Architectural Adaptability in Parallel Programming via Control Abstraction

Lawrence A. Crowl   Thomas J. LeBlanc

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 359

January 1991

Abstract

Parallel programming involves finding the potential parallelism in an application, choosing an algorithm, and mapping it to the architecture at hand. Since a typical algorithm has much more potential parallelism than any single architecture can effectively exploit, we usually program the parallelism that the available control constructs easily express and that the given architecture efficiently exploits. This approach produces programs that exhibit much less parallelism than the original algorithm and whose performance depends entirely on the underlying architecture. To port such a program to a new architecture, we must rewrite the program to remove any ineffective parallelism and to recover any lost parallelism appropriate for the new machine.

In this paper we show how to adapt a parallel program to different architectures using control abstraction. 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 we choose to exploit without otherwise changing the source code. This approach produces programs that exhibit most of, if not all, the potential parallelism in an algorithm, and whose performance can be tuned for a specific architecture simply by choosing among the various implementations for the control constructs in use.

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

1 Introduction

Algorithms generally contain more *potential* parallelism than any one machine can effectively *exploit*. Although an algorithm may have an efficient realization on a wide range of architectures, including vector processors, bus-based multiprocessors, non-uniform memory access multiprocessors, and distributed memory machines, 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 (other than that it is von Neumann). Parallel programs, on the other hand, typically embed assumptions about the effective granularity of parallelism, the mapping of parallelism to processors, the cost of communication and synchronization, and the distribution of data. When an architecture violates any of these assumptions, the program must be restructured to avoid a potentially serious performance degradation. This restructuring can be complex, because the underlying assumptions are rarely explicit, and the ramifications of each assumption are difficult to discern. We can measure architectural adaptability by the extent of source code changes necessary to adapt a program to an architecture, and the intellectual effort required to select those changes.

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 especially important because we cannot always predict the most efficient parallelization for a given architecture in advance. In addition, porting to a new architecture may require a significant change in parallelization, and a drastic change in any source code developed without architectural adaptability in mind.

Our approach to adaptability requires that a program specify the potential parallelism in an algorithm that an architectures of interest might exploit. Only a subset of the potential parallelism will be realized on a given architecture, but the inclusion of additional potential parallelism in the source code will facilitate future adaptation to another architecture.

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

We are interested in programs written in explicitly parallel programming languages. 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 number 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 user-defined control construct accepts as a parameter a body of code to execute and its execution environment. The implementation of the construct defines 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 user defines the exact semantics of the construct, as well as the implementations.

Much like data abstraction, which hides the implementation of an abstract data type from users of the type, control abstraction hides the exact sequencing of operations from the user of the control construct. 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 without changing the program and thereby achieve architectural adaptability.

This paper describes how to achieve architectural adaptability using control abstraction. Section 2 describes some related approaches to architectural adaptability. Section 3 presents a set of programming language primitives for parallel programming based on control abstraction and provides an introduction to using and defining control constructs. Section 4 uses these primitives to build some common parallel control constructs. Section 5 introduces architectural adaptability via annotations that select among multiple implementations of a control construct. Sections 6 and 7 bring these notions together in extended examples, developing parallel programs for Gaussian elimination and subgraph isomorphism. Section 8 discusses the interaction between synchronization and control abstraction. Section 9 suggests a general methodology for parallel programming with control abstraction. Section 10 describes language implementation strategies for control abstraction that achieve execution efficiency comparable to that of conventional programming languages. Finally, we present our conclusions in section 11.

2 Related Work

In creating a parallel program, a 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 and aid architectural adaptability. In this paper we focus on the first problem, specifying and exploiting parallelism. Our solution to this problem is compatible with techniques developed by others to address mapping [Hudak, 1986; Snyder, 1984], distribution [Coffin and Andrews, 1989; Harrison (Alverson) and Notkin, 1990], and communication [Black *et al.*, 1986].

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

Parallel Function Evaluation Functional programs have no side effects, so expressions may be evaluated in any order. As a consequence, we can evaluate all expressions in parallel, and parallelism is implicit in functional programs. There are two sources of parallelism in function evaluation: parallel evaluation of multiple arguments to a function and lazy evaluation of the value of a function. Owing to the difficulty of automatically finding and exploiting the optimal sources of parallelism in a functional program, several researchers have suggested the use of annotations to specify lazy, eager, parallel, and distributed function evaluation [Burton, 1984; Halstead, 1985; Hudak, 1986].

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

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

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

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

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

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

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

Any single control construct may not easily express all the parallelism in an algorithm, however. Languages that depend on a fixed set of control constructs for parallelism limit their ability to express certain algorithms easily. When the given constructs do not easily express the parallelism in an algorithm, the programmer must either accept a loss of parallelism, or use the available constructs to express excessive parallelism, and then remove the excess using explicit synchronization. The former approach limits the potential parallelism that can be exploited, while the latter approach results in programs that are difficult to adapt to different architectures. In the particular case of Par, programmers must express all parallelism with `co`. 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.

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

Data Abstraction Data distribution and parallelism play an equally important role in architectural adaptability. Communication costs vary significantly across architectures, and the degree of parallelism and the distribution of data among processors determines the need for communication. Indeed, the primary focus of Par and Chameleon is on the use of data abstraction to hide data and processing distributions that may vary across architectures. Likewise, ParAlfl provides facilities for the distribution of data and computation across processors. Par and Chameleon provide the minimal control mechanisms needed to support architectural adaptability via data abstraction; our approach to parallelism via control abstraction is complementary to their approach to data distribution via data abstraction.

3 Parallel Programming Model

This section introduces a parallel programming model and its notation so that we can present concrete examples in our presentation. Our parallel programming model relies on a combination of only four control mechanisms, *operation invocation*, *statement sequencing*, *early reply*, and *first-class closures*, to define sequential and parallel control constructs uniformly. These mechanisms are part of the Matroshka parallel programming model; see [Crowl, 1988] for additional details. With these mechanisms, programmers may build a rich variety of control constructs to represent precisely the parallelism in an algorithm.

Operation Invocation By operation invocation, we refer to either procedure invocation in procedural languages, or to method invocation in object-based languages. In this paper, we use a procedural notation for operation invocation. Operation invocation is synchronous with respect to the caller. That is, the caller waits for the result before proceeding. For example,

```
power ( 3, 4 )
```

computes and returns 3^4. For conciseness, we use a conventional prefix/infix expression notation for sequential data operations, such as integer addition.

Statement Sequencing A sequence of statements defines a total order on statement executions. Notationally, we separate statements by a semicolon, as in the following example:

```
n := 3+4; j := 4*5; k := n+j
```

Early Reply An invocation may reply with a result and then continue executing in parallel with the caller. The caller waits for a reply, but does not wait for termination of the operation. Early reply is the sole source of parallelism in Matroshka. This mechanism is not new [Andrews *et al.*, 1988; Liskov *et al.*, 1986; Scott, 1987], but its expressive power does not appear to be widely recognized. We denote the value-returning reply statement with the keyword `reply` preceding the expression. For example,

```
reply 8
```

Replies that return control, but no value, omit the expression. Each operation or closure may have (and execute) only one reply.

First-Class Closures General control abstraction requires a mechanism for encapsulating and manipulating the body of a control construct. This code must have access to the environment that invokes the control construct. Like Lisp [Steele, 1984], Smalltalk [Goldberg and Robson, 1983], and their derivatives, we use first-class *closures* to capture the code and its environment. Closures may accept parameters and return results. Named procedures are a form of closure, so all claims about closures also apply to procedures.

Closures capture their environment at point of elaboration and may affect variables in their environments that are not visible to the callers of the closures. Closures are reusable. Programmers may invoke closures any number of times. In addition, closures may execute concurrently. The model provides no synchronization between multiple invocations of a closure. Programmers are responsible for ensuring that they invoke closures at the proper time.

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. For notational convenience, when a closure takes no parameters, we omit the parameter list. We also omit the reply when it is the last statement in a closure and it returns no value. For example, we write a closure that accepts an integer parameter and returns twice its value as:

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

This is a similar to a Lisp λ-expression. We use a type syntax similar to Pascal, including reference parameters. The type of this closure is:

```
closure ( i: integer ): integer
```

Given a variable `twice` that references such a closure, we invoke the closure just as we would an operation:

```
twice ( 4 )
```

As an example of the use of closures in a control construct, consider the `for` construct for iteration over an integer range. It takes three parameters: an integer lower bound, an integer upper bound, and a closure that accepts an integer parameter. The definition (as opposed to the implementation) is:

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

An example of its use is:

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

Noting the Partial Order of Execution The presence of an early reply in a closure definition (and also in an operation definition) specifies a partial order of execution, which admits parallelism. For example, given the closure definition,

(*parameter*: *type*) { s_1; ...; s_i; `reply` *expr*; s_j; ...; s_n }

the statements calling the closure

...; s_x; *closure* (*arg*); s_y; ...

result in the following partial order of execution:

$$\ldots \rightarrow s_x \rightarrow \text{evaluate } arg \rightarrow s_1 \rightarrow \ldots \rightarrow s_i \rightarrow \text{evaluate } expr \begin{array}{l} \nearrow \; s_j \rightarrow \ldots \rightarrow s_n \\ \\ \searrow \; s_y \rightarrow \ldots \end{array}$$

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

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

⇂ `for ( lower, upper, work )` → ⇂ `work ( lower )`
↾ `work (` i `)` → ⇂ `work (` $i+1$ `)` [i : `lower` $\leq i <$ `upper`]
↾ `work ( upper )` → ↾ `for ( lower, upper, work )`

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

Conditional Execution Thus far we have presented no mechanism for conditional execution. We adopt the approach of Smalltalk [Goldberg and Robson, 1983] and depend on a *Boolean* type and an `if` operation that conditionally executes a closure. In our case, the operation is:

```
define if ( condition: boolean; body: closure ( ) )
```

⇂ `if (` *true*, `body )` → ⇂ `body ( )`
↾ `body ( )` → ↾ `if (` *true*, `body )`

We invoke this operation just as we would any other. For example, in

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

the assignment executes only when $y > 0$.

Given the `if` operation, we can build many other common control constructs. For example, the `while` construct may have the following recursive implementation:

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

For convenience in the remainder of the paper, we introduce two additional sequential control constructs, `ifelse` and `repeat`. Their implementations in terms of `if` are:

```
implement ifelse ( condition: boolean; affirm, negate: closure ( ) )
{ if ( condition, affirm ); if ( not condition, negate ) }

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

Note that we can pass the closure parameters to `ifelse` directly to `if` — we need not create literal closures for each construct. We expect that compilers will recognize the primitive control constructs (such as those found in most sequential languages) and generate code for them directly.

Synchronization We have presented no mechanism for synchronization other than that implicit in an operation invocation waiting for the reply. We assume that any languages based on our model will provide some primitive synchronization mechanism(s). We do not assume any particular synchronization operations in our examples; atomic memory accesses, atomic instructions like test-and-set, or higher-level synchronization primitives are all acceptable.

4 Building Common Control Constructs

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

4.1 Fork and Join

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

```
define type forkjoin;
define fork ( var mailbox: forkjoin; work: closure ( ): integer );
define join ( var mailbox: forkjoin ): integer
```

$\downarrow$`fork ( mailbox, work )` $\rightarrow$ $\downarrow$`work ( )`
$\uparrow$`work ( )` $\rightarrow$ $\uparrow$`join ( mailbox )`

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

$\uparrow$`fork ( mailbox, work )` $\rightarrow$ $\downarrow$`join ( mailbox )`

These partial orders permit parallel execution. However, they do not guarantee parallelism because the rules state no order between the reply from `fork` and the invocation of `work`. The additional order:

$\uparrow$`fork ( mailbox, work )` $\rightarrow$ $\downarrow$`work ( )`

which states that `fork` must reply before invoking `work`, would guarantee concurrent execution. We clarify the reason for this omission in section 5.

Assume a `power` operation that accepts two integers and returns the first argument raised to the power given by the second argument. We can use the definition of `fork` and `join` to evaluate two invocations of the `power` operation in parallel.

```
var mailbox: forkjoin;
var n, sum: integer;
fork ( mailbox, { reply power ( 3, 4 ) } );
n := power ( 5, 6 );
sum := n + join ( mailbox )
```

The following implementation of `fork` and `join` illustrate the use of early reply and explicit synchronization to achieve parallelism. This implementation is based solely on the mechanisms described in section 3, with the addition of atomic Boolean reads and writes. Busy-waiting synchronizes the two computations. We could easily change this implementation to use semaphores for synchronization and avoid busy waiting.

```
implement forkjoin = record ready: boolean; result: integer end

implement fork
```

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

```
    ( var mailbox: forkjoin; work: closure ( ): integer )
{ mailbox.ready := false;
  reply; --- no reply value, caller continues
  forkjoin.result := work ( );
  forkjoin.ready := true }

implement join ( var mailbox: forkjoin ): integer
{ repeat ( { reply not mailbox.ready } ); --- busy wait
  reply mailbox.result }
```

4.2 Cobegin

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

```
define cobegin ( work1, work2: closure ( ) )
```

```
↓ cobegin ( work1, work2 ) → ↓ work1 ( )
↓ cobegin ( work1, work2 ) → ↓ work2 ( )
↑ work1 ( ) → ↑ cobegin ( work1, work2 )
↑ work2 ( ) → ↑ cobegin ( work1, work2 )
```

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

```
↓ work1 ( ) → ↑ work2 ( )
↓ work2 ( ) → ↑ work1 ( )
```

state that `cobegin` must invoke both closures before waiting on the replies.

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

```
var n, m, sum: integer;
cobegin ( { n := power ( 3, 4 ) }, { m := power ( 5, 6 ) } );
sum := n + m
```

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

```
implement cobegin ( work1, work2: closure ( ) )
{ var mailbox: forkjoin;
  fork ( mailbox, work1 ); work2 ( ); join ( mailbox ) }
```

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

4.3 Forall

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

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

$\downarrow$ `forall ( lower, upper, work )` $\rightarrow \downarrow$ `work (` i `)` $\quad [i:$ `lower` $\leq i \leq$ `upper`$]$

$\downarrow$ `work (` i `)` $\rightarrow \downarrow$ `work (` $i+1$ `)` $\quad [i:$ `lower` $\leq i <$ `upper`$]$

$\uparrow$ `work (` i `)` $\rightarrow \uparrow$ `forall ( lower, upper, work )` $\quad [i:$ `lower` $\leq i \leq$ `upper`$]$

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

$\downarrow$ `work (` i `)` $\rightarrow \uparrow$ `work (` j `)` $\quad [i,j:$ `lower` $\leq i \leq$ `upper` $\wedge$ `lower` $\leq j \leq$ `upper`$]$

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

We use `cobegin` and recursion to build a parallel *divide-and-conquer* implementation of `forall`.

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

This implementation executes each iteration of `forall` in parallel, and therefore would only be appropriate in cases where the granularity of parallelism supported by the architecture was well matched to the granularity of each iteration. Otherwise, it would be better to use an alternative parallel implementation that creates a fixed number of tasks, each of which executes a set of iterations. The degree of parallelism provided by this alternative implementation is easily changed, but cannot be selected using annotations for operation implementations alone.

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

[3] Unlike CLU, our emphasis is on the separation of semantics and implementation for general control constructs, rather than the ability to iterate over the values of any abstract type.

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

5 Architectural Adaptability

In implementing an algorithm, we must choose a subset of the potential parallelism to exploit. There are two reasons why we might want to change the parallelism we actually exploit.

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

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

Our approach uses control abstraction to define many different control constructs. The *algorithm* determines the control constructs used to represent potential parallelism; the *architecture* determines the implementations used to exploit parallelism.

5.1 Multiple Implementations

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

Our rules for each of the control constructs in section 4 deliberately left the partial orders underspecified, so as to admit either a parallel or sequential implementation. We complete the example constructs in section 4 by providing alternative implementations here. To distinguish each implementation, we annotate it with a descriptive identifier that follows the operation identifier. We assume programmers will annotate each implementation of a control construct with a name that describes the degree of parallelism exploited by the implementation. For example, our parallel divide-and-conquer implementation of `forall` from the previous section would be annotated as follows:

```
implement forall $DIVIDED ( ...
```

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

```
implement forall $GROUPED ( ...
```

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

```
implement type forkjoin $SEQUENTIAL = record result: integer end

implement fork $SEQUENTIAL
    ( mailbox: forkjoin; work: closure ( ): integer )
{ mailbox.result := work( ) }  -- caller waits for work to finish

implement join $SEQUENTIAL ( mailbox: forkjoin ): integer
{ reply mailbox.result }
```

This sequential implementation of `forkjoin` could be used to produce a sequential implementation of `cobegin`. Alternatively, we could modify the implementation of `cobegin` to execute the two statements in sequence without the use of `forkjoin`.

```
implement cobegin $SEQUENTIAL ( work1, work2: closure ( ) )
{ work1 ( ); work2 ( ) }
```

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

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

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

5.2 Selecting Implementations

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

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

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

```
power $PARALLEL ( 3, 4 )
```

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

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

In the following example we illustrate the use of annotations to select a particular parallelization for Quicksort. There are two potential sources of parallelism we consider. When the 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.

```
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 `$PARALLEL` annotation to select the parallel implementation of the second `cobegin`) and chose not to exploit the finer-grain parallelism available during partition. We could experiment with fine-grain parallelism by simply changing the `$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, if not impossible, to find automatically.

The control constructs of section 4 have several possible implementations. We may adapt many parallel programs simply by choosing to use different implementations of these constructs on different architectures.

6 Gaussian Elimination Example

We will use Gaussian elimination (without pivoting) as an extended example of using control abstraction for architectural adaptability. Gaussian elimination is a well-known algorithm, has nontrivial synchronization constraints, and admits several different exploitations of parallelism. Our goal is to create a single source program that represents these different exploitations, each of which can be selected by an appropriate choice of annotations, and thereby duplicate previous extensive experience in the development and tuning of parallel Gaussian elimination on the BBN Butterfly [Crowther *et al.*, 1985; Thomas, 1985; LeBlanc, 1986; LeBlanc, 1988] without the same substantial effort.

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

We present this example as a sequence of programs derived from the standard sequential algorithm, reflecting our earlier experiences with this application. Later, in section 9, we propose a methodology that avoids the intermediate steps in this sequence and proceeds directly to the final form.

6.1 The First Cut

Our first attempt is based on the standard sequential algorithm for upper triangulation.[7]

```
var system: array [ 1..SIZE ] of array [ 1..SIZE ] of float;
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]
```

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

```
                -:= fraction * system[pivot][variable] } ) } ) } )
```

One straightforward parallel implementation of this algorithm parallelizes the two inner loops with `forall`.[8] Section 4 showed that the `forall` construct has both a parallel and sequential implementation. By using annotations to select a parallel implementation for both loops, we create an extremely fine-grain parallel implementation.

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

Vector processors could exploit the parallelism in the inner loop by invoking vector instructions, rather than using the parallel implementation of `forall`. On a vector processor we would expect our compiler to recognize a `$VECTOR` annotation and produce vector instructions for the innermost loop.[9] To port the program to a vector multiprocessor, such as the Alliant FX, we would use both a parallel implementation for the outer `forall` and a vector implementation for the inner `forall`.

The Butterfly lacks vector processors, and could not profitably exploit the parallelism in the inner loop. Therefore, we can select an implementation that does not attempt to exploit fine-grain parallelism by choosing the `$SEQUENTIAL` annotation for the inner loop. The resulting program exhibits a series of phases separated by the selection of a pivot. This was precisely the first program developed in our earlier work [LeBlanc, 1988]. Experimentation with this version of the program highlighted the time processors spent waiting for other processors to complete each phase. These empirical results led us to develop an implementation based on the synchronization constraints for the problem.

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

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

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

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

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

```
var system: array [ 1..SIZE ] of array [ 1..SIZE ] of float;
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] } )
```

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

6.2 A New Control Construct

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

```
define triangulate
    ( size: integer; work: closure ( pivot, reduce: integer ) )
```

$$\downarrow \texttt{triangulate (size, work)} \rightarrow \downarrow \texttt{work (} \ i, \ j \ \texttt{)} \qquad [i, j : 1 \leq i < j \leq \texttt{size}]$$
$$\uparrow \texttt{work(} \ i, \ j \ \texttt{)} \rightarrow \downarrow \texttt{work (} \ k, \ j \ \texttt{)} \qquad [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}]$$
$$\uparrow \texttt{work(} \ i, \ j \ \texttt{)} \rightarrow \downarrow \texttt{work (} \ j, \ k \ \texttt{)} \qquad [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}]$$

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

```
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 ) } ) } ) }
```

By substituting `forall $DIVIDED` for `forall $SEQUENTIAL` we get `triangulate $PHASED`, which exploits the same parallelism as the earlier phased version of the program. In addition, we can also substitute `forall $GROUPED` for `forall $SEQUENTIAL` to obtain a `triangulate $PHASED_GROUPED`.

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

```
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 more parallelism than `triangulate $PHASED`, but may have higher execution overhead because of the need to accommodate synchronization. As earlier, we can substitute `forall $GROUPED` for `forall $DIVIDED` to obtain a `triangulate $SYNCHED_GROUPED`.

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 float;
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 the previous parallelizations of this problem. Programmers can select twenty different implementations of this program by varying the two annotations to select a divide-and-conquer, grouped, sequential, or vector implementation of `forall`, and a synchronized divide-and-conquer, synchronized grouped, phased divide-and-conquer, phased grouped, or sequential implementation of `triangulate`. Our experience has shown that `triangulate $SYNCHED_GROUPED` and `forall $SEQUENTIAL` is the most efficient implementation on the Butterfly. We expect that `triangulate $SYNCHED_GROUPED` and `forall $VECTOR` would be the most appropriate for the Alliant. This same program has been ported to a Sun workstation by selecting `triangulate $SEQUENTIAL` and `forall $SEQUENTIAL`. The key to the adaptability in our solution is the introduction of an algorithm-specific control construct.

7 Subgraph Isomorphism Example

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

A graph isomorphism is a mapping from each vertex in one graph to a unique vertex in the second, such that if two vertices are connected in the first graph then their corresponding vertices in the second graph are also connected. In subgraph isomorphism, the second graph is an arbitrary subset of a larger graph.

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

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

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

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

```
implement graph = array of set of integer;
var small_neighbors, large_neighbors: graph
```

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

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

```
implement tree_node = array of set of integer;
var node: tree_node
```

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

Searching Possibilities The coarsest grain of parallelism arises when searching among the various possibilities for a given small vertex. Given a set of possibilities in the set `node[current_small]`, we need to examine each postulated mapping. Using a language with the typical fixed control constructs, we would write:

```
implement search ( current_small: integer; node: tree_node )
{ forall ( 1, maximum_large, ( large_vertex: integer )
    { if ( large_vertex in node[current_small],
        { examine( current_small, large_vertex, node ) } ) } }
```

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

The problem with the above code is that we wish to iterate over the elements of the set, but the `forall` forces us to iterate over the representation for the set and then test for membership. A better approach is to combine data abstraction and control abstraction and define an iterator for sets, as we would in CLU [Liskov *et al.*, 1977].[10] This enables us to specify precisely that the parallelism is over actual elements, and not over potential elements. We define a `forall_elements` operation that executes a closure (or operation) for each element of the set.

```
define forall_elements
    ( members: set of integer; work: closure ( member: integer ) )
```

$\downarrow$ `forall_elements ( members, work )` $\rightarrow$ $\downarrow$ `work (` i `)` $\qquad [i : i \in$ `members`$]$

$\uparrow$ `work (` i `)` $\rightarrow$ $\uparrow$ `forall_elements ( members, work )` $\qquad [i : i \in$ `members`$]$

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

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

```
implement search ( current_small: integer; node: tree_node )
{ forall_elements ( node[current_small], ( postulate: integer )
    { examine( current_small, postulate, node ) } ) }
```

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

This representation is clearer and potentially more efficient, but it requires a mechanism to define control abstractions that interact with data abstractions in order to build the iterator. The closure mechanism serves this need.

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

Examining a Mapping The next task is to examine a single proposed mapping and propagate the constraints of that mapping. The first task is to enforce the minimal constraints — the small vertex may map to no other and no other small vertex may map to the chosen large vertex. Next, we check to see if the incomplete isomorphism is a leaf in the search. If so, we report the isomorphism,[11], otherwise we apply better constraints.

```
implement examine
    ( current_small, current_large: integer; node: tree_node )
{ minimal_constraints ( current_small, current_large, node );
  ifelse ( current_small = maximum_small,
    { report_possible_isomorphism ( node ) },
    { constrain ( current_small, current_large, node ) } ) }
```

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

```
implement constrain
    ( current_small, current_large: integer; var node: tree_node )
{ cobegin ( { distance_filter ( current_small, current_large, node ) },
            { connect_filter ( current_small, current_large, node ) } );
  if ( no_empty_mapping ( node ),
      { search ( current_small+1, node ) } ) }
```

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

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

Distance Filter Two small vertices separated by a distance x cannot map to two large vertices separated by a distance $y > x$.[12] We rely on two precomputed arrays, `small_distance` and `large_distance`, to retrieve distance information. Using the current small vertex and its postulated mapping as one vertex of each pair, we successively choose each small vertex as the second small vertex and remove those possible mappings with an inconsistent distance. Using `forall_elements`, the operation is:

```
implement distance_filter
    ( current_small, current_large: integer; var node: tree_node )
{ forall ( 1, maximum_small, ( other_small: integer )
    { forall_elements ( node[other_small], ( other_large: integer )
        { if ( small_distance[current_small,other_small]
               < large_distance[current_large,other_large],
             { remove_element ( other_large, node[other_small] ) }
) } ) } ) }
```

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

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

This definition leaves room for several different implementations.

Given `forall_elems_cond`, the distance filter becomes:

```
implement distance_filter
    ( current_small, current_large: integer; var node: tree_node )
{ forall ( 1, maximum_small, ( other_small: integer )
```

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

```
  { forall_elems_cond ( map[other_small],
      ( other_large: integer )
      { reply small_distance[current_small,other_small]
              < large_distance[current_large,other_large] },
      ( other_large: integer )
      { remove_element ( other_large, map[other_small] ) } ) } ) }
```

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

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

```
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 ]
```

Its implementation must synchronize with other operations on the set.

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

```
implement distance_filter
    ( current_small, current_large: integer; var node: tree_node )
{ forall ( 1, maximum_small, ( other_small: integer )
    { remove_element_cond ( &node[other_small],
        ( other_large: integer )
        { reply small_distance[current_small,other_small]
                < large_distance[current_large,other_large]
} ) } ) }
```

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

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

```
implement connect_filter
    ( current_small, current_large: integer; var node: tree_node )
{ forall_elements ( small_neighbors[current_small],
    ( other_small: integer )
    { assign_intersection( node[other_small],
                           large_neighbors[current_large] } ) } ) }
```

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

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

```
implement assign_intersection
    ( var members: set of integer; others: set of integer )
{ remove_element_cond ( members, ( member:integer )
    { reply not element_of_set ( member, others ) } ) }
```

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

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

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

the implementation of operation	may annotate the invocation of	with any of the annotations
`search`	`forall_elements`	`$SEQUENTIAL $GROUPED $DIVIDED`
`constrain`	`cobegin`	`$SEQUENTIAL $PARALLEL`
`distance_filter`	`forall`	`$SEQUENTIAL $GROUPED $DIVIDED`
	`remove_elem_cond`	`$SEQUENTIAL $VECTOR`
`connect_filter`	`forall_elements`	`$SEQUENTIAL $GROUPED $DIVIDED`
	`assign_intersection`	`$SEQUENTIAL $VECTOR`

Selecting combinations of these annotations provides us with 216 possible implementations of subgraph isomorphism. Through the use of iterators, conditional iterators, and conditional data operations, this example shows how data and control abstraction interact to provide powerful mechanisms for representing and exploiting parallelism.

8 Synchronization and Control

The presence of parallelism in a program generally implies the presence of synchronization. When we introduce parallelism, we must also introduce synchronization. Ideally, we select synchronization with the same mechanism that exploits parallelism. Parallel programs exhibit two types of synchronization: *data synchronization* ensures consistent access to data by independent threads of control; *control synchronization* coordinates between threads created to perform some work in parallel. In particular, synchronization that supports a data dependence is control synchronization. As in Multilisp [Halstead, 1985], we assume that data synchronization is embedded in data abstractions. In this section, we show how control abstraction can enable simultaneous selection of parallelism and control synchronization, as well as accommodate data dependence.

The technique we use to select parallelism and synchronization simultaneously is to define control constructs whose semantics (partial orders of execution) include the necessary synchronization. Each implementation for a construct then embeds the synchronization appropriate to its exploitation of parallelism. An implementation that does not exploit parallelism need not include synchronization.

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

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

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

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

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

This generator provides the parallelism of our `triangulate` construct, but without the synchronization constraints. As a result, the Uniform System program included explicit synchronization within the body of the work.[13] Gaussian elimination using `GenOnHalfArray` looks like this:

```
var system: array [ 1..SIZE ] of array [ 1..SIZE ] of float;
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] } );
    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 was a reasonable one. Nevertheless, a more efficient implementation would have been possible had the correct control construct been available or easily created. With control abstraction, we can build constructs that contain the necessary synchronization.

Explicit Versus Implicit Synchronization In the implementation of a control construct, we often have a choice between relying on the synchronization implicit in other control constructs or using explicit synchronization. There is no single resolution of this

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

choice for all cases. For example, the synchronization implicit in the outer loop of our phased implementation of Gaussian upper triangulation unnecessarily limits the amount of parallelism in the program. On the other hand, some of the explicit synchronization used in the Uniform System program is both expensive and unnecessary. The `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. It also uses a `for` loop to serialize the application of pivots to a single equation, in place of explicit synchronization in the Uniform System program.

Splitting Closures In the above example, we were able to concentrate solely on the partial order rules to derive a new control construct and embed synchronization within the construct. We may not always be able to do so. For example, consider a sequential loop of the form:

```
for ( 1, N, ( i: integer )
  { statement list 1; x := a[i]; a[i+1] := y; statement list 2 } )
```

This loop has a loop-carried data dependence between iteration i and iteration $i+1$. We cannot use `forall` to specify parallelism because we would violate the dependence. One possible approach is to insert explicit synchronization around the statements containing the data dependence. Unfortunately, the presence of synchronization within the body of the loop would then be separate from the implementation of the loop, which is where we choose whether to exploit parallelism. To apply our general technique of moving the synchronization into a control construct, we must split the body of the loop and expose the dependence in the definition of the construct. We create a construct that accepts the loop in three pieces, corresponding to the statements that can execute in parallel before and after the data dependence, and the statements containing the data dependence.

```
define forall3
    ( lower, upper: integer;
      head, body, tail: closure ( iteration: integer ) )
```

$\downarrow$ `forall3 ( lower, upper, head, body, tail )` $\rightarrow$ $\downarrow$ `head` (i) $\quad [i:$ `lower` $\leq i \leq$ `head`$]$

$\uparrow$ `head` (i) $\rightarrow$ $\downarrow$ `body` (i) $\quad [i:$ `lower` $\leq i \leq$ `head`$]$

$\uparrow$ `body` (i) $\rightarrow$ $\downarrow$ `tail` (i) $\quad [i:$ `lower` $\leq i \leq$ `head`$]$

$\uparrow$ `body` (i) $\rightarrow$ $\downarrow$ `body` ($i+1$) $\quad [i:$ `lower` $\leq i <$ `head`$]$

$\uparrow$ `tail` (i) $\rightarrow$ $\uparrow$ `forall3 ( lower, upper, head, body, tail )` $\quad [i:$ `lower` $\leq i \leq$ `head`$]$

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

```
forall3 ( 1, N, ( i: integer ) { statement list 1 },
                ( i: integer ) { x := a[i]; a[i+1] := y },
                ( i: integer ) { statement list 2 } )
```

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

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

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

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

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

9 Programming Methodology

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

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

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

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

Use Precise Constructs When the control constructs we use to specify parallelism do not *precisely* express the parallelism appropriate to an algorithm, we must introduce explicit synchronization to restrict excessive parallelism or we must accept less parallelism than the algorithm permits.

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

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

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

Expose Data Dependences Occasionally, the natural expression of control and its work places a data dependence deep within the body of a loop, rather than at the beginning or end. If we follow our previous advice and avoid explicit synchronization, this dependence forces us to choose a control construct that provides more synchronization than the algorithm actually requires. The solution is to break the body into separate bodies and then use a construct that handles the multiple bodies. This more complex construct is also more precise, which gives us more flexibility in exploiting parallelism.

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

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

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

10 Implementation

We showed the importance of control abstraction in parallel programming, and how to exploit different grains of parallelism by selecting an appropriate implementation for each control construct. Although descriptive power is an important property, programmers use parallelism to improve performance. Any programming language that uses closures and operation invocation to implement the most basic control mechanisms might appear to sacrifice performance for expressibility. With an appropriate combination of language and compiler, however, user-defined control constructs can be as efficient as languaged-defined constructs. We now describe straightforward optimizations that reduce the execution cost of these mechanisms.

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

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

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

In-line substitution is especially important for the efficient execution of sequential control constructs. When the compiler can determine the implementation of a construct statically, it can replace the invocation with the implementation, and propagate the closure parameter through to its use. Using this technique, we can convert control constructs using late replies into equivalent machine branch instructions.

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

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

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

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

The typical solution to this problem is to use a LIFO scheduling queue [Halstead, 1990], which encourages a depth-first execution and the number of active nodes is relatively small. When a processor has an empty scheduling queue, it takes tasks from other processor's queues. In contrast to Concert Multilisp and Mul-T, we take tasks *least* recently enqueued rather than *most* recently enqueued. This provides minimal impact on the locality of busy processors.

Using these optimizations, our prototype implementation of Matroshka [Crowl, 1988] produces sequential code comparable to that produced by a C compiler without optimization. Our prototype performs only the most basic optimizations and Matroshka programs execute at half the rate of comparable C programs compiled with an optimizing compiler. Four additional low-level optimizations bring Matroshka execution times to within 2% of comparable C programs. We expect that a production compiler for Matroshka would be competitive with an optimizing C compiler.

11 Conclusions

Previous approaches to architectural adaptability separate potential parallelism from exploited parallelism via multiple implementations for predefined control constructs or mechanisms. We extend this work with control abstraction, which enables multiple implementations for user-defined control constructs.

With control abstraction, programmers are not limited to a fixed set of control constructs. Users can create new constructs that express arbitrary partial orders of invocations and store them in a library for use by others. We presented a model of parallel programming based on a small set of primitive mechanisms for control abstraction and showed how the model can directly implement common parallel control constructs.

With the ability to define algorithm-specific control constructs, we can more precisely represent the potential parallelism within an algorithm. Each control construct can have multiple implementations, each of which exploits a different subset of the potential parallelism defined by the construct. Selecting different implementations of a construct at its points of use exploits different sources of parallelism within a program. By embedding synchronization in the implementation of control constructs, separate from the program logic, programmers select parallelism and synchronization simultaneously.

We showed how to use control abstraction to achieve architectural adaptability in explicitly parallel programs. In developing adaptable programs, programmers must identify potential control constructs, ensuring that they encapsulate any necessary synchronization. Programmers adapt parallel programs by selecting an implementation for each use of a construct and then experimentally measuring the effect. Programmers can choose among existing implementations of a construct or build new implementations as needed. The set of implementations will expand during program tuning and porting, leaving different exploitations documented within the source. In addition, we have presented several optimizations that facilitate an efficient implementation of control abstraction. Based on our experience, we believe the benefits and reasonable cost of control abstraction argue for its inclusion in explicitly parallel programming languages.

Acknowledgements

We thank Alan L. Cox, Robert J. Fowler, César A. Quiroz, Michael L. Scott and Jack E. Veenstra for their many helpful comments during the development of this 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 — Parallel Programming: Experience with Applications, Languages and Systems*, pages 42–56, July 1988, appeared in ACM SIGPLAN Notices 23(9), September 1988.

[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.

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

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

[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 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.

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

[Crowl, 1988] Lawrence A. Crowl, "A Uniform Object Model for Parallel Programming," In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 25–27, September 1988, appeared in ACM SIGPLAN Notices 24(4), April 1989.

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

[Fowler *et al.*, 1988] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on

Large-Scale Multiprocessors," In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, May 1988.

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

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

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

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

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

[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, in SIGPLAN Notices 21(7), July 1986.

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

[LeBlanc, 1986] Thomas J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986, also appeared as Butterfly Project Report 3, Computer Science Department, University of Rochester, January 1986.

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

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

[Liskov *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.

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

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

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

[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," Technical Report 86–03–04, Department of Computer Science, University of Washington, Seattle, Washington, March 1986.

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

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

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

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

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

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