_2 \wedge \downarrow \mathtt{s}_2 \not\rightarrow \downarrow \mathtt{s}_{C_j}$ and therefore $\downarrow \mathtt{s}_{C_j} \parallel \downarrow \mathtt{s}_2$.

Condition Variables

So far, we have presented no mechanism for synchronization other than that implicit in an operation invocation waiting for a reply. For the purposes of our examples, we use condition variables, which are easy to describe and are sufficient for our examples. We assume that an imperative parallel programming language based on the Matroshka model would provide other synchronization primitives, such as compare-and-swap or semaphores.

---

[3] Early reply differs from rendezvous in that a new execution stream is created by early reply, whereas rendezvous is a synchronization mechanism between two existing execution streams.

A condition variable has atomic `signal` and `pending` operations. The `signal` operation, which may only be invoked once for each condition variable, certifies that the condition associated with the variable has been established. The `pending` operation returns true if the condition has not yet been established, and false otherwise. It does not wait for the `signal`. The syntax for these operations is defined as follows:

```
define pending( var cond: condition ): boolean
define signal( var cond: condition )
```

The precedence rules are:

$\uparrow$ `pending:`*true* $\rightarrow$ $\downarrow$ `signal`
$\downarrow$ `signal` $\rightarrow$ $\uparrow$ `pending:`*false*

The first rule says that a reply from `pending` that returns *true* must precede the call to `signal`. The second rule says that any reply from `pending` that returns *false* must occur after the call to `signal`. Of course these rules only apply to invocations on the same condition variable.

2.3 Building Common Control Constructs

In this section, we provide examples of defining, implementing, and using parallel control constructs. We also show how to use precedence relations to verify that an implementation of a construct satisfies the precedence constraints for the construct. Our first example is the implementation of a busy-waiting operation on condition variables. We use this operation in the implementation of a parallel `cobegin` construct. We then use `cobegin` in the implementation of a parallel `forall` construct.

Wait on Condition

Given the condition variables defined in §2.2, we construct a `wait` operation that does not reply until a condition has been signaled. Our implementation will use busy-waiting; alternative implementations based on blocking synchronization are also possible.

```
define wait( var cond: condition )
```

$\downarrow$ `signal` $\rightarrow$ $\uparrow$ `wait`

This construct has a straightforward implementation using `if` and recursion:

```
implement wait( var cond: condition )
    { if( pending( cond ), { wait( cond ) } ) }
```

We use induction and the precedence relations of the primitive operations `pending` and `signal` to verify that this implementation satisfies the partial order specified in the definition of the construct. The base case occurs when `signal` has already been invoked at the time `pending` replies:

$\downarrow$ `signal` $\xrightarrow{1}$ $\uparrow$ `pending:`*false*
$\downarrow$ `wait` $\xrightarrow{2}$ `pending:`*false* $\xrightarrow{3}$ $\downarrow$ `if(` *false*`,` *stmt* `)` $\xrightarrow{4}$ $\uparrow$ `if` $\xrightarrow{5}$ $\uparrow$ `wait`

Precedence 1 derives from the definition of the primitive operations on condition variables. Precedence 2 derives from the first statement of a closure following the invocation of the closure. Precedence 3 derives from the evaluation of `pending` as an argument before invoking `if`. Precedence 4 derives from the definition of `if` with a false argument. Finally, precedence 5 derives from the implicit reply at the end of a closure, statement sequencing, and the definition of closures. By the transitivity of the precedes relation, we can infer $\downarrow \texttt{signal} \rightarrow \uparrow \texttt{wait}$.

The induction step occurs when `signal` has not yet been invoked when `pending` replies:

$\downarrow$ `signal` $\xrightarrow{1}$ $\uparrow$ $\texttt{wait}_{recursive}$
$\downarrow$ `wait` $\xrightarrow{2}$ `pending:`*true* $\xrightarrow{3}$ $\downarrow$ `if(` *true*`,` *stmt* `)` $\xrightarrow{4}$ $\texttt{wait}_{recursive}$ $\xrightarrow{5}$ $\uparrow$ `if` $\xrightarrow{6}$ $\uparrow$ `wait`

Precedence 1 is the inductive assumption derived above. Precedence 2 derives from the first statement of a closure following the invocation of the closure. Precedence 3 derives from the evaluation of `pending` as an argument before invoking `if`. Precedences 4 and 5 derive from the definition of `if` with a true argument. Precedence 6 derives from the implicit reply at the end of a closure, statement sequencing, and the definition of closures. Finally, by transitivity, we have $\downarrow$ `signal` $\rightarrow$ $\uparrow$ `wait`, which proves that the implementation meets the precedence constraints in the definition.

Cobegin

Our next example is a `cobegin` construct that allows two closures to execute concurrently, replying only when both have completed execution. Its syntax and precedence constraints are defined as follows:

```
define cobegin( stmt1, stmt2: closure() )
↓ cobegin → stmt1 → ↑ cobegin
↓ cobegin → stmt2 → ↑ cobegin
↓ stmt2 ↛ ↓ stmt1
```

These rules state, respectively, that both closures start execution after the `cobegin`, both closures reply before the `cobegin` replies, and in no implementation may `stmt2` precede `stmt1`.

These precedence rules permit, but do not guarantee, concurrent execution. We could add another precedence rule, $\downarrow$ `stmt1` $\not\rightarrow$ $\downarrow$ `stmt2`, and guarantee concurrent execution, since

$\downarrow$ `stmt2` $\not\rightarrow$ $\downarrow$ `stmt1` $\wedge$ $\downarrow$ `stmt1` $\not\rightarrow$ $\downarrow$ `stmt2` $\Rightarrow$ $\downarrow$ `stmt1` $\parallel$ $\downarrow$ `stmt2`

However, if $\downarrow$ `stmt1` $\parallel$ $\downarrow$ `stmt2`, then neither statement may precede the other, which implies that `cobegin` must invoke both closures before waiting on either reply, which in turn precludes a sequential implementation. In general, we avoid using control constructs that guarantee concurrency as building blocks for other constructs, because they exclude sequential implementations.

Note that the definition of `cobegin` given here is asymmetric, in that `stmt1` may precede `stmt2`, but `stmt2` may not precede `stmt1`. We could have defined a symmetric `cobegin` construct by simply eliminating the third precedence constraint. In that case, an implementation could execute `stmt1` and `stmt2` in any order, which would prohibit explicit synchronization between `stmt1` and `stmt2`. By including the third precedence constraint, we accommodate both a sequential implementation of `cobegin` and explicit synchronization between `stmt1` and `stmt2`. We require however that `stmt1` execute before `stmt2` in any sequential implementation, and that `stmt1` never wait for `stmt2` to proceed. In the next section we exploit this ordering of `stmt1` and `stmt2` in `cobegin` to build an implementation of `forall` in which lower-numbered iterations never wait for higher-numbered iterations.

One possible parallel implementation of `cobegin` appears in figure 1. It uses only the primitive mechanisms defined earlier, and the `wait` operation defined above. We use early reply as the source of parallelism and a condition variable for synchronization.

```
implement cobegin( stmt1, stmt2: closure() )
    { var done: condition;
      --- define and execute a closure to execute one argument
      { reply;           --- remainder of the closure executes in parallel
        stmt1();         --- invoke stmt1
        signal( done )   --- signal stmt2 after stmt1 has replied
      }();               --- directly execute the closure
      --- execution continues here, in parallel, after the reply executes
      stmt2();           --- invoke stmt2
      wait( done )       --- wait for signal indicating stmt1 has replied
                         --- implicit reply from cobegin
    }
```

Figure 1: Implementation of `cobegin`

We can show that the implementation meets the specification as follows:

$\downarrow \texttt{cobegin} \stackrel{1}{\to} \downarrow \text{inner closure} \stackrel{2}{\to} \uparrow \text{inner closure}$

$\uparrow \text{inner closure} \stackrel{3}{\to} \texttt{stmt1} \stackrel{4}{\to} \downarrow \texttt{signal} \stackrel{5}{\to} \uparrow \texttt{wait}$

$\uparrow \text{inner closure} \stackrel{6}{\to} \texttt{stmt2} \stackrel{7}{\to} \downarrow \texttt{wait} \stackrel{8}{\to} \uparrow \texttt{wait}$

$\uparrow \texttt{wait} \stackrel{9}{\to} \uparrow \texttt{cobegin}$

$\downarrow \texttt{stmt2} \stackrel{10}{\not\to} \downarrow \texttt{stmt1}$

Precedence 1 derives from the first statement in a closure executing after the closure is invoked. Precedence 2 is inherent to an operation replying after its invocation. Precedence 3 derives from the first statement after a reply in a closure executing after the reply. Precedences 4, 6 and 7 derive from statement sequencing. Precedence 5 derives from the definition of `wait`. Precedence 8 is inherent to an operation replying after its invocation. Precedence 9 derives from the implicit reply at the end of `cobegin` and the definition of closures. By transitivity of the precedes relation, we can infer that the implementation meets the first two precedence rules in the definition of `cobegin`. And finally, precedence 10 (and hence the third rule in the definition of `cobegin`) derives from the concurrent execution of the statement after a reply with the statement after a call to a closure containing an early reply. Thus, we have shown that the implementation satisfies the definition.

Forall

In our next example we define an iterator over a range of integers, analogous to a parallel `for` loop or a CLU iterator. The syntax for the construct is:

```
define forall( lower, upper: integer; body: closure( iteration: integer ) )
```

The precedence rules are:

$\downarrow$ `forall( lower, upper, body )` $\rightarrow$ $\downarrow$ `body(` i `)` $\qquad [i :$ `lower` $\leq i \leq$ `upper`$]$
$\uparrow$ `body(` i `)` $\rightarrow$ $\uparrow$ `forall( lower, upper, body )` $\qquad [i :$ `lower` $\leq i \leq$ `upper`$]$
$\downarrow$ `body(` j `)` $\not\rightarrow$ $\downarrow$ `body(` i `)` $\qquad [i, j :$ `lower` $\leq i < j \leq$ `upper`$]$

These rules state, respectively, that the `forall` starts before any iteration, all iterations reply before `forall` does, and lower-numbered iterations do not wait on higher-numbered iterations.[4] Once again, we purposely omit a rule that guarantees concurrency such as:

$\downarrow$ `body(` i `)` $\|$ $\downarrow$ `body(` j `)` $\qquad [i, j : i \neq j \wedge$ `lower` $\leq i, j \leq$ `upper` $]$

which states that the implementation must start all iterations before waiting on the reply of any iteration.

We can use `cobegin` and recursion to build a parallel *divide-and-conquer* implementation of `forall` as follows:

```
implement forall( lower, upper: integer; body: closure( iteration: integer ) )
    { if( lower = upper, { body( lower ) } );
      if( lower < upper, { middle := (lower + upper) div 2;
                           cobegin( { forall(lower, middle, body) },
                                    { forall(middle+1, upper, body) } ) } ) }
```

We omit the detailed proof that this implementation satisfies the definition, but note that we rely on the third precedence rule of `cobegin` to satisfy the third precedence rule of `forall`.

2.4 Multiple Implementations for Control Constructs

Control abstraction separates the definition of a control construct from its implementation, which permits multiple implementations for a given control construct. Since our rules for each of the control constructs defined previously deliberately left the partial order of execution underspecified, we can provide either a parallel or sequential implementation.

Given that we have multiple implementations for a given control construct, we need a mechanism for selecting an appropriate implementation. We use program annotations to associate the use of a control construct with an implementation. Each implementation of a control construct is named using an annotation; that name is then used to select the corresponding implementation at the point of use. In our examples we use descriptive names that denote the parallelism provided by an implementation (e.g., `$SEQUENTIAL`, `$PARALLEL`, `$BLOCKED`), but our compiler uses simple string matching of annotations to select implementations.

Cobegin

Our earlier implementation of `cobegin` used early reply and a condition variable to execute two statements in parallel. We can construct a sequential implementation of `cobegin` using statement sequencing:

[4] This last constraint imposes an ordering on iterations analogous to the `ORDERED` qualifier for `PARALLEL DO` in PCF Fortran [Leasure, 1990]. We exploit this property of `forall` in sections 3.2 and 3.3.

```
implement cobegin $SEQUENTIAL ( stmt1, stmt2: closure() )
    { stmt1(); stmt2() }
```

Note that the precedence rules in the definition of `cobegin` require that `stmt1` precede `stmt2` in any sequential implementation.

We can select either this sequential implementation of `cobegin` or the parallel implementation given above simply by using the corresponding annotation at the point of use.

Forall

We have already shown a divide-and-conquer implementation of `forall` based on cobegin. If we truly desire a parallel implementation of `forall` then we must add an annotation to that implementation so as to select the parallel implementation of `cobegin`. We will refer to the parallel divide-and-conquer implementation based on a parallel implementation of `cobegin` using the annotation `$DIVIDED`.

Of course there are many other possible implementations of `forall`. For example, rather than implement all iterations in parallel, it might be preferable to implement iterations in blocks of size N, where N is determined by the number of processors, the number of iterations remaining in the loop, or the granularity of parallelism that can be efficiently implemented on the target machine. One implementation based on this approach follows; other implementations based on dynamic loop scheduling algorithms, such as guided-self scheduling [Polychronopoulos and Kuck, 1987], could be implemented in a similar fashion.[5]

```
implement forall $BLOCKED ( lower, upper: integer;
                            body: closure( iteration: integer ) )
    { ifelse( lower+N > upper,
          { for( lower, upper, body ) },
          { cobegin $PARALLEL (
                { for( lower, lower+N-1, body ) },
                { forall $BLOCKED ( lower+N, upper, body ) } ) } ) }
```

Straightforward modifications to this implementation would allow consecutive iterations to execute in parallel, while iterations separated by P (the number of processors) execute in sequence. We will refer to this implementation as `$CYCLIC`.

In some cases vector processors can exploit the parallelism in a `forall` loop by invoking vector instructions. We would expect the compiler to recognize a `$VECTOR` annotation and produce vector instructions for the loop.[6] On a vector multiprocessor, such as the Alliant FX, a single program can use both the parallel and vector implementations of `forall`.

[5] There are several techniques that could be used to select a value for N at the point of use of the `forall` construct. We could add a parameter to the definition of the `forall` construct, however doing so would change the interface to the `forall` construct, and would require that we supply a value for N even in the case of a sequential implementation. Alternatively, we could use a form of macro substitution to define values for parameters in annotations.

[6] We claim no particular advantage over vectorizing compilers in this case, however this example does illustrate how control abstraction can be used to represent fine-grain parallelism explicitly.

In addition to the many parallel implementations of `forall` there are also valid sequential implementations. For example, we can implement `forall` using the built-in sequential `for` operation as follows:

```
implement forall $SEQUENTIAL ( lower, upper: integer;
                               body: closure( iteration: integer ) )
    { for( lower, upper, body ) }
```

We can also construct a sequential implementation by modifying the parallel divide-and-conquer implementation of `forall` to select a sequential implementation of `cobegin`. Although either approach results in a sequential implementation, the use of the built-in `for` operation has two advantages: the implementation of `forall` no longer requires an implementation of `cobegin`, and we avoid any overhead associated with invoking the user-defined `cobegin` operation.

These examples illustrate the power of control abstraction when used to define parallel control flow mechanisms. With control abstraction, the definition of a control construct represents potential parallelism, while the implementation specifies the parallelism that is actually exploited during execution. The programmer can vary the parallelism in a program by using annotations to select among the implementations for a set of control constructs, and thereby tune the program to a specific architecture or set of inputs. In the following section, we use several example programs to illustrate this process.

3 Parallel Programming with Control Abstraction

In this section we use concrete example programs to illustrate the issues that arise when writing parallel programs with control abstraction. We first show how to select a parallel implementation for Quicksort based on predefined control abstractions. We then use Gaussian elimination to illustrate the process of representing application-specific parallelism with control abstraction, including the interactions between control abstraction and explicit synchronization. We use a simple model of a light bulb to illustrate how to expose data dependences within closures so they may be incorporated directly into a control abstraction. In our final example, we examine the relationship between control abstraction and data abstraction in the context of a parallel program for subgraph isomorphism.

3.1 Selecting Parallelism with Predefined Control Constructs

In the following example we illustrate the use of annotations to select a particular parallelization for Quicksort using the predefined control construct `cobegin`. There are two potential sources of parallelism we consider.[7] When the input array is partitioned, the search for an element in the bottom half of the array that belongs in 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.

[7] In our examples, we assume a sequential implementation for any control construct for which no annotation is given.

```
var sorting: array[1..SIZE] of integer;
implement quicksort $COARSE ( lower, upper: integer )
{ var rising, falling, key: integer;
  if( lower < upper,
      { rising := lower;
        falling := upper;
        key := sorting[lower];
        while( { cobegin $SEQUENTIAL (
                     { repeat( { rising +:= 1;
                                 reply key >= sorting[rising] } ) },
                     { repeat( { falling -:= 1;
                                 reply key < sorting[falling] } ) } );
                 reply rising <= falling },
               { swap sorting[rising] and sorting[falling] } );
        sorting[lower] := sorting[falling];
        sorting[falling] := key;
        cobegin $PARALLEL ( { quicksort( lower, falling ) },
                            { quicksort( falling+1, upper ) } ) } ) }
```

In this particular implementation we chose to exploit the coarse-grain parallelism available during the recursive calls (using the `$PARALLEL` annotation to select the parallel implementation of the second `cobegin`) and chose not to exploit the finer-grain parallelism available during partitioning of the elements. We could experiment with fine-grain parallelism by simply changing the `$SEQUENTIAL` annotation to select the parallel implementation of the first `cobegin`.

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 to find automatically.

3.2 Representing Application-Specific Parallelism

There are two distinct approaches to deriving a parallel program with control abstraction. One approach begins with a sequential algorithm, and exposes any parallelism that does not violate the data dependences inherent in the problem. The alternative approach expresses all parallelism in the problem, and adds explicit synchronization that enforces data dependences. We will illustrate these two alternatives using Gaussian elimination as an example. We then show how to incorporate explicit synchronization within a control construct, and discuss the benefits of doing so.

To solve 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. In this example we will concentrate on the control constructs needed to compute the upper triangular matrix, which is calculated by eliminating (zeroing) the entries below the diagonal.

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. We refer to this operation as reducing row i with j. We cannot perform this operation until 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.*, $M_{i,k} = 0, \forall k < j$. These two constraints limit the amount of parallelism that we can expect to achieve.

Parallelizing a Sequential Program

A straightforward derivation of a parallel program for Gaussian elimination begins with the sequential algorithm for upper triangulation.[8]

```
var system: array[1..SIZE] of array[1..SIZE] of real;
for( 1, SIZE-1, ( pivot: integer )
    { for( pivot+1, SIZE, ( reduce: integer )
          { var fraction := system[reduce][pivot] / system[pivot][pivot];
            for( pivot, SIZE, ( variable: integer )
                { system[reduce][variable]
                      -:= fraction * system[pivot][variable] } ) } ) } )
```

A simple parallel implementation of this algorithm uses a parallel implementation of `forall` for the two inner loops.[9] This implementation exhibits very fine-grain parallelism, as the innermost loop consists of a small number of arithmetic operations and a single assignment statement. Vector processors could exploit the parallelism in the inner loop using the `$VECTOR` annotation. To port the program to a vector multiprocessor, we would use a parallel implementation for the outer `forall` and a vector implementation for the inner `forall`.

Many multiprocessors lack vector units and could not profitably exploit the parallelism in the inner loop. On these machines we could select an implementation that does not attempt to exploit fine-grain parallelism by choosing the `$SEQUENTIAL` annotation for the innermost loop. The resulting program, which has a sequential loop nested within a parallel loop nested within a sequential loop, exhibits a series of phases separated by the selection of a pivot. The partial order of execution is illustrated in figure 2.

Our experiments with this implementation on the BBN Butterfly showed that processors spend too much time waiting for other processors to complete each phase. These empirical results suggest a need for more parallelism in the implementation.

Understanding an Application's Parallelism

An alternative implementation of Gaussian elimination can be derived using the synchronization constraints of the problem, rather than the implicit synchronization that comes from serializing the outermost loop in the sequential algorithm. The problem constraints 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, these constraints (shown in figure 3) are expressed as follows:

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

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

[8] We choose pivot equations in index order; numerically robust programs choose pivot equations based on the data.

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

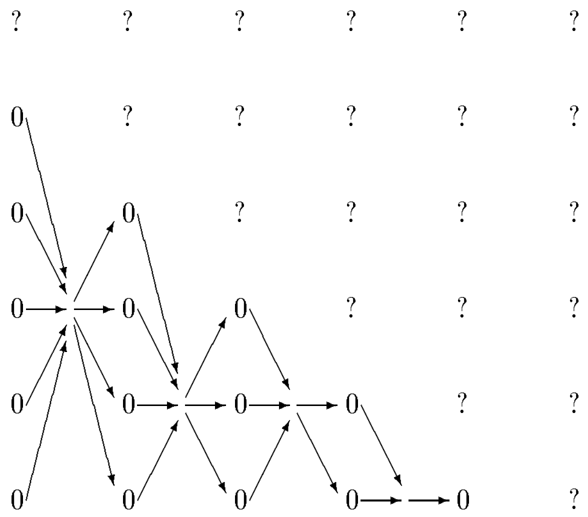

Figure 2: Phased Implementation of Gaussian Elimination

Rather than enforce these precedence constraints with serial execution, we can admit greater parallelism in the implementation and enforce the constraints with explicit synchronization. In this new implementation, we process all rows in parallel, and use condition variables to enforce the synchronization constraints.

```
var system: array[1..SIZE] of array[1..SIZE] of real;
var done: array[1..SIZE] of condition;
signal( done[1] );
forall $DIVIDED ( 2, SIZE, ( reduce: integer )
    { for( 1, reduce-1, ( pivot: integer )
          { wait( done[pivot] );
            var fraction := system[reduce][pivot] / system[pivot][pivot];
            forall( pivot, SIZE, ( variable: integer )
                { system[reduce][variable]
                      -:= fraction * system[pivot][variable] } ) } );
      signal( done[reduce] ) } )
```

In this fully-parallel implementation, all rows are processed in parallel. The `for` loop ensures that all entries of a row are eliminated in sequence, as is required by the problem constraints. As with the previous implementation, a particular pivot row is applied to all the entries in a row in parallel.

It is important to note that we cannot derive this particular version of the program from the sequential algorithm simply by selecting an appropriate combination of implementation choices for the `forall` construct. These two implementation choices represent a tradeoff between the execution overhead of explicit synchronization and the benefits of additional parallelism.

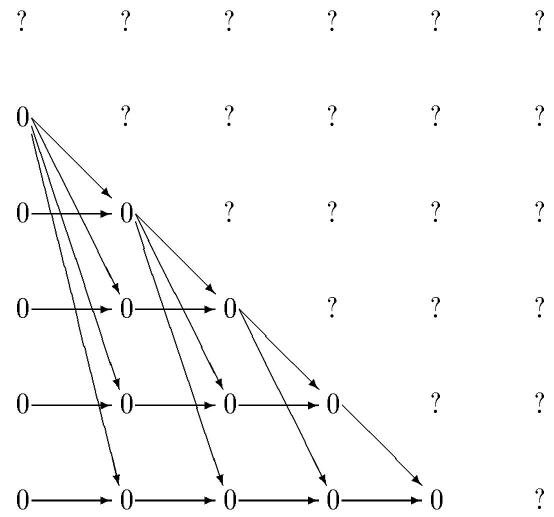

Figure 3: Precedence Constraints for Gaussian Elimination

Incorporating Explicit Synchronization in Control Constructs

There is a serious problem with the second implementation of Gaussian elimination given above: we cannot select the use of explicit synchronization in tandem with the parallelism we plan to exploit. In particular, we would have to remove the explicit synchronization if we change the annotation associated with the outermost loop to `$SEQUENTIAL`. The problem is that we have embedded parallelism in the loop control construct, and synchronization in the body of the loop.

To solve this problem we define a new control construct, `triangulate`, that moves synchronization from the body of the loop into the control construct. `Triangulate` takes two parameters: the number of equations in the system, and a closure containing the work to be performed for each pivot and reduction row pair. The construct encapsulates the possible parallelism and required synchronization in selecting pairs of pivot and reduction equations. The `triangulate` construct invokes the closure with the appropriate pairings, while maintaining the synchronization necessary for correct execution. We define `triangulate` as follows:

`define triangulate( size: integer; work: closure( pivot, reduce: integer ) )`
$\downarrow$ `triangulate( size, work )` $\rightarrow$ $\downarrow$ `work(` i, j `)` $\qquad [i, j : 1 \leq i < j \leq \texttt{size}]$
$\uparrow$ `work(` i, j `)` $\rightarrow$ $\downarrow$ `work(` k, j `)` $\qquad [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}]$
$\uparrow$ `work(` i, j `)` $\rightarrow$ $\downarrow$ `work(` j, k `)` $\qquad [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}]$

This construct has several implementations, including all of those discussed above. For example, we can create a sequential implementation of `triangulate` simply by selecting a sequential implementation of `forall` as follows:

```
implement triangulate $SEQUENTIAL
    ( size: integer; work: closure( pivot, reduce: integer ) )
    { for( 1, size-1, ( pivot: integer )
        { forall $SEQUENTIAL ( pivot+1, size, ( reduce: integer )
            { work( pivot, reduce ) } ) } ) }
```

Choosing the `$DIVIDED` annotation for `forall` produces `triangulate $PHASED`, which corresponds to the execution in figure 2. In addition, we can implement `triangulate $PHASED_BLOCKED` by selecting the `$BLOCKED` implementation of `forall`. To exploit more of the parallelism allowed by the problem's synchronization constraints, we can use the following implementation based on explicit synchronization:

```
implement triangulate $SYNCHED
    ( size: integer; work: closure( pivot, reduce: integer ) )
{ var done: array[1..size] of condition;
  signal( done[1] );
  forall $DIVIDED ( 2, size, ( reduce: integer )
      { for( 1, reduce-1, ( pivot: integer )
            { wait( done[pivot] );
              work( pivot, reduce ) } );
              signal( done[reduce] ) } ) }
```

This implementation admits greater parallelism than `triangulate $PHASED`, but may incur higher execution overhead due to synchronization. As before, we can replace `forall $DIVIDED` with `forall $BLOCKED` or `forall $CYCLIC` to obtain `triangulate $SYNCHED_BLOCKED` and `triangulate $SYNCHED_CYCLIC`.

This `triangulate` construct is similar to the built-in task generator `GenOnHalfArray` in BBN's Uniform System [Thomas and Crowther, 1988]. A Uniform System task 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.

`define GenOnHalfArray( size: integer; work: closure( index1, index2: integer ) )`

$\downarrow$ `GenOnHalfArray( size, work )` $\rightarrow$ $\downarrow$ `work(` i, j `)` $\quad [i, j : 1 \leq i < j \leq \texttt{size}]$

$\uparrow$ `work(` i, j `)` $\rightarrow$ $\uparrow$ `GenOnHalfArray( size, work )` $\quad [i, j : 1 \leq i < j \leq \texttt{size}]$

This generator provides the parallelism of our `triangulate` construct, but without the synchronization constraints. As a result, the Uniform System implementation must include explicit synchronization within the body of the work. Using our notation and closures, Gaussian elimination using `GenOnHalfArray` looks like this:

```
var system: array[1..SIZE] of array[1..SIZE] of real;
var pivot_done: array[1..SIZE] of condition;
var element_done: array[1..SIZE, 1..SIZE] of condition;
signal( pivot_done[1] );
GenOnHalfArray $DIVIDED ( SIZE, ( pivot, reduce: integer )
    { wait( pivot_done[pivot] );
      if( pivot > 1, { wait( element_done[reduce][pivot-1] ) } );
      var fraction := system[reduce][pivot] / system[pivot][pivot];
      forall $DIVIDED ( pivot, SIZE, ( variable: integer )
          { system[reduce][variable]
                -:= fraction * system[pivot][variable] } )
      signal( element_done[reduce][pivot] );
      if( pivot = reduce-1, { signal( pivot_done[reduce] ) } ) } )
```

This implementation uses explicit synchronization to provide the serialization implicit in the `for` loop in `triangulate $SYNCHED`. Given the limited facilities for creating new generators in the Uniform System, and the existence of `GenOnHalfArray`, this implementation is reasonable for the Uniform System. Nevertheless, a more efficient implementation is possible if the correct control construct is available or can be created easily.

Our original phased implementation of Gaussian elimination (based on the sequential algorithm), and the implementations based on `triangulate` and `GenOnHalfArray` illustrate the tradeoff between explicit synchronization and the synchronization implicit in sequential control constructs. For example, the synchronization implicit in the outermost sequential loop of our original phased implementation unnecessarily limits the amount of parallelism in the program. On the other hand, the explicit synchronization used in the Uniform System program is both expensive and unnecessary. The `triangulate $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, and a sequential `for` loop to serialize the application of pivots to a single equation, thus avoiding the extraneous explicit synchronization required in the Uniform System implementation.

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

```
var system: array[1..SIZE] of array[1..SIZE] of real;
triangulate $SYNCHED ( SIZE, ( pivot, reduce: integer )
    { var fraction := system[reduce][pivot] / system[pivot][pivot];
      forall $DIVIDED ( pivot, SIZE, ( variable: integer )
          { system[reduce][variable]
                -:= fraction * system[pivot][variable] } ) } )
```

By selecting an appropriate implementation for `triangulate` and the `forall` construct embedded in its body, we can describe all of the previous parallelizations of this problem. The programmer can select thirty five different implementations of this program by varying the two annotations to select a divide-and-conquer, blocked, cyclic, sequential, or vector implementation of `forall`, and a synchronized divide-and-conquer, synchronized blocked, synchronized cyclic, phased divide-and-conquer, phased blocked, phased cyclic, or sequential implementation of `triangulate`. In our experience, `triangulate $SYNCHED_CYCLIC` and `forall $SEQUENTIAL` produce the most efficient implementation for the Butterfly. (See figure 4 for a performance comparison of `triangulate $SYNCHED_CYCLIC` and `triangulate $PHASED_CYCLIC` on the Butterfly.) We would expect `triangulate $SYNCHED_CYCLIC` and `forall $VECTOR` to be the most appropriate combination for the Alliant. We can execute this same program on a Sun workstation by using `triangulate $SEQUENTIAL` and `forall $SEQUENTIAL`.

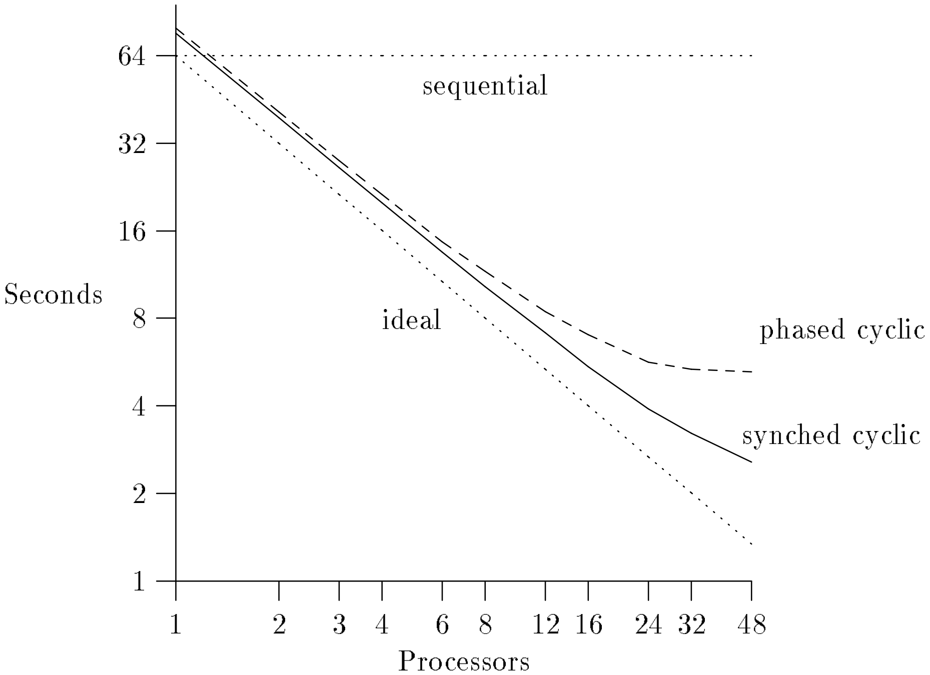

Figure 4: Performance of `triangulate` Annotations

In this simple example, the implementation of triangulate is more than half the size of the entire program. We expect that the amount of code dedicated to implementing control constructs in complete applications will be a much smaller fraction of the total code, especially when programmers have access to a library of control abstractions. Even in cases where the control constructs are a significant portion of the code, control abstraction isolates changes due to parallelism from the main logic of the program, including any required changes in synchronization.

3.3 Splitting Closures to Expose Data Dependences

In the previous example we were able to separate synchronization from the main body of computation (reducing a single equation in the matrix) and embed it in the `triangulate` control construct. The code to reduce an equation was unaffected by this change, since the dependences in the code body were between entire iterations. Our next example illustrates how to expose data dependences within a computation so as to isolate synchronization in the control construct.

Our example implements a model of an incandescent light bulb. The model accepts as input an initial temperature and a time-varying current. It produces as output the history of power dissipated (P), filament temperature (T), and luminance of the light bulb (L). The sequential code for this example is:

```
var P, T, L: array[0..N] of real;
T[0] := AMBIENT_TEMPERATURE;
for( 1, N, ( time: integer )
    { P[time] := Power_Dissipated( Current( time ) );
      T[time] := Temperature( T[time-1], P[time] );
      L[time] := Luminance( T[time] ) } )
```

This example has a loop-carried data dependence between iteration i and iteration $i+1$ in the calculation of temperature. We cannot use `forall` to specify parallelism in this example because we would violate this dependence. One possible approach is to insert explicit synchronization around the second statement in the loop, which contains 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 select the parallelism to exploit.

In order to move the synchronization into a control construct, we must split the body of the loop and expose the dependence to the loop construct. We therefore define a new `forall` construct that uses a form of pipelining. It 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.

```
define forall3( lower, upper: integer;
                head, body, tail: closure( iteration: integer ) )
↓forall3( lower, upper, 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 ) → ↑forall3( lower, upper, head, body, tail )     [i: lower ≤ i ≤ head]
```

For every iteration, the implementation of `forall3` must execute $\texttt{head}_i$, $\texttt{body}_i$, and $\texttt{tail}_i$ in sequence. In addition, the implementation must execute $\texttt{body}_i$ before $\texttt{body}_{i+1}$. Within these constraints, the construct admits several different parallel implementation. One implementation that might be produced by a parallelizing compiler executes all of the heads in parallel, each of the bodies in sequence, and all of the tails in parallel.

```
implement forall3( lower, upper: integer;
                   head, body, tail: closure( iteration: integer ) )
    { forall $DIVIDED ( lower, upper, head );
      for( lower, upper, body );
      forall $DIVIDED ( lower, upper, tail ) }
```

An alternative implementation that allows heads and tails to execute in parallel, thereby allowing even greater parallelism, uses explicit synchronization to enforce the dependence:

```
implement forall3( lower, upper: integer;
                   head, body, tail: closure( iteration: integer ) )
{ var done: array[lower..upper+1] of condition;
  signal( done[lower] );
  forall $DIVIDED ( lower, upper, ( i: integer )
      { head( i );
        wait( done[i] ); body( i ); signal( done[i+1] );
        tail( i ) } ) }
```

Using this control construct, we can write our light bulb example as follows:

```
var P, T, L: array[0..N] of real;
T[0] := AMBIENT_TEMPERATURE;
forall3( 1, N,
  ( time: integer ){ P[time] := Power_Dissipated( Current( time ) ) },
  ( time: integer ){ T[time] := Temperature( T[time-1], P[time] ) },
  ( time: integer ){ L[time] := Luminance( T[time] ) } )
```

By splitting the closure (which represents the loop body) to expose the data dependence, and defining a control abstraction that respects that dependence, we have isolated synchronization within the control construct. Once again, we can select the appropriate degree of synchronization and parallelism in tandem.

3.4 Data and Control Abstraction

In this section we use subgraph isomorphism, a well-known NP-complete problem, as an example to illustrate the relationship between data and control abstraction in parallel programs. Given two graphs, one small and one large, the problem is to find one or more isomorphisms from the small graph to arbitrary subgraphs of the large graph. An isomorphism is a mapping from each vertex in the small graph to a unique vertex in the large graph, such that if two vertices are connected by an edge in the small graph, then their corresponding vertices in the large graph are also connected by an edge.

Subgraph Isomorphism Algorithm and Data Representation

Before describing the interactions between data and control abstraction in subgraph isomorphism, we first describe the algorithm and data representation.

In our representation of graphs, 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 v, and contains the set of labels for the neighbors of v.

```
type SmallVertex = 1..MaxSmallVertex;
type LargeVertex = 1..MaxLargeVertex;
type SmallGraph = array[SmallVertex] of set of SmallVertex;
type LargeGraph = array[LargeVertex] of set of LargeVertex;
var smallG: SmallGraph;
var largeG: LargeGraph;
```

Our algorithm is based on Ullman's sequential tree-search algorithm [Ullman, 1976]. This algorithm postulates a mapping from one vertex in the small graph to a vertex in the large graph. This mapping constrains the possible mappings for other vertices of the small graph. The algorithm then postulates a mapping for a second vertex in the small graph, again constraining the possible mappings for the remaining vertices of the small graph. This process continues until an isomorphism is found, or until the constraints preclude such a mapping, at which point the algorithm postulates a different mapping for an earlier vertex.

The search for isomorphisms takes the form of a tree, where each node in the search tree is a *partial isomorphism*. For each vertex i in the small graph, a partial isomorphism contains the set of vertices j in the large graph to which we are still considering the possibility of mapping vertex i. When every vertex of the small graph has exactly one possible mapping to a vertex in the large graph, then the isomorphism is complete. If some vertex has no postulated mapping, then the partial isomorphism is invalid, and we prune that node from the search tree.

We represent nodes in the search tree, which correspond to postulated mappings of vertices in the small graph to vertices in the large graph, with an array of sets. Each element of the array corresponds to a vertex in the small graph, and the set contains the vertices in the large graph to which the vertex in the small graph might be mapped.

```
type PartialIsomorph = array[SmallVertex] of set of LargeVertex;
var root: PartialIsomorph;
```

The children of a node are constructed by selecting one possible mapping at the next level of the tree and then removing any conflicting mappings. Since the vertex i in the small graph may map to only one vertex j in the large graph, we remove all other mappings for the small graph vertex. In addition, no two vertices in the small graph may map to the same vertex in the large graph, so we remove the postulated large graph vertex from the possible mappings of all other small graph vertices.

Since the search space is very large, it is prudent to eliminate possible mappings early, before they are postulated in the search. We do this by applying a set of *filters* to the partial isomorphisms, reducing the number of elements in each mapping set, and pruning nodes in the search tree before they are visited. In our implementation we use only two filters, *vertex distance* and *vertex connectivity*. The vertex distance filter eliminates mappings where the distance between two vertices in the small graph is less than the distance between two vertices in the large graph. The vertex connectivity filter ensures that the possible mappings of a vertex in the small graph are consistent with the possible mappings of its neighbors.

There are many ways to exploit parallelism in the implementation of subgraph isomorphism. The coarsest granularity of parallelism occurs in the tree search itself; we can search each subtree of the root node in parallel with depth-first, sequential search at the remaining levels.[10] At each node of the tree, several filters must be applied so as to prune the search tree, and this set of filters could be executed in parallel.[11] We can also exploit parallelism when applying a filter to a candidate mapping. We will examine these alternative parallelizations in greater detail in Section 4. In this

[10] We could choose to implement search parallelism at any depth in the search tree, rather than solely at the root. We do not consider these other forms of search parallelism in this paper.

[11] Our implementation uses only two filters, but others are possible.

section we focus on the interactions between control and data abstraction in the implementation of the distance filter.

Iterators for Abstract Data Types

We begin our discussion of data and control abstraction with a straight-forward parallel implementation of the distance filter. This implementation has several problems, which we resolve over the next few sections through successive refinement using control abstraction.

We use the distance filter to ensure that no two vertices in the small graph separated by a distance x map to two vertices in the large graph separated by a distance $y > x$.[12] We rely on two precomputed arrays containing shortest-paths, `smallDist` and `largeDist`, to store distance information. For a set of possible mappings between vertices in the small graph to vertices in the large graph, and a given mapping from a particular vertex in the small graph to a vertex in the large graph, we eliminate any other possible mappings between vertices in the small graph to vertices in the large graph that violate the distance filter. The following is a straight-forward parallel implementation of the distance filter:

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall( 1, MaxSmallVertex, ( i: SmallVertex )
      { forall( 1, MaxLargeVertex, ( j: LargeVertex )
            { if( j in mapping[i],
                  { if( smallDist[smallV,i] < largeDist[largeV,j],
                        { remove_element( j, mapping[i] ) } ) } ) } ) } ) }
```

There is a problem with this implementation of the distance filter. If we select a parallel implementation of the innermost `forall`, we must pay the overhead of starting each parallel task that results. Since many postulated mappings are sparse, the first `if` condition is often false, and the corresponding task immediately terminates. In such cases, the overhead of creating parallel tasks may not be justified. The problem is that we want to iterate over the elements of a set, but `forall` iterates over the integer representation for vertices in the set and then tests for set membership. A better approach is to combine data abstraction and control abstraction by defining an iterator for sets, as in CLU. The resulting `forall_elements` construct executes a closure for each element of a set.

```
define forall_elements( members: set of integer;
                        work: closure( member: integer ) )
↓ forall_elements( members, work ) → ↓ work( i )                    [i : i ∈ members]
↑ work( i ) → ↑ forall_elements( members, work )                    [i : i ∈ members]
```

Given an implementation of `forall_elements`, we can rewrite the distance filter as follows:

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

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall( 1, MaxSmallVertex, ( i: SmallVertex )
      { forall_elements( mapping[i], ( j: LargeVertex )
            { if( smallDist[smallV,i] < largeDist[largeV,j],
                  { remove_element( j, mapping[i] ) } ) } ) } ) }
```

Given this version of the distance filter, we can choose to iterate over the possible elements of a set in sequence, and then apply the distance filter to each actual element in parallel. By doing so, we avoid the overhead of creating parallel threads of control for each possible element of a set.

Conditional Iterators

The last implementation of the distance filter uses iteration over the elements of a set to avoid creating a task for every potential element of a sparse set. Nonetheless, this implementation still suffers from the problem noted above; the first statement of the closure passed to `forall_elements` is an `if` condition, which may cause a newly created task to terminate immediately. Unfortunately, we cannot evaluate this condition in terms of the members of the set alone, and therefore cannot fold the test into a simple iterator. We can define a *conditional iterator* however, which solves the problem. Conditional iterators accept a condition to apply to elements of a data abstraction, as well as the work to perform on each element that satisfies the condition. With a conditional iterator, we can evaluate the conditions in sequence, avoiding the overhead of creating a parallel task for each `if` statement, while creating a parallel task for those elements that pass the test.

```
define forall_elements_cond
    ( members: set of integer;
      test: closure( member: integer ): boolean;
      work: closure( member: integer ) )
↓forall_elements_cond( members, test, work ) → ↓test( i )        [i : i ∈ members]
↑test( i ) → ↓work( i )                              [i : i ∈ members ∧ test( i )]
↑work( i ) → ↑forall_elements_cond( members, test, work )
                                                     [i : i ∈ members ∧ test( i )]
↑test( i ) → ↑forall_elements_cond( members, test, work )
                                                   [i : i ∈ members ∧ ¬ test( i )]
```

Given an implementation of `forall_elements_cond`, the distance filter becomes:

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall( 1, MaxSmallVertex, ( i: SmallVertex )
      { forall_elements_cond( mapping[i],
          ( j: LargeVertex ){ reply smallDist[smallV,i] < largeDist[largeV,j] },
          ( j: LargeVertex ){ remove_element( j, mapping[i] ) } ) } ) }
```

The performance benefits of using a conditional iterator in place of an iterator depend on the time required to evaluate the condition and the time required to operate on those elements that meet the condition. In this particular example, it could be that the time spent on elements that meet the condition is very small (comparable to the time required to apply the condition to an element), and therefore the tasks created by `forall_elements_cond` are too fine-grain for the architectures of interest. If so, the implementation based on `forall_elements` would suffice. However, if the condition is easy to evaluate, but the operation on elements that meet the condition is time-consuming, it would be worthwhile to separate the choice of parallelism for evaluating conditions from the choice of parallelism for operating on elements using `forall_elements_cond`.

Conditional Modification of Abstract Data Types

All of our implementations of the distance filter are based on a `set` abstract data type, which must export the `remove_element` operation. As in Multilisp [Halstead, 1985], we assume that data synchronization is embedded in data abstractions. Thus, the `remove_element` implementation must provide any synchronization needed to manage multiple invocations of `remove_element`. Unfortunately, there is no explicit coordination between the two set operations `forall_elements_cond` and `remove_element`, so the implementation of `remove_element` cannot know whether `forall_elements_cond` invokes `remove_element` in parallel or not. Thus, we cannot select parallelism and synchronization together.

As an alternative, we can combine the selection of parallelism and synchronization within a single operation that removes those elements of a set that meet a specified condition. This new operation, `remove_elements_cond`, applies a condition to each element of a set, and removes those elements that satisfy the condition.

```
define remove_element_cond( var members: set of integer;
                            test: closure( member: integer ): boolean )
↓remove_element_cond( members, test ) → ↓test( i )            [i : i ∈ members ]
↑test( i ) → ↑remove_element_cond( members, test )            [i : i ∈ members ]
```

The distance filter, using `remove_element_cond`, is as follows:

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall( 1, MaxSmallVertex, ( i: SmallVertex )
      { remove_element_cond( mapping[i], ( j: LargeVertex )
            { reply smallDist[smallV,i] < largeDist[largeV,j] } ) } ) }
```

In this final implementation we have two sources of parallelism, the implementation of `forall` and the implementation of `remove_element_cond`. Since the parallel tasks generated by `forall` operate on different partial isomorphisms (corresponding to `mapping[i]`), there is no need to synchronize operations on these sets. Since the parallel tasks in `remove_element_cond` all operate on the same set, we create a control construct that encapsulates both the parallelism and synchronization for that set operation.

Representation-Dependent Control Abstractions

Given a packed implementation of sets, in which each element is represented by a single bit in a 32-bit word, an implementation of `remove_element_cond` might process each word of the representation in parallel, and each bit in a word in sequence. This implementation offers substantial parallelism and yet requires no explicit synchronization. However, it depends both on the representation of sets, and the knowledge that no other operation is concurrently modifying the same set. This knowledge, which is available to `remove_element_cond`, is not available if the iterator and removal operation are separated.

In addition to the standard primitive operations on sets, such as `create`, `add_element`, and `remove_element`, we have added a variety of more complicated operations, including `forall_elements`, `forall_elements_cond`, and `remove_element_cond`. We can use these operations to implement other set operations, such as `intersection`, which given two sets S_1 and S_2, assigns $S_1 \cap S_2$ to S_1:

```
implement intersection( var S1: set of integer; S2: set of integer )
   { remove_element_cond( S1, ( i:integer ){ reply not membership( i, S2 ) } ) }
```

In building this rich variety of set operations there is a tradeoff to be made between representation-independent implementations and representation-dependent implementations. Given the power of control abstraction, the designer of a data abstraction might be tempted to provide a small set of primitive operations that exploit the underlying representation (such as `create`, `membership`, and `add_element`), and rely on control abstraction and representation-independent operations to provide all other operations. While this approach may simplify the implementation of the abstract data type, it precludes certain optimizations in the implementation of the control abstractions. For example, given a bit-vector representation of sets, the representation-independent implementation of `intersection` given above must evaluate the condition separately for every element in the set and remove those elements that meet the condition. In contrast, a representation-dependent implementation of `intersection` can exploit the bit-vector representation for sets, using logical "and" to implement intersection, and thereby avoid evaluating conditions and handling individual elements. In general, we will want a rich set of operations on each data type, so as to cope with the myriad parallelizations we might choose to exploit, and many of these operations will want to exploit the representation so as to maximize the potential for parallelism.

In summary, control abstraction encourages data representation-independent programming, which *users* of abstractions desire for architectural adaptability. *Designers* of abstractions must be careful to include a sufficiently rich variety of operations so that *implementors* of abstractions can take advantage of the parallelism inherent in the representation.

4 Performance Tuning with Control Abstraction

When implementing a parallel program, programmers must strike a delicate balance between the costs and benefits of parallelism. The potential benefits include faster execution due to parallel hardware, and better load balancing properties due to a fine-grain decomposition of work. The costs include the overhead of process management, synchronization, and communication. A significant change in any of these costs affects the decision about the appropriate granularity of parallelism in an application.

There are several situations where programmers must make decisions about how to parallelize a program:

- When implementing the program for the first time.
- When there is a dramatic change in the number of available processors.
- When porting the program from one machine to another.
- When exploiting special hardware features, such as vector processors.
- When optimizing the program for a particular class of input values.

The ease with which a programmer can tune the parallelization of a program to specific circumstances depends on the ease with which alternative parallelizations can be selected or implemented. The significance of program tuning depends on whether there is one best parallelization (and tuning therefore consists of a one-time search for that parallelization) or whether there is no single best parallelization (and therefore tuning is an ongoing effort that changes with circumstances).

In this section, we use subgraph isomorphism as an example application to illustrate the benefits of control abstraction during performance tuning. Subgraph isomorphism is representative of a large class of search problems, but more importantly, it contains many different sources of parallelism, and there is no obvious best choice. In fact, our experiments show that the best parallelization for subgraph isomorphism depends on the specific machine, the specific input, and the specific problem (the number of isomorphisms required). As a result, performance tuning is an on-going process, and the ability to change parallelizations easily is crucial for this application.

4.1 Parallelizations of Subgraph Isomorphism

Our algorithm for subgraph isomorphism has four primary sources of parallelism: (1) searching subtrees of a partial isomorphism in parallel (*search parallelism*), (2) applying multiple filters to a node of the search tree in parallel (*filter parallelism*), (3) applying a filter to all the vertices of a graph in parallel (*graph parallelism*), and (4) operating on all the elements in a set of vertices in parallel (*set parallelism*). We describe each of these sources of parallelism below, but our experiments focus primarily on the tradeoffs between search parallelism and graph parallelism.

Search Parallelism

The coarsest grain of parallelism we consider arises when examining the various possible mappings for the first small vertex. Given the set of possibilities in `mapping[smallV]`, we need to examine each postulated mapping. We can choose to examine each mapping in parallel using `forall_elements` as follows:

```
implement search $PARALLEL ( smallV: SmallVertex; mapping: PartialIsomorph )
{ forall_elements $DIVIDED ( mapping[smallV], ( postulate: LargeVertex )
      { examine( smallV, postulate, mapping ) } ) }
```

Alternatively, we can choose not to exploit parallelism in traversing the search tree simply by selecting a sequential implementation of `forall_elements`.

Search parallelism is relatively coarse grain, and therefore is suitable for most multiprocessors. Search parallelism is also *speculative* however, in that we might not need to search every subtree of the root in order to find the required number of solutions. In particular, if we only need one solution, and if the solution space is dense (as in the case where the smaller graph has few edges and the larger graph is almost fully connected), then a solution may be found in the first subtree. In this case, any time spent searching other subtrees is wasted.

Filter Parallelism

When we arrive at a node in the search tree, we must examine a single proposed mapping and propagate the constraints of that mapping. We must first enforce the minimal constraints of the proposed mapping: the vertex in the small graph must be mapped to a unique vertex in the large graph, and no other vertex in the small graph may be mapped to the same vertex in the large graph. Next, we must check to see if the partial isomorphism is a leaf in the search tree. If so, we report the isomorphism.[13] Otherwise, we apply better constraints.

```
implement examine
    ( smallV: SmallVertex; largeV: LargeVertex; mapping: PartialIsomorph )
{ minimal_constraints( smallV, largeV, mapping );
  ifelse( smallV = MaxSmallVertex,
          { report_possible_isomorphism( mapping ) },
          { constrain( smallV, largeV, mapping ) } ) }
```

We use two non-trivial constraints, vertex connectivity and vertex distance, to filter possible mappings. Each filter deletes those postulated mappings associated with a node in the search tree that violate the constraints imposed by the filter, thereby pruning the search space below the node. Since these filters only remove elements from sets of possible mappings, we may execute them in parallel.

```
implement constrain
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ cobegin $PARALLEL ( { distance_filter( smallV, largeV, mapping ) },
                      { connect_filter( smallV, largeV, mapping ) } );
  if( no_empty_mapping( mapping ),
      { search( smallV+1, mapping ) } ) }
```

Given only two filters, each node in the search tree can exploit at most two-way parallelism by executing the filters in parallel. This parallelism is not without cost however, since executing the filters in sequence allows the second filter to avoid examining any postulated mappings removed by the first filter.

Graph Parallelism

Each filter removes potential mappings based on some relationship between the candidate vertex in the small graph and other vertices in the small graph. For a given candidate vertex, we can examine constraints on the remaining vertices of the small graph in parallel. We can exploit this

[13]The constraint filters may leave some invalid isomorphisms at the leaves of the search tree. A separate check eliminates these leaf nodes before they are reported.

parallelism by choosing a parallel implementation of `forall` in the distance filter and a parallel implementation of `forall_elements` in the connectivity filter:

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall $BLOCKED ( 1, MaxSmallVertex, ( smallRel: SmallVertex )
      { remove_element_cond( mapping[smallRel],
            ( largeRel: LargeVertex )
            { reply smallDist[smallV,smallRel]
                    < largeDist[largeV,largeRel] } ) } ) }

implement connect_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall_elements $BLOCKED ( smallG[smallV],
      ( smallRel: SmallVertex )
      { assign_intersection( mapping[smallRel], largeG[largeV] } ) }
```

These implementations of filters offer many opportunities to exploit parallelism, but each parallel thread of control is relatively fine-grain. This source of parallelism may only be appropriate on machines and software systems that support fine-grain parallelism.

Set Parallelism

The finest grain of parallelism we consider arises when removing mappings that violate the constraints of a filter from the set of possible mappings for a small vertex. Given that we can represent the set of possible mappings as a vector of booleans, we can exploit parallelism in set operations in three ways. First, we can apply vector parallelism and operate on individual boolean values separately but in parallel. Second, we can pack multiple boolean values into a single machine word and use the bit operations common to most architectures to operate on multiple boolean values together and in parallel. Third, we can use both of the above and vectorize word-parallel operations. We refer to these implementations using the annotations `$VECTOR`, `$WORD`, and `$WORD_VECTOR`, respectively.

```
implement distance_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall( 1, MaxSmallVertex, ( smallRel: SmallVertex )
    { remove_element_cond $VECTOR ( mapping[smallRel],
        ( largeRel: LargeVertex )
        { reply smallG[smallV,smallRel] < largeG[largeV,largeRel] } ) } ) }

implement connect_filter
    ( smallV: SmallVertex; largeV: LargeVertex; var mapping: PartialIsomorph )
{ forall_elements( smallG[smallV],
    ( smallRel: SmallVertex )
    { assign_intersection $WORD ( mapping[smallRel], largeG[largeV] } ) }
```

The proper choice of annotation for set parallelism depends on the machine architecture and the exact implementation of the algorithm. The architecture may or may not have vector processors or the appropriate instructions for operating on the bits in a word. In addition, the cost of communication for a given machine determines the savings associated with using a packed representation of sets in communication operations. The algorithm determines the frequency with which we must pack and unpack sets into words, which could offset any savings from parallel set operations.

Summary of Parallelizations of Subgraph Isomorphism

We have identified four main sources of parallelism in our algorithm, and each source of parallelism has multiple implementations. Table 1 summarizes the set of choices that a programmer faces when choosing a specific implementation of subgraph isomorphism.

Operation	Control Construct	Annotations
`search`	`forall_elements`	`$SEQUENTIAL $BLOCKED $CYCLIC $DIVIDED`
`constrain`	`cobegin`	`$SEQUENTIAL $PARALLEL`
`distance_filter`	`forall`	`$SEQUENTIAL $BLOCKED $CYCLIC $DIVIDED`
	`remove_element_cond`	`$SEQUENTIAL $VECTOR $WORD $WORD_VECTOR`
`connect_filter`	`forall_elements`	`$SEQUENTIAL $BLOCKED $CYCLIC $DIVIDED`
	`assign_intersection`	`$SEQUENTIAL $VECTOR $WORD $WORD_VECTOR`

Table 1: Annotations for Parallelizations of Subgraph Isomorphism.

Choosing the best parallelization from among this myriad set of choices is a daunting task for the programmer. Search parallelism is coarse grain, and therefore likely to be worthwhile on most machines. However, search parallelism is also speculative, and may be of no value in cases where the solution space of a particular problem instance is dense. Graph parallelism is fine grain, and therefore may not be worth exploiting on hardware and software architectures that don't support fine-grain parallelism efficiently. On the other hand, graph parallelism may be required in order to exploit a large number of processors. The tradeoff between sequential and parallel application of filters depends on the cost and benefits of parallelism versus the reduction in work that comes from the serial application of filters. The choices for set parallelism depend on the capabilities and costs of the machine, and the frequency of certain set operations in the program.

Control abstraction did not create this difficult problem for the programmer, since these choices are inherent in the subgraph isomorphism algorithm. The majority of these choices would typically be ignored by the programmer or determined by the programming environment. Rather than choose a specific parallelization based on limitations of the programming environment, or implicit assumptions about the machine or the input, we specify the alternatives explicitly using control abstraction, and use experimentation (where appropriate) to analyze the tradeoffs involved.

4.2 Experimental Comparison of Parallelizations

In order to explore the tradeoffs between the various sources of parallelism described in the previous section, we performed a number of experiments with implementations of subgraph isomorphism.

These experiments cover a wide range of inputs, problem instances, machines, and parallelizations. The machines we used in this study include a 20 processor Sequent Balance, a 19 processor Sequent Symmetry, a 7 processor IBM 8CE, a 39 processor BBN Butterfly, a 21 processor BBN Butterfly TC2000, an 8 processor Silicon Graphics Iris multiprocessor workstation (a member of the Power Series), and a 32 processor Kendall Square Research KSR-1. We used randomly generated inputs, where the probability that a given leaf in the search tree represents a valid isomorphism is either 10^{-2}, 10^{-13}, 10^{-18}, or zero, representing a range of dense and sparse solution spaces. The problem instances vary between searching for one solution and searching for 256 solutions.

We implemented subgraph isomorphism in Natasha, a prototype parallel programming language that supports control abstraction. The Natasha program for subgraph isomorphism, which uses control abstraction and annotations, only runs on the BBN Butterfly. For the remaining machines, we used conditional compilation of a single source code program to mimic the effect of the Natasha compiler and runtime.

In the following discussion, we do not consider filter parallelism at all. Also, we don't examine the tradeoffs involved in the application of word parallelism. In our results, we report the minimum execution time (in seconds) achieved over the entire range of processors on a given machine, including word parallelism in those cases where it helps, and ignoring it in those cases where it does not help.

Tuning to a Particular Class of Inputs

Graph parallelism reduces the time needed to move from the root of the search tree to a leaf, while search parallelism expands the number of paths between the root and leaves that can be considered in parallel. Due to the speculative nature of search parallelism, we would expect it to be most effective when searching for a single solution in a sparse solution space. In a very dense solution space, we would expect every subtree to contain a solution, and therefore we should minimize the time spent on the path from the root node to any leaf using graph parallelism if we only need one solution. Thus, assuming we only require one solution, we might want to use search parallelism for one class of inputs (sparse solution space), and graph parallelism for another (dense solution space).

The results in table 2 confirm this hypothesis. On the 8CE, Butterfly, and Iris, graph parallelism performs better than search parallelism when the solution space is extremely dense (that is, when the probability that a leaf node represents an isomorphism is 10^{-2}). When the solution space is sparse, search parallelism dominates graph parallelism on all three machines. When there are no solutions to be found, the two parallelizations are comparable in performance.

Given that graph parallelism is four times faster than search parallelism in one case, while search parallelism is 67 times faster than graph parallelism in another case, there is clearly enormous benefit to having both parallelizations in the source code. In addition, the ability to move easily from one parallelization to the other is essential if we expect to tune the program to a given class of inputs.

Changing the Problem Instance

When searching for a single solution in a dense solution space, it is best to minimize the time spent on the path from the root to a leaf node using graph parallelism. If we require multiple

solution density:		10^{-2}	10^{-13}	10^{-18}	empty
8CE	graph	0.29	13.80	163.87	0.66
	search	1.12	11.09	3.09	0.59
Butterfly	graph	0.73	33.72	541.51	1.77
	search	2.33	3.76	8.00	1.49
Iris	graph	0.02	1.10	13.25	0.05
	search	0.08	0.74	0.24	0.04

Table 2: Searching for one solution while varying solution space.

solutions however, a single subtree might not contain all the solutions we need, and therefore search parallelism might be of some benefit. The results in table 3 confirm this hypothesis.

solutions desired:		1	128	256
Butterfly	graph	0.73	4.04	7.20
	search	2.33	2.98	3.26
Symmetry	graph	0.32	1.31	2.32
	search	1.32	1.67	1.80
Iris	graph	0.02	0.09	0.17
	search	0.08	0.11	0.13

Table 3: Searching a dense solution space for a varying number of solutions.

We can see that on each of the three machines in table 3, graph parallelism performs much better than search parallelism when only one solution is required. However, if we require 256 solutions, search parallelism performs better. Clearly there is a crossover point, and as seen in table 3, this point is different on the different machines. (The crossover point is below 128 solutions on the Butterfly, and above 128 solutions on the Symmetry and Iris.) Thus, for a given class of inputs, we would like to vary the parallelization depending on the number of solutions required, with the choice of parallelization also depending on the machine.

Porting to a New Machine

When porting a program from one machine to another, we must reconsider all of the architectural assumptions that underlie our choice of parallelization. Any two machines are likely to differ in the number of processors, the speed of the processors, and the cost of communication. These differences may be significant enough that the best parallelization for one machine may not be the best for another.

As seen in table 4, if we require 128 solutions, then whether searching in a sparse or dense solution space, the choice between graph and search parallelism depends on the machine. For this problem, search parallelism is strongly preferred on the KSR-1 and the Butterfly, while graph parallelism is preferred on the 8CE. On the Symmetry, search parallelism is preferred when the solution space is sparse, and graph parallelism is preferred when the space is dense. On the Balance, graph parallelism is preferred when the solution space is dense, and the two parallelizations are comparable when the solution space is sparse. The reverse situation occurs on the TC2000, where search parallelism is preferred when the solution space is sparse, and the two parallelizations are comparable when the solution space is dense.

		8CE	Butterfly	Balance	Iris	Symmetry	TC2000	KSR1
sparse	graph	24.67	75.73	91.67	2.06	29.77	11.04	10.68
	search	36.05	12.60	86.73	2.59	15.84	3.78	2.24
dense	graph	1.06	4.04	4.21	0.09	1.31	0.55	0.46
	search	1.53	2.98	6.52	0.11	1.67	0.52	0.26

Table 4: Searching for 128 solutions.

There are a variety of reasons why each machine performs best using a particular parallelization under particular circumstances, and a complete discussion of these results is beyond the scope of this paper. (See [Crowl *et al.*, 1993] for a complete analysis of subgraph isomorphism and an explanation for each of these performance results, and [Crovella and LeBlanc, 1993] for a description of the tool we developed to aid in this analysis.) Here, we note only that a primary source of overhead under graph parallelization is load imbalance, while the primary source of overhead under search parallelization is wasted speculation. For machines with large numbers of processors, such as the KSR1, the degree of load imbalance under graph parallelism grows quite large, as do the benefits of speculation. In contrast, a machine with a smaller number of processors, such as the Iris, can exploit graph parallelism without introducing significant load imbalance, but does not have enough processors to fully exploit speculation.

Exploiting Multiple Sources of Parallelism

In the experiments described above, we compared the performance of search parallelism versus graph parallelism. Although we used set parallelism in conjunction with search or graph parallelism on some machines, we did not consider search parallelism in tandem with graph parallelism. It is possible that such a combination performs best on all machines, or on all inputs for a given machine.

The results of experiments using this hybrid form of parallelization indicate that it performs best when the solution space is neither very sparse nor very dense.[14] When the solution space is very dense, speculative parallelism is of no help, and therefore the implementation that uses only graph parallelism performs best. When the solution space is very sparse, we have to examine most of the search tree, and the coarse-grain implementation based on search parallelism introduces less

[14]The number of processors assigned to each source of parallelism was determined statically. Varying the number of processors assigned to each source of parallelism would bias the results towards graph or search parallelism, but would not affect our primary conclusion regarding the suitability of the hybrid parallelization.

overhead than the fine-grain implementation based on graph parallelism. These observations hold on all of the machines in our study. Thus, a combination of search and graph parallelism, with or without set parallelism, is not the best parallelization for this problem in all cases.

4.3 Summary of Results

Our experimental results clearly indicate that there is no one best parallelization for subgraph isomorphism. In fact, whenever we vary the input, the problem, or the machine, we may require a new parallelization. This conclusion points out the need for multiple parallelizations, and the ability to change the parallelization easily.

Using control abstraction, we can specify potential parallelism early in the programming process without worrying about a detailed analysis of the costs and benefits associated with each source of parallelism. Later, we can use a combination of analysis and experimentation to tune the implementation of control constructs to specific circumstances. Any implementations created during the course of program development, tuning, or porting can remain in place as documentation of the alternatives, and can be used in future experiments. Thus, control abstraction helps to document the design space of parallelizations, facilitate the analysis of this space, and simplify the task of optimizing the parallelization for specific circumstances.

5 Implementation of Control Abstraction

The examples in the previous sections demonstrate the power of control abstraction for expressing parallelism, and the need for flexibility in tuning parallelizations. Programmers use parallelism to improve performance however, and if control abstraction is to be used frequently, it must be cheap. We now consider the performance implications of control abstraction using our prototype implementation of the Natasha programming language [Crowl, 1991]. Natasha uses the primitive mechanisms for control abstraction described in section 2. The Natasha compiler uses the C language as an intermediate form, and relies on GNU's `gcc` compiler to generate machine code. We have implemented a runtime environment for Natasha on the BBN Butterfly.

5.1 Natasha Compiler

Any programming language that uses closures and operation invocation to implement the most basic control mechanisms might appear to sacrifice performance for expressibility. However, with an appropriate combination of language and compiler, user-defined control constructs can be as efficient as language-defined constructs [Kranz *et al.*, 1986]. Several straightforward optimizations, taken together, can essentially eliminate the execution overhead due to the control abstraction mechanisms. These optimizations include the following.

Invocations as procedure calls: Since an invocation may execute concurrently with its caller after executing its reply, a conservative implementation of invocation requires a separate thread of control for each invocation. This approach is prohibitively expensive. We can reduce this cost by recognizing when an operation has no statements after the reply, and therefore can be implemented as a procedure.

Delayed replies: In those cases where an operation replies early, it is often safe to delay the reply until the invocation completes. This delay admits a procedure implementation for the operation, which exchanges parallelism for the efficiency of sequential execution. We can safely delay a reply if no statement following the reply requires resources (such as synchronization variables) that statements following the invocation release. Since it may be difficult to detect whether delaying a reply is safe, we use two different forms of reply in Natasha: one indicates that the reply may not be delayed in any implementation, and the other indicates that the reply may be delayed in some implementations. An annotation determines whether a reply that may be delayed is actually delayed in a given implementation. This optimization, together with the previous optimization, allow 98% of the (static) invocations in our examples to be implemented as procedure calls.

In-line substitution: Even if we avoid creating a new thread of control for each operation invocation, the overhead of a procedure call for each invocation remains. We can eliminate this overhead by identifying the implementation of operations (either through static typing or type analysis), which makes it possible to use in-line substitution. 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 an equivalent set of machine branch instructions.

Stack allocation of closures: Closures in Smalltalk and Lisp require that their environments remain in existence for the lifetime of the closure. As a result, 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. Fortunately there are several language-dependent approaches to reducing the cost of closure environments. One approach 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]. Another approach restricts the assignment of closures such that the environment is guaranteed to exist (much like Algol68 reference variables). A third approach, which we used in our implementation for expedience, simply defines as erroneous any program that invokes a closure after its environment has been destroyed. Each of these approaches enables stack allocation for closures, significantly reducing the overhead associated with their use.

Even with these optimizations, Natasha suffers a significant performance penalty because we use C as an intermediate language. We therefore apply three additional optimizations to the C code produced by the Natasha compiler:

1. convert `*(&(var))` to `var`[15]
2. represent Natasha activation variables as C activation variables (rather than members of a structure within the activation)[16]
3. represent Natasha global variables as C global variables

Taken together, these optimizations bring the execution time of a Natasha program to within 2% of the execution time of a comparable C program.

[15]This optimization, when performed in isolation, can increase execution time because the compiler no longer eliminates some common sub-expressions.

[16]Both this optimization and the previous one are required to promote Natasha activation variables to registers.

5.2 Natasha Runtime Environment

The Natasha runtime environment is responsible for implementing a shared address space for each application program, and for task creation and scheduling. Our runtime on the Butterfly implements non-preemptive scheduling and blocking synchronization. Since the Butterfly is a distributed-shared-memory multiprocessor, the physical memory associated with the shared address space is distributed among the processors. In addition, each processor has its own scheduling queues. Our implementation exploits the following optimizations in the scheduling of tasks.

Sharing stacks when possible: Since tasks may block, while waiting on other tasks or synchronization variables, we must allocate a separate stack for each task. When a task will not block, because it accesses no synchronization variables and creates no tasks, we avoid allocating a separate stack and use the scheduler's stack to execute the task. The Lynx implementation also uses this technique [Scott, 1987].

Direct access to scheduler queues: We expect the runtime environment to provide implementations of common control constructs that exploit direct access to scheduler queues. For example, our runtime environment on the Butterfly provides both blocked and cyclic implementations of `forall`, thereby allowing the programmer to select a distribution of iterations among processors using annotations. These implementations exploit knowledge of the structure and location of scheduler queues to manipulate those queues directly, resulting in a very efficient implementation.

Task generators: Control operations, such as `forall`, often require the creation of a large number of tasks. Rather than create all tasks at the beginning of the loop, we place a *task generator* on the scheduling queue. Scheduling a task on a processor involves generating the task from the description on the queue. This technique distributes task creation overhead among all processors, and avoids allocating space for tasks that are not able to run due to a lack of processors.

LIFO scheduling: During execution, a program based on our model creates a tree of parallel tasks, where each early reply generates a branch in the tree. A FIFO scheduling strategy roughly corresponds to a breadth-first traversal of this tree of tasks, where the number of active tasks grows very quickly, consuming storage without actually executing. One solution to this problem is to use LIFO scheduling [Halstead, 1990], which encourages a depth-first execution. LIFO scheduling reduces the number of active tasks, and the storage needed to represent tasks. On machines with processor caches LIFO scheduling can also minimize cache corruption between tasks, by returning the processor to a recently executed task before other tasks can evict its data from the local cache.

Avoiding the scheduler: When the currently executing task creates a new task, we block the current task and transfer control to the new task, without invoking the scheduler. This optimization is possible because the original task must block (awaiting an early reply from the new task), and the new task must run next under LIFO scheduling.

Load balancing: It is often advantageous for an idle processor to balance the workload by shifting work from another processor to itself. In our implementation, when a processor has an empty scheduling queue, it may take tasks from another processor's queue. In contrast to Concert Multilisp and Mul-T [Halstead, 1990], we remove tasks from the end of the LIFO queue rather than the front. Tasks at the end of the LIFO queue are likely to be higher in the task tree than tasks at the front of the queue, and therefore are more likely to generate additional tasks for the local work queue.

5.3 Performance Evaluation

To evaluate the effectiveness of our parallel implementation of Natasha, we compared the performance of the Natasha program for Gaussian elimination against an existing hand-tuned parallel program written in C [LeBlanc, 1988]. As seen in figure 5, the unoptimized Natasha program executes between three and four times slower than the hand-tuned C program.

To obtain a more realistic estimate of the performance of production quality implementations, we applied a set of optimizations to the Natasha program, each well within present compiler technology, but beyond the scope of our prototype compiler. We first applied the optimizations described at the end of section 5.1 to the inner loop. Since the lack of these optimizations in the code inhibits induction variable elimination, we also manually applied induction variable elimination. These optimizations had some effect for a small number of processors, but were of no help on the maximum number of processors, where communication dominates.

An examination of the C code produced by the Natasha compiler showed that the translation to C introduces two unnecessary copies of the pivot row when reducing an equation; one from the parameter to the activation record, and another from the activation record to the argument list of the reduction operation. We removed these redundant copy operations. Since our prototype compiler only uses in-line substitution for predefined control constructs, we applied in-line substitution to the closures passed to `triangulate` by hand. Finally, we modified the code to copy only the non-zero portion of the pivot row; this optimization requires a language that can pass sub-arrays as parameters. The execution time of the resulting hand-optimized Natasha program is within 4% of the time required by the original C program.

Since all of the optimizations we applied by hand are well within current compiler technology, we would expect a production-quality implementation of Natasha to be competitive with hand-tuned C programs for parallel programming.

6 Conclusions

In this paper we have demonstrated the benefits of using control abstraction for parallel programming. These benefits include the following:

- Programmers are not limited to a fixed set of control constructs. New constructs can be created and stored in a library for use by others.
- Programmers can use constructs that reflect the potential parallelism of an algorithm, isolating final decisions on parallelism and synchronization within the implementation of constructs, and away from the rest of the algorithm.
- Programmers can use control abstraction to define operations on abstract data types, facilitating representation-dependent parallelism.
- Each control construct can have multiple implementations, corresponding to different parallelizations. In tuning a program for a specific architecture, or in porting a program to a new architecture, programmers can experiment with alternative parallelizations by selecting implementations from a library of control constructs.

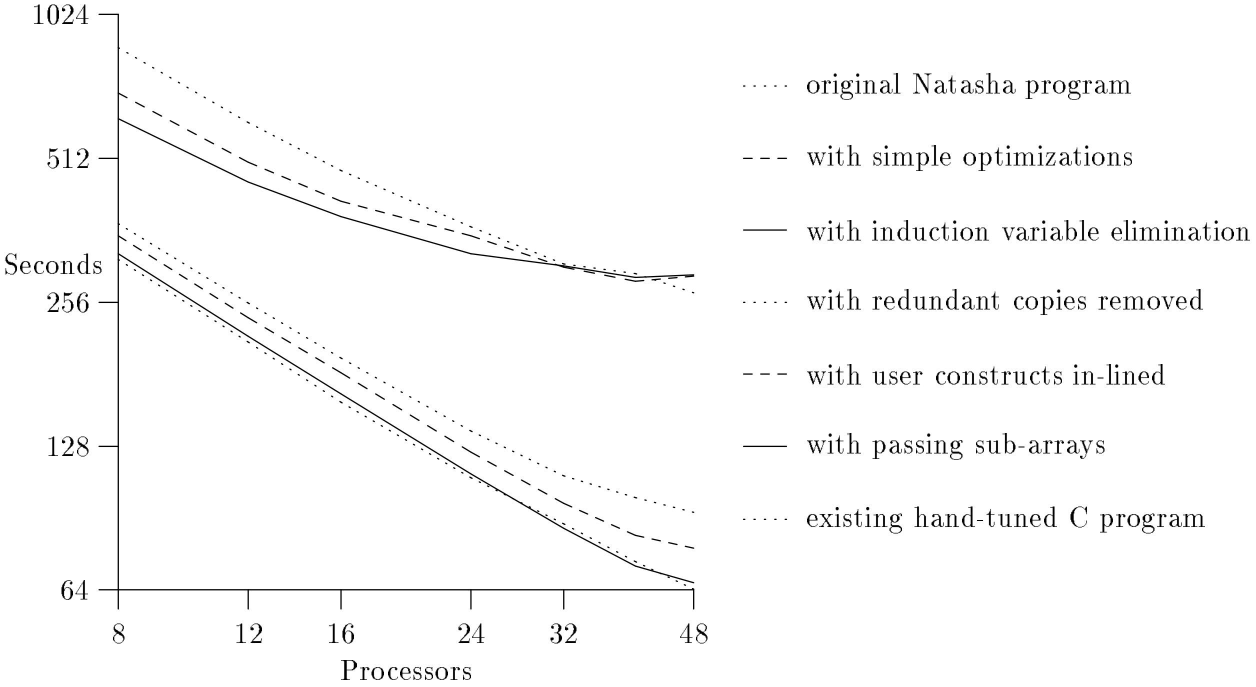

Figure 5: Performance of Natasha Implementation

We presented a notation for specifying control flow in control constructs, introduced a small set of primitive mechanisms for control abstraction, and defined those mechanisms in terms of our notation. We showed how to define and implement new control constructs, verifying that the implementations meet the definitions. We used a number of concrete example programs to illustrate the issues that arise when writing parallel programs with control abstraction, and showed how control abstraction makes it easy to tune programs for a specific architecture. We also described the implementation of a programming language based on our primitive mechanisms for control abstraction, and outlined a number of optimizations that allow an implementation of these mechanisms to be competitive with procedural languages.

When properly employed, control abstraction can greatly reduce the effort needed to make changes to the source code, whether debugging, tuning, porting, or otherwise modifying parallel programs. However, programmers with little or no experience using control abstraction are likely to be unaware of the costs and benefits of control abstraction. Just as data abstraction requires a change in programming methodology, so does the introduction of control abstraction. Our experience with parallel programming using control abstraction has led to the following insights.

Use control abstraction early in the programming process. In principle, abstraction is always good because it delays commitment [Thimbleby, 1988], which localizes the program's assumptions and reduces the effort needed to change a program. In practice, abstraction mechanisms have a cost, and delayed commitment may not be worth this cost if there is an obvious best implementation. Since most sequential machines share the same von Neumann type architecture, there often is a best implementation. In contrast, there are several common type architectures for parallel machines [Snyder, 1986], and the performance of a given exploitation of parallelism may vary widely among these type architectures. When employed early in the design of a program, control abstraction simplifies the process of adapting programs among different type architectures.

Use precise control constructs. When the control constructs used to specify parallelism do not *precisely* express the parallelism appropriate to an algorithm, we must either introduce explicit synchronization to restrict excessive parallelism, or use control constructs that admit less parallelism than the algorithm permits. The definition of a control construct should admit the widest possible range of parallelizations, while each implementation strikes a different balance between the overhead of explicit synchronization and the potential performance benefits of parallelism.

Embed synchronization in control constructs. The need for explicit synchronization depends on the degree of parallelism in the implementation. By embedding synchronization within a control construct, we can select the appropriate synchronization in tandem with the parallelism. If, in the development of a program, it becomes necessary to introduce synchronization into the body of work passed to a control construct, the construct should be redesigned to expose any dependences, so that synchronization can be embedded in the implementation of the construct.

Integrate control and data abstractions. Where appropriate, programmers should use data abstraction and control abstraction together. The designer of a data abstraction should provide a rich set of control abstractions that operate on the data, so as to allow the implementor of the abstraction sufficient latitude to exploit the parallelism inherent in the representation.

Reuse code. A library of correctly-implemented and well-understood data and control abstractions is the programmer's most effective productivity tool. Although we may require an application-specific control construct now and then, we only need to build implementations for the architecture at hand, ignoring many possible sources of parallelism. The set of implementations will naturally expand during program tuning and porting, and each implementation remains available for use in the future. A program's investment in architectural adaptability is primarily in the constructs it uses, and secondarily in the set of implementations for those constructs, since changing a control construct is a serious undertaking, whereas using an alternative implementation of a construct is not.

Experiment with alternative parallelizations. It may be difficult to predict the performance implications of each implementation of a control construct in an application. Fortunately, it is easy to experiment with alternative parallelizations simply by changing the annotations on each construct. In our experiments, we first examined the performance implications of each source of parallelism in isolation (starting with coarse-grain parallelism and then moving towards fine-grain parallelism), so as to understand the tradeoffs involved in each potential source of parallelism. Based on our understanding of these tradeoffs, we examined reasonable combinations of implementations so as to develop a performance model of the interactions between different sources of parallelism. Control abstraction greatly facilitates our ability to conduct the numerous experiments needed to understand the performance implications of all possible implementations over a wide range of machines.

Based on our experiences in implementing a language with control abstraction, and developing a range of application programs on a wide variety of shared-memory multiprocessors, we conclude that the large benefits and modest costs of control abstraction argue for its inclusion in explicitly parallel programming languages.

Acknowledgements

Mark Crovella implemented subgraph isomorphism on the KSR-1 and, together with Michael Scott, was instrumental in the analysis of this application. The authors would like to thank Mark, Michael, Ricardo Bianchini, Tim Budd, Margaret Burnett, Alan Cox, Rob Fowler, Mike Quinn, César Quiroz, and Jack Veenstra for their many helpful comments on this paper. We would also like to thank Argonne National Laboratories for the use of their TC2000, International Business Machines for providing the 8CE, Sequent Computer Systems for providing the Balance and Symmetry, and Donna Bergmark and the Cornell Theory Center for their assistance and the use of their KSR-1. We are also indebted to the anonymous referees, whose many suggestions substantially improved the paper.

References

[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*, pages 42–56, July 1988.

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

[Alverson and Notkin, 1992] Gail A. Alverson and David Notkin, "Abstracting Data-Representation and Partitioning-Scheduling in Parallel Programs," In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 315–338. MIT Press, 1992.

[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.

[Black *et al.*, 1987] Andrew P. Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[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.

[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, 1990] Michael H. Coffin, *Par: An approach to architecture-independent parallel programming*, PhD thesis, University of Arizona, August 1990.

[Coffin, 1992] Michael H. Coffin, *Parallel Programming: A New Approach*, Silicon Press, Summit, New Jersey, 1992.

[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.

[Crovella and LeBlanc, 1993] Mark Crovella and Thomas LeBlanc, "Performance Debugging Using Parallel Performance Predicates," In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, May 1993.

[Crowl *et al.*, 1993] Lawrence Crowl, Mark Crovella, Thomas LeBlanc, and Michael Scott, "Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search," Technical Report 451, Computer Science Department, University of Rochester, April 1993.

[Crowl, 1991] Lawrence A. Crowl, "Architectural Adaptability in Parallel Programming," Technical Report 381, Computer Science Department, University of Rochester, May 1991, Ph.D. Dissertation.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, 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 Springer-Verlag Lecture Notes in Computer Science, pages 161–181, 1990.

[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 Springer-Verlag Lecture Notes in Computer Science, pages 2–57, 1990.

[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.

[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.

[Leasure, 1990] B. Leasure, *PCF Fortran: Language Definition 3.1*, Parallel Computing Forum, Champagne, Illinois, August 1990.

[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.

[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 *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.

[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.

[Polychronopoulos and Kuck, 1987] Constantine D. Polychronopoulos and David J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Software Engineering*, C-36(12), December 1987.

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

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

[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," In *Annual Review of Computer Science*, 1986.

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

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

[Thomas and Crowther, 1988] Robert H. Thomas and Will Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

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