OREGON STATE
UNIVERSITY
COMPUTER
SCIENCE
DEPARTMENT

Control Abstraction in Parallel Programming Languages

Lawrence A. Crowl
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Thomas J. LeBlanc
Department of Computer Science
University of Rochester
Rochester, New York 14627-0226

91-80-3

Control Abstraction in Parallel Programming Languages

Lawrence A. Crowl Thomas J. LeBlanc*

Technical Report 91-80-3

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

November 1991

Abstract

Control abstraction is the process by which programmers define new control constructs by specifying an ordering of statement execution. Using control abstraction, we can create new control constructs for parallel programming, separate the specification of parallelism and synchronization from the rest of the application code, and vary the paralellism exploited during execution by selecting alternative implementations of control constructs.

This paper argues for the inclusion of control abstraction in parallel programming languages by demonstrating that the benefits of control abstraction far outweigh the costs. We present a notation for precisely defining the meaning of a parallel control construct; and use that notation in the definition a small set of primitive mechanisms for parallel programming with control abstraction. We show how to define and implement new parallel control constructs using these primitive mechanisms, and provide examples of application-specific control constructs used in real programs. Finally, we describe techniques that can be used to implment a parallel programming language based on control abstraction, and demonstrate the efficiency of those techniques in an implementation on the BBN Butterfly multiprocessor.

Keywords: parallel programming languages, control abstraction, architectural adaptability, closures, early reply, Matroshka

---

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.

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

1 Introduction

Sequential programming languages use sequencing, repetition, and selection to define a total ordering of statement executions in a program. Parallel programming languages[1] use additional control flow constructs, such as `fork`, `cobegin`, or parallel `for` loops, to introduce a partial order on statement executions, which admits a parallel implementation. Since parallelism is primarily an issue of control flow, the control constructs provided by the language can either help or hinder attempts to express and exploit parallelism.

Control constructs typically define an ordering on statement execution, without regard to a specific implementation of that ordering. The specification of a control construct at language-definition time is an example of *control abstraction*. More generally, control abstraction is the process by which programmers specify a statement ordering (parameterized with respect to the statements being ordered) separately from an implementation of that ordering. A *control construct* is the result of that process.

Although the benefits of control abstraction in sequential programming are widely recognized, particularly in the implementation of abstract data types, few parallel programming languages support control abstraction, and the benefits of control abstraction in parallel programming are not recognized.

- With control abstraction, programmers can extend the set of common control constructs without changing the language definition or implementation. Given the importance of control flow in parallel programming, it seems premature to base a language on a small, fixed set of control constructs. With control abstraction, a large set of control constructs becomes feasible (even when based on a very small set of primitive mechanisms) because they may be developed as needed and placed in a library.

- Control abstraction separates the definition and use of a control construct from its implementation, which enables parallel control constructs to have several different implementations, each exploiting a different amount of parallelism. Programmers may easily choose among these implementations at the points where constructs are used, thus tuning the application to a given architecture.

- Programmers may define application-specific control constructs that provide precisely the parallelism their algorithms admit, no more and no less. By defining a construct that provides no less parallelism than can be used, programmers have maximum flexibility in adapting to different architectures. By defining a construct that provides no more parallelism than can be tolerated, programmers avoid mixing explicit synchronization with program logic.

- Programmers may associate control operations with abstract data structures, so that the generation and distribution of parallelism cooresponds to the generation and distribution of data. Association of control with data also increases the accuracy of programs in representing their potential parallelism, leaving more room for alternate implementations and architectural adaptability.

---

[1] Our focus is on explicitly parallel imperative programming languages, such as SR [Andrews *et al.*, 1988] and Ada [U. S. DoD, 1983].

Control abstraction is not free. We have to consider both its programming costs and its implementation costs. Section 3 presents a notation for precisely defining the partioal order of a parallel control construct. Section 4 introduces our particular mechanisms for control abstraction, defines these mechanisms in terms of our notation, and argues that they are sufficient. These mechanisms are part of the Matroshka programming language [Crowl, 1991], which we designed and have implemented on the BBN Butterfly and Alliant FX multiprocessors. Section 5 gives simple, straightforward examples of how to define and implement common parallel control constructs, thus indicating that programming costs are reasonable. In the process, we show that the partial order of the implementations of these constructs satisfy the partial order of the constructs' definitions. Section 6 gives examples of how application-specific control constructs can improve the clarity, efficiency and portability of parallel programs. Section 7 presents a set of optimizations that enable Matroshka to be nearly as efficient as compiled languages. We compare the performance of a Matroshka program with that of a C program on the BBN Butterfly. We conclude in section 8 that the benefits of control abstraction to parallel programming far outweigh its costs and that parallel programming languages should support control abstraction.

2 Related Work

Hilfinger [1982] provides a short history of major abstraction mechanisms in programming languages, from procedure and variable abstraction in Fortran, through data structure abstraction in Algol68 and Pascal, to data type abstraction in Alphard, CLU, and Euclid. However, Hilfinger does not mention control abstraction, even though significant mechanisms for control abstraction are present in Lisp [Steele, 1984]. In Lisp, control abstraction is present to enhance the expressiveness of the language.

Control abstraction has also been used in sequential languages designed to support data abstraction. For example, CLU iterators [Liskov *et al.*, 1977] (or generators) are a form of control abstraction intended to support abstract data structures. With iterators, the user of an abstract structure can operate on the elements of the structure without knowing the representation of the structure. In CLU, and other languages for data abstraction, control abstraction plays a secondary role to the specification and representation of data abstractions.

Given that parallelism is a form of control flow, control abstraction is particularly important for parallel programming, in part because control more directly affects performance. Yet, to our knowledge, only those parallel programming languages that inherit control abstraction from a parent sequential language support it. For example, both Multilisp [Halstead, 1985] and Paralation [Sabot, 1988] use Lisp closures in the implementation of the parallel programming constructs presented to users, but their developers have not argued the benefits of control abstraction as a parallel programming tool.

One of the primary benefits of control abstraction is that it separates the use of control from its implementation. This separation enables multiple implementations of a construct, each exploiting different amounts of parallelism. This separation has been used for architectural adaptability in Par [Coffin and Andrews, 1989] and Chameleon [Alverson, 1990; Alverson and Notkin, 1991]. Par is a programming language that admits multiple implementations of the `co` statement, a type of parallel for loop. Chameleon is set of C++ classes for implementing task generators and schedulers. Neither Par nor Chameleon support general control abstraction however, since both lack a mechanism

similar to closures. We present the case for using general control abstraction to achieve architectural adaptability in parallel programs in [Crowl and LeBlanc, 1991] and [Crowl, 1991].

3 Defining Control Constructs

In order to use a control construct, we must know what it does. This section introduces a notation for precisely defining the meaning of a control construct.

A control construct defines an order of execution for a set of compound statements passed as parameters to the control construct. Each compound statement parameter represents *work* to be performed. For example, the (else-less) Pascal `if` statement has two parameters, the test and the then body [Jensen and Wirth, 1975]. The implementation of a construct executes the compound statement parameters in an order consistent with the definition of the construct.

Sequential control constructs define a *total order* on the execution of the its compound statement parameters. In contrast, parallel control constructs define a *partial order* of execution. To prevent confusion regarding the meaning of a parallel control construct, we must be precise about this partial order. We will use the *precedes* relation to describe the partial order for parallel control constructs. Given two events, a and b, the expression $a \rightarrow b$ states that a must precede b. That is, in all possible executions of the program, event a occurs before event b. The precedes relation is transitive, but of course not reflexive.

Control constructs execute a compound statement as a single, indivisible unit. In addition, control constructs may be used in many different situations, and therefore do not in general know the internal structure of compound statements they execute. Given these two facts, control constructs may only define a partial order of execution between compound statements in terms of two events that take place during the execution of a compound statement: a control transfer from the construct to the compound statement and the corresponding return. We use $\downarrow$ `work` to denote the 'call' to a body of work, and $\uparrow$ `work` to denote its 'return'. Note that the call to a body of work always precedes its return, that is $\downarrow$ `work` $\rightarrow$ $\uparrow$ `work` $\forall$ `work`.

Programmers define a control construct by listing its parameters and specifying the partial orders with the precedes relation. For example, we can define the Pascal `if` as follows:

`if` *test* `then` *work*
$\downarrow$ `if` *test=true* $\rightarrow$ $\downarrow$ *work* $\rightarrow$ $\uparrow$ *work* $\rightarrow$ $\uparrow$ `if`
$\downarrow$ `if` *test=false* $\rightarrow$ $\uparrow$ `if`

It is often the case in the definition of a control construct that we mention no events within a body of work, and simply execute the body and wait on its return. The corresponding precedence notation, $\rightarrow \downarrow$ `work` $\rightarrow \uparrow$ `work` $\rightarrow$, sometimes gets unwieldy so we use the the shorthand expression $\rightarrow$ `work` $\rightarrow$. Using this shorthand the first `if` rule becomes:

$\downarrow$ `if` *test=true* $\rightarrow$ *work* $\rightarrow$ $\uparrow$ `if`

If the precedes relation is the only means of specification, implementors may *always* provide a sequential implementation corresponding to a topological sort of the precedes relations. However, in the presence of explicit synchronization, deadlock may result if a sequential implementation is used. For example, if the body of work passed to a sequential implementation of a parallel `for` construct explicitly synchronizes among its various activations, then we may have a situation in which the

construct waits on an iteration that waits on another iteration. To avoid this situation, we must introduce another relation — the *anti-preceeds* relation. When a control construct specifies that a anti-preceeds b, $a \not\rightarrow b$, then in no implementation of the construct may $a \rightarrow b$ be true. In other words, b cannot wait (even indirectly) on a. However, a can wait on b. Also, while $a \rightarrow b \Rightarrow b \not\rightarrow a$ the converse is not true, that is $b \not\rightarrow a \not\Rightarrow a \rightarrow b$. In other words, not waiting does not imply preceeding. The anti-preceeds relation is neither reflexive nor transitive.

For notational convenience, we define the *concurrent* relation. When a control construct specifies that two events a and b are concurrent, $a \parallel b$, then $a \not\rightarrow b \wedge b \not\rightarrow a$. That is, in no implementation of the construct may either $a \rightarrow b$ or $b \rightarrow a$ be true. Practically, this means that implementations of the construct must use at least blocking coroutines, if not true parallelism. The concurrent relation is reflexive, but not transitive.

4 Mechanisms for Control

Every imperative programming language must provide a set of primitive control mechanisms. If these mechanisms support control abstraction, programmers may implement new control constructs more suited to their application. This section introduces a small set of mechanisms for parallel programming with control abstraction and defines their semantics using the notation of section 3. These mechanisms are *operation invocation*, *statement sequencing*, *first-class closures*, *early reply*, *conditional execution*, and *wait-free synchronization*. These mechanisms are part of the Matroshka parallel programming language; see [Crowl, 1991] for additional details. With these mechanisms, programmers may build a rich variety of control constructs to represent precisely the parallelism in an algorithm.

4.1 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. For conciseness, we also use a conventional prefix/infix expression notation for sequential data operations, such as integer addition. These expressions have the semantics of their equivalent functional expressions.

Operations both accept parameters and return results. We define argument evaluation to occur before operation invocation. That is, given `f(g())` we know that `g` $\rightarrow$ `f`.

Operations may call themselves recursively. We choose recursion over iteration because it enables more direct expression of divide-and-conquer algorithms, which are important to parallel programming.

4.2 Statement Sequencing A sequence of statements defines a total order on statement executions (operation invocations). Notationally, we separate statements by a semicolon. For example, given `f(); g()` we know that `f` $\rightarrow$ `g`.

4.3 First-Class Closures General control abstraction requires a mechanism for encapsulating and manipulating the body of work passsed to a control construct. This work 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 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, and programmers may invoke any single closure many times. Each invocation produces a separate activation. These activations have no implicit synchronization. To communicate between closure activations and the control construct, closures may accept parameters and return results.

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. We denote the value-returning reply statement with the keyword `reply` preceding the return value expression. Replies that return control, but no value, omit the expression. 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 )`. We can also call a closure at the point we define it:

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

The first pair of parentheses defines the parameter type, the braces define the body, and the second pair of parenthesis invoke the closure and pass the argument.

The invocation of a closure preceeds the first statement in the closure and the evaluation of the reply value preceeds the closure reply. For example, given the closure definition

```
( p: integer ) { f1(); ...; fi(); reply g(); }
```

the statements calling the closure

```
...; fx(); closure( h() ); fy(); ...
```

result in the following partial order of execution:

$$\cdots \rightarrow \texttt{f}_x \rightarrow \texttt{h} \rightarrow \downarrow \textit{closure} \rightarrow \texttt{f}_1 \rightarrow \cdots \rightarrow \texttt{f}_i \rightarrow \texttt{g} \rightarrow \uparrow \textit{closure} \rightarrow \texttt{f}_y \rightarrow \cdots$$

As an example of the use of closures in a control construct, consider the Pascal `if` statement described in section 3. It takes two parameters, the test and the body of work. To cast this into our closure notation, we must convert the `if` statement into an operation call, and convert the work into a closure parameter. The definition (as opposed to the implementation) is:

```
operation if( test: boolean; work: closure( ) )
```

An example of its use is:

```
if( y > 0, { print y } )
```

This example introduces two notational shortcuts. First, when a closure takes no parameters, we omit the parameter list. Second, if the a value-less reply the last statement in a closure, we omit the reply.

Closures are, in essence, the in-line definition of a nested operation. Operations are simply named closures. So all claims about closures also apply to operations. In particular operations as well as closures may be passed as arguments for later invocation.

4.4 Early Reply An invocation of a operation (or closure) 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.

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

```
( p: integer ) { f1(); ...; fi(); reply g(); fj(); ...; fn() }
```

the statements calling the closure

```
...; fx(); closure( h() ); fy(); ...
```

result in the following partial order of execution:

$$\cdots \rightarrow \texttt{f}_x \rightarrow \texttt{h} \rightarrow \downarrow \textit{closure} \rightarrow \texttt{f}_1 \rightarrow \cdots \rightarrow \texttt{f}_i \rightarrow \texttt{g} \rightarrow \uparrow \textit{closure} \begin{cases} \nearrow \texttt{f}_j \rightarrow \cdots \rightarrow \texttt{f}_n \\ \rightarrow \end{cases} \uparrow \textit{closure} \rightarrow \texttt{f}_y \rightarrow \cdots$$

The statements $\texttt{f}_j \ldots \texttt{f}_n$ may execute in parallel with statement $\texttt{f}_y$ and its successors, that is $\downarrow \texttt{f}_j \not\rightarrow \downarrow \texttt{f}_y \wedge \downarrow \texttt{f}_y \not\rightarrow \downarrow \texttt{f}_j$ and therefore $\downarrow \texttt{f}_j \parallel \downarrow \texttt{f}_y$.

4.5 Conditional Execution For conditional execution, we adopt the Smalltalk approach [Goldberg and Robson, 1983] and depend on a *Boolean* type and an `if` operation that conditionally executes a closure. In our case, the operation syntax is that developed in section 4.3 and the semantics are those described in section 3.

```
operation if( test: boolean; work: closure() )
↓ if( true, work ) → work → ↑ if
↓ if( false, work ) → ↑ if
```

We invoke this operation just as we would any other. For example, in `if( y>0, { z := x/y } )` the assignment executes only when $\texttt{y} > 0$.

The implementations of `ifelse`, `while` and `repeat` in terms of `if` are straightforward.

4.6 Synchronization So far, we have presented no mechanism for synchronization other than that implicit in an operation invocation waiting for the reply. This is not sufficient for general parallel programming; we need to synchronize explicitly.

Our examples use a simple wait-free condition variable for synchronization. The condition variable has atomic `signal` and `blocked` operations. The `signal` operation may be invoked only once, while the `blocked` operation may be invoked many times. Thier syntax and semantics are:

```
type condition
operation blocked( var cond: condition ): boolean
operation signal( var cond: condition )
↑ blocked = true → ↓ signal
↓ signal → ↑ blocked = false
```

The first rule says that `blocked` returns *true* before the call to `signal`. The second rule says that `blocked` returns *false* after the call to `signal`.

This condition variable is not powerful enough to provide mutual exclusion, and is therefore insufficient as a synchronization primitive. We use it here because it meets the needs of our examples, and has simple semantics. We expect parallel programming languages to provide wait-free synchronization primitives richer than our condition variable, such as compare-and-swap, so that programmers may achieve mutual exclusion. We also expect languages and implementations to provide blocking synchronization.

These Mechanisms are Sufficient In designing a parallel control construct, programmers need to be able to generate an arbitrary partial order of events. The control mechanisms presented above can generate arbitrary (computable) partial orders. To see this, first note that sequence, conditional execution, and recursion enable us to invoke an arbitrary number of closures, each with an identity that can be computed and passed via its parameters or environment. Each invocation of a closure may then reply early, creating an independent thread of control. Each thread may set conditions and wait on any (computable) function of other conditions.

The capacity to generate arbitrary partial orders is not helpful in coordinating existing processes. To do this we need the synchronization primitives that allow processes to achieve consensus. Herlihy [1988] presents several wait-free synchronization primitives and analyses them for their ability to enable two or more processes to achieve consensus, and hence mutual exclusion. For example, compare-and-swap is sufficient for an arbitrary number of processes to achieve consensus. With a primitive like compare-and-swap, the mechanisms we presented can generate arbitrary partial orders and coordinate among arbitrary existing processes.

5 Building Control Constructs

In this section, we provide examples of defining, implementing, and using parallel control constructs. In the first two examples, we show that the partial order of the construct's implementation implies the partial order of the construct's definition, thus showing that the implementation satisfies the definition. Our first example is a `wait` operation on condition variables. We use `wait` in the implementation of a parallel `cobegin` operation. We then use `cobegin` in the implementation of a parallel `forall`.

5.1 Wait on Condition Given the condition variable defined in section 4.6 we define a `wait` operation that will not reply until the condition has been signaled:

```
operation wait( var cond: condition )
↓ signal → ↑ wait
```

It has a straightforward implementation based on `if` and recursion:[2]

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

We can derive the defined partial order (above) by induction from the partial orders of `blocked` and `signal`. For each precedence relation, we note the sections that define the rules used to derive the relation. The base case occurs when `signal` has already executed.

$$\downarrow \texttt{signal} \xrightarrow{1} \uparrow \texttt{blocked} = \textit{false} \xrightarrow{2} \downarrow \texttt{if(}\ \textit{false}, \ \ldots\ \texttt{)} \xrightarrow{3} \uparrow \texttt{if(}\ \textit{false}, \ \ldots\ \texttt{)} \xrightarrow{4} \uparrow \texttt{wait}$$

Precedence 1 derives from the definition of conditions (4.6). Precedence 2 derives from the evaluation of `blocked` as an argument before invoking `if` (4.1). Precedence 3 derives from the definition of `if` with a false argument (4.5). And finally, precedence 4 derives from the implicit reply and the definition of closures (4.3).

The induction step occurs when `signal` has not already executed.

$$\texttt{blocked} = \textit{true} \xrightarrow{5} \downarrow \texttt{if(}\ \textit{true}, \ \ldots\ \texttt{)} \xrightarrow{6} \texttt{wait}_{recurse} \xrightarrow{7} \uparrow \texttt{if} \xrightarrow{8} \uparrow \texttt{wait}$$

Precedence 5 derives from the evaluation of `blocked` as an argument before invoking `if` (4.1). Precedences 6 and 7 derive from the definition of `if` with a true argument (4.5). And finally, precedence 8 derives from the implicit reply and the definition of closures (4.3).

An alternate implementation of `condition` and `wait` could provide blocking synchronization, rather than the busy waiting presented here.

5.2 Cobegin The `cobegin` construct may execute two closures concurrently and replies only when both have replied.[3] Its syntax and semantics are:

```
operation cobegin( work1, work2: closure() )
↓ cobegin → work1 → ↑ cobegin
↓ cobegin → work2 → ↑ cobegin
↓ work2 ↛ ↓ work1
```

These rules state, respectively, that both closures start after the `cobegin`, both closures reply before the `cobegin` replies, and `work1` does not wait on `work2`.[4]

The above rules permit but do not guarantee concurrent[5] execution (as opposed to truly parallel execution). The rule that guarantees concurrent execution, ↓ `work1` ↛ ↓ `work2`, states that `work2` does not wait on `work1`. In conjuction with the third rule above, ↓ `work1` ∥ ↓ `work2`, and neither work may wait on the other, which implies that `cobegin` must invoke both closures before waiting on their replies. In general, it is not good practice to use control constructs that guarantee concurrency because they exclude processor-efficient sequential implementations.

---

[2] Tail recursion elimination will avoid stack growth.

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

[4] This rule is primarily useful when using `cobegin` to implement other control constructs. We rely on this rule in our implementation of `forall` in section 5.3.

[5] We use the term concurrent to include those implementations that may execute correctly on uniprocessors. Such implementations need some form of blocking thread.

One possible parallel implementation of cobegin follows. It uses only the mechanisms defined in section 4 and the wait operation defined in section 5.1. We use early reply as the source of parallelism and our condition variable as the source of synchronization.

```
implement cobegin( work1, work2: closure() )
{ var done: condition;
  --- define and execute a closure to execute one argument
  { reply;              --- remainder of closure executes in parallel
    work1();            --- do work1
    signal( done )      --- work1 has finished, signal it
  }();                  --- directly execute closure
  --- execution continues here, in parallel, after the reply executes
  work2();              --- do work2
  wait( done )          --- wait for signal indicating work1 finished
                        --- implicit reply from cobegin
}
```

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

$\downarrow$ `cobegin` $\xrightarrow{1}$ $\downarrow$ inner closure $\xrightarrow{2}$ $\uparrow$ inner closure
$\uparrow$ inner closure $\xrightarrow{3}$ `work1` $\xrightarrow{4}$ $\downarrow$ `signal` $\xrightarrow{5}$ $\uparrow$ `wait`
$\uparrow$ inner closure $\xrightarrow{6}$ `work2` $\xrightarrow{7}$ $\downarrow$ `wait` $\xrightarrow{8}$ $\uparrow$ `wait`
$\uparrow$ `wait` $\xrightarrow{9}$ $\uparrow$ `cobegin`
$\downarrow$ `work2` $\overset{10}{\not\rightarrow}$ $\downarrow$ `work1`

Precedences 1 and 2 derive from the first statement in a closure executing after the closure is invoked (4.3). Precedence 3 derives from the first statement after a reply in a closure executing after the reply (4.4). Precedences 4, 6 and 7 derive from statement sequencing (4.2). Precedence 5 derives from the definition of wait (5.1). Precedence 8 is inherent to an operation replying after its invocation. Precedence 9 derives from the implicit reply and the definition of closures (4.3). And finally, precedence 10 derives from the concurrent execution of the statement after a reply and the statement after a call to a closure (4.4). With the exception of the calls to signal and wait, the derivation of precedences is straightforward.

5.3 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].[6] Its syntax and semantics are:

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

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

[6] 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.

These rules state, respectively, that the `forall` starts before any iteration; all iterations reply before `forall` does; and higher-numbered iterations do not wait on lower-numbered iterations.[7] Again, we omit the rule that guarantees concurrency:

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

which states 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 ) } ) } ) }
```

We omit the detailed verification of the specification. Note, however, that meeting the third `forall` rule relies on the third `cobegin` rule.

In addition to the parallel implementation, `forall` also has valid sequential implementations. In particular, we can implement `forall` in with a sequential `for` operation.

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

These examples show the power of control abstraction when used to define parallel control constructs. Using closures and early reply we can represent many different forms of parallelism. In particular, we used closures, conditional execution, recusrion, and wait-free synchronization to implement waiting synchronization, which we then used with closures and early reply to implement `cobegin`. We then used `cobegin` with closures, conditional execution, and recursion to implement `forall`. These examples show that control abstraction enables programmers to extend the set of control constructs beyond those defined by the language designer.

6 Parallel Programming with Control Abstraction

We have programmed several parallel applications using control abstraction in the Matroshka language. Our experiences have confirmed our intuition about the benefits of control abstraction, and produced some specific lessons on how to use control abstraction in parallel programs. In this section we give examples of how application-specific control constructs can improve parallel programs. The first example is Gaussian elimination and shows the ability to select easily among multiple implementations of a control construct and exploit different parallelizations. The second example is part of a program to compute subgraph isomorphism and shows how associating control operations with data abstractions can improve the clarity and precision of parallel programs. For more detailed treatment, see [Crowl and LeBlanc, 1991] or [Crowl, 1991].

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

6.1 Select Among Multiple Implementations In implementing a parallel algorithm, programmers are faced with the task of balancing the potential speedup of parallelism with the overhead of starting parallel tasks. In addition, they must balance the use of explicit synchronization (and its corresponding debugging problems) with the improved performance that a more precise description of parallelism may bring. Most current parallel programming languages force programmers to make such decisions early in program development because the choice of parallelism affects program development.

In Gaussian elimination, the primary source of parallelism is in the production of the upper triangular matrix (LU decomposition). Data flow constraints for Gaussian elimination state that pivot equations must reduce any given equation in order, and an equation must be reduced completely before it can be used as a pivot. Our goal is to represent the most general applicable partial order that respects these constraints directly in a program's control constructs.

Using control abstraction, it is both possible and desireable to base the specification of control on the synchronization constraints inherent in the algorithm. We do this by defining a new control construct, `triangulate`, that encapsulates precisely the partial order required by Gaussian elimination. It takes two parameters: the number of equations in the system, and the statements to be executed for each pivot and reduction equation pair. This construct encapsulates all parallelism and synchronization in selecting pairs of pivot and reduction equations. We encapsulate the statements that implement a 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.

```
define triangulate( size: integer; work: closure( pivot, reduce: integer ) )
↓triangulate( size, work ) → ↓work( i, j )                    [i,j : 1 ≤ i < j ≤ size]
↑work( i, j ) → ↓work( k, j )                  [i,j,k : 1 ≤ i < j ≤ size ∧ i < k ≤ size]
↑work( i, j ) → ↓work( j, k )                  [i,j,k : 1 ≤ i < j ≤ size ∧ i < k ≤ size]
```

This construct has several different implementations, from sequential to maximally parallel, each embedding only the synchronization it needs. Because synchronization is embedded in the implementation of the `triangulate` control construct, and not in the statements that reduce an equation, we were able to simultaneously select parallelism and synchronization by choosing an implementation for `triangulate`.

Written with `triangulate`, the code to form the upper triangular matrix[8] is:

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

By annotating each use of `triangulate` and `forall` with the desired implementation, we were able to implement many different parallelizations of Gaussian elimination (changing only the annotations) and compare the performance of different parallelizations on the Butterfly and Alliant

[8] For historical and expository purposes, we us an algorithm without pivoting. The algorithm is numerically unstable.

architectures. For example, selecting a sequential implementation of `triangulate` and a sequential implementation of `forall`, yields an efficient sequential program. Selecting a vector implementation of `forall` takes advantage of any vector hardware. For processors with hardware support for barriers, we might wish to use an implementation of `triangulate` based on the implicit barrier synchronization of `forall`. For processors without hardware support for barriers, such as the Butterfly, an implementation of `triangulate` based on condition variables is most efficient. We can adapt this program to a wide variety of architectures simply by changing the implementation annotations for each control construct.

6.2 Associate Control and Data Control abstraction is especially powerful when combined with data abstraction. The relationship between control abstraction and data abstraction shows clearly in our implementation of subgraph isomorphism.

The problem is to find the set of isomorphisms from a small graph to subgraphs of a larger graph. 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 is based on tree-search with constraint propagation. This paper concentrates on one of those constraints — the distances from the current vertex in the small graph to other verticies in the small graph must be no larger than the distances from the current vertex in the large graph to other verticies in the large graph. We remove from the set of possible mappings those which are inconsistent with the distance constraint.

In a sequential language we would typically write code that iterates over possible elements of the set of mappings, testing for membership, and then testing the distance condition. When parallelized, the source of parallelism in this code is the possible elements of the set, rather than the much smaller number of actual elements.

With control abstraction, we can define a set operation that iterates in parallel over actual elements of the set, testing them for removal. We define a new construct, `remove_elements_cond`, that removes those elements of a set that meet a given test.

```
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 ]
```

Because this iterator combines the specification of paralelism across elemets of the set with the synchronization required by the removal operation, the implementation can restrict its generation of parallelism so that removals do not need to synchronize, thus improving the efficiency of the program.

A distance filter based on this construct expresses our algorithmic intent precisely, while leaving a great deal of latitude in the possible implementations of `remove_element_cond`.

```
implement distances( curr_small, curr_large: integer; var node: tree_node )
{ --- for all verticies in the small graph
  forall( 1, maximum_small, ( other_small: integer )
    { remove_element_cond(          --- remove elements from the
        node[other_small],          --- set of possible mappings of that vertex
        ( other_large: integer )    --- to a vertex in the large graph
            {                       --- that do not meet the distance constraint
              reply small_distance[curr_small,other_small]
                    < large_distance[curr_large,other_large] } ) } ) }
```

We cannot reasonably expect language designers to anticipate application-specific control constructs such as `triangulate` and `remove_element_cond`, so only languages that support control abstraction can support such constructs.

7 Implementation

Earlier sections showed the importance of control abstraction in parallel programming. 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.

In the case of our mechanisms, seven straightforward optimizations reduce the execution cost of these mechanisms to that comparable with compiled languages.

Invocations as Procedure Calls: An invocation may reply early, causing concurrent execution. So, a conservative implementation of invocation provides a separate thread of control for each invocation. We can reduce this cost by implementing operations that have no statements after the reply, and hence no concurrency, as procedure calls.

Delayed Replies: Some early replies can be safely delayed until after the last statement, again enabling a procedural implementation.

In-line Substitution: By statically identifying the procedure that implements an operation, we can do in-line substitution. This technique is effective in implementing sequential control constructs as machine branches.

Stack Allocation of Closures: We can reduce the cost of closures by allocating their activations on the stack, rather than from the heap. This requires either language restrictions or program analysis [Kranz *et al.*, 1986].

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 that implementation. In particular, implementations of `forall` are most efficient when they can directly manipulate scheduler queues.

Stack Borrowing: When a the body of work passed to a parallel construct does not block, we can execute it from within the scheduler and avoid task creation.

Last-In-First-Out Scheduling: FIFO scheduling is equivalent to a breadth-first search of the tree of tasks, which requires $O(n)$ simultaneous activations. The representation of these activations could swamp available memory. On the other hand, LIFO scheduling is equivalent to depth-first search and requires only $O(\log n)$ simultaneous activations.

Using these optimizations, our prototype implementation of Matroshka [Crowl, 1991] produces programs that executes two to four times slower than equivalent C programs compiled with an optimizing compiler. The performance difference arises primarily because the Matroshka compiler uses C as an intermediate language (which causes substantial inefficiencies), and secondarily because the compiler does not do inlining of user-defined operations (and hence control abstractions) and because the prototype language does not permit passing sub-arrays. These problems have straightforward corrections. Applying the corrections by hand brings Matroshka execution times to within 4% of comparable C programs for several programs on the BBN Butterfly multiprocessor. Figure 1 shows the corresponding execution times for the Gaussian elimination example. We expect that a production compiler for Matroshka would be competitive with an optimizing C compiler.

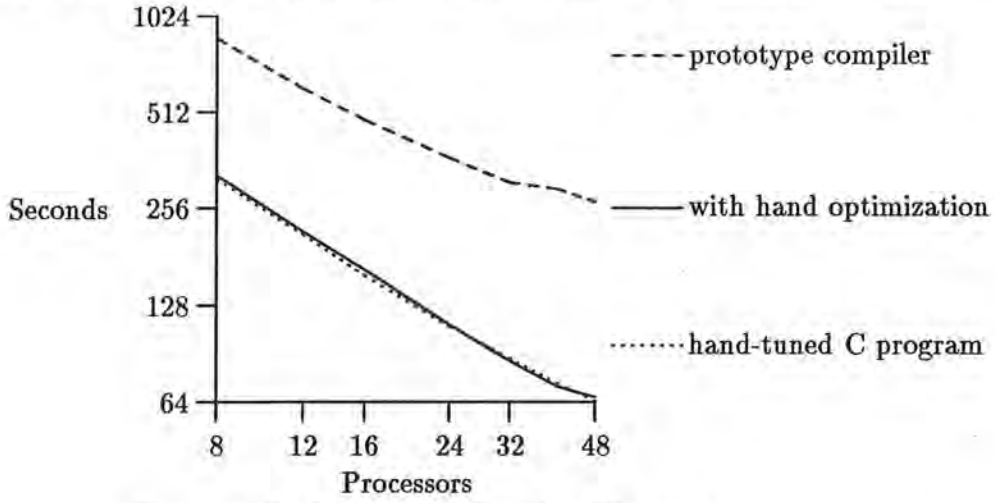

Figure 1: Performance of Gaussian Elimination

8 Conclusions

Control abstraction has four benefits that are particularly important for parallel programming.

- Programmers are not limited to a fixed set of control constructs. New constructs that express arbitrary partial orders on statement execution can be created and stored in a library for use by others. This is particularly important because application-specific control constructs can provide substantial improvements to parallel programs.
- Each control construct can have multiple implementations, corresponding to different parallelizations. In tuning a program for a specific architecture, or in porting a program to a new architecture, programmers can experiment with alternative parallelizations by selecting implementations from a library of control constructs.

- Programmers can use constructs that reflect the potential paralleism of the algorithm, isolating decisions on actual parallelism and synchronization within the implementation of constructs and away from program logic.
- Programmers can associate control operations with data structures, thus providing expressive and concise data-dependent parallelism.

We presented a notation for precisely defining control constructs, introduced a small set of primitive control mechanisms for control abstraction and defined them in terms of our notation. We then showed how to define and implement new control constructs, verifying that the implementations meet the definitions. We gave examples of the value of application-specific control constructs in parallel programming. Finally, we described several optimizations that admit an implementation of our mechanisms competitive with compiled languages. We conclude that the enormous benefits and reasonable costs of control abstraction argue for its inclusion in explicitly parallel programming languages.

References

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

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

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

[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.

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

[Crowl and LeBlanc, 1991] Lawrence A. Crowl and Thomas J. LeBlanc, "Architectural Adaptability in Parallel Programming via Control Abstraction," Technical Report 359, Computer Science Department, University of Rochester, January 1991.

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

[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.

[Herlihy, 1988] Maurice P. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization," In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

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

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

[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.

[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.

[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.

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

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