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

Abstract

Control abstraction is the process by which programmers define new control constructs, specifying a statement ordering separately from an implementation of that ordering. We argue that control abstraction can and should play a central role in parallel programming. Control abstraction can be used to build new control constructs for the expression of parallelism. A control construct can have several implementations, representing the varying degrees of parallelism to be exploited on different architectures. Control abstraction also reduces the need for explicit synchronization, since it admits a precise specification of control flow. Using several examples, we illustrate these benefits of control abstraction. We also show that we can efficiently implement a parallel programming language based on control abstraction. We conclude that the enormous benefits and reasonable costs of control abstraction argue for its inclusion in explicitly parallel programming languages.

1: Introduction

Sequential programming languages use sequencing, repetition, and selection to define a total ordering of statement executions in a program. Parallel programming languages 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.

Control abstraction has been used in sequential programming languages such as CLU [11] to define control constructs that iterate over the elements of an abstract data type. Few parallel programming languages support control abstraction however, since the benefits of using control abstraction for parallel programming are not generally recognized. These benefits include:

New control constructs: With control abstraction, programmers can build new constructs for parallel programming without changing the language definition or implementation.

Alternative implementations: Control constructs can have several different implementations, each exploiting a different degree of parallelism. Programmers may choose among these implementations to tune the application to a given architecture.

Precise specification of parallelism: Programmers can build application-specific control constructs that admit only the parallelism inherent in the algorithm. This approach provides flexibility in adapting to different architectures, and reduces the need for explicit synchronization.

Coordination of process and data distribution: By associating control constructs with abstract data types, programmers can create a direct correspondence between the exploitation of parallelism and the distribution of data.

In this paper we argue that general control abstraction can play a central role in parallel programming. We first introduce a notation and a primitive set of mechanisms for defining control constructs, and then show how to use the primitive mechanisms to implement some common parallel programming constructs. These general examples are followed by more detailed

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.

examples of application-specific control constructs that illustrate many of the advantages of control abstraction. We claim that programs written with control abstraction can be nearly as efficient as programs written in C, and support that claim by describing optimizations used in our implementation. We conclude that the benefits of control abstraction in parallel programming far outweigh the costs, and that parallel programming languages should support it.

2: Related Work

Hilfinger [8] provides a short history of major abstraction mechanisms in programming languages, with an emphasis on procedure and data abstraction. This history does not mention control abstraction, although the mechanisms for control abstraction are present in Lisp.

Control abstraction has often been used in sequential languages to support data abstraction. For example, CLU iterators [11] (or generators) are a form of control abstraction. With iterators, the user of an abstract type can operate on the elements of the type without knowing the underlying representation. In CLU, and other languages designed to support data abstraction, control abstraction plays a secondary role to the specification and representation of abstract data types.

Given that parallelism is a form of control flow, control abstraction is particularly important for parallel programming, 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 [6] and Paralation [12] 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 primary benefit 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 [3] and Chameleon [1]. 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 [4].

3: Defining Control Constructs

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

In defining a control construct, we must be precise about the allowable execution orderings. We will use the *precedes* relation to describe the partial order for parallel control constructs. Given two events, a and b, the 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 not reflexive.

Control constructs execute a compound statement as a single, indivisible unit. In addition, control constructs may be used in many different contexts, and therefore do not in general know the internal structure of the compound statements they execute. Given these two facts, a control construct can only impose a partial order of execution between compound statements in terms of two events that take place during the execution of a compound statement: the control transfer from the control construct to a compound statement, and the corresponding return.

We will use $\downarrow stmt$ to denote the transfer of control to a compound statement, and $\uparrow stmt$ to denote its return. The transfer to a compound statement always precedes its return; that is $\forall\ stmt,\ \downarrow stmt \rightarrow \uparrow stmt$. In addition, a control construct must be invoked before it can invoke any statements, that is, $\downarrow construct \rightarrow \downarrow stmt$. Although this precedence constraint is implicit in every control construct, the absence of any such rule for a given statement signifies that the statement is not invoked at all. That is, a control construct invokes a statement passed as a parameter *if and only if* the statement's invocation is present in the precedence rules.

We define a control construct by listing its parameters (including the compound statements to be executed by the control construct) and specifying a partial order of execution using the precedes relation. For example, the Pascal `if` statement has two parameters, a boolean value representing the condition and a compound statement representing the `then` clause. We can define the Pascal `if` statement as follows:

$\texttt{if}\ cond\ \texttt{then}\ stmt$
$\downarrow \texttt{if(}\ true,\ stmt\ \texttt{)} \rightarrow \downarrow stmt \rightarrow \uparrow stmt \rightarrow \uparrow \texttt{if}$
$\downarrow \texttt{if(}\ false,\ stmt\ \texttt{)} \rightarrow \uparrow \texttt{if}$

The first line represents the syntax of the construct, while subsequent lines provide the partial order rules. The first rule says that when the condition parameter is true, the invocation of `if` precedes the invocation of *stmt*, the invocation of *stmt* precedes its reply, and the *stmt* reply precedes the reply from `if`. (In future examples, we will use the shorthand notation $\rightarrow$ *stmt* $\rightarrow$ in place of $\rightarrow \downarrow$ *stmt* $\rightarrow \uparrow$ *stmt* $\rightarrow$.) The second rule says that when the condition parameter is false, the invocation of `if` precedes its return. Note that the absence of an invocation of *stmt* in this rule means that the statement is not invoked when the condition is false.

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

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

4: Control Abstraction Mechanisms

This section introduces a small set of mechanisms for parallel programming with control abstraction and defines their semantics using the notation of §3. The mechanisms are *statement sequencing*, *operation invocation*, *first-class closures*, *early reply*, *conditional execution*, and *wait-free synchronization*. These mechanisms are part of the Matroshka parallel programming model [4]. With these mechanisms, programmers may build a wide variety of control constructs to represent the parallelism in an algorithm.

4.1: Statement sequencing

A sequence of statements defines a total order on statement execution. Notationally, we separate statements by a semicolon. For example, $\texttt{s}_1\texttt{;}\ \texttt{s}_2$ means that $\texttt{s}_1 \rightarrow \texttt{s}_2$.

4.2: Operation invocation

Operations are recursive procedures (or methods) that accept parameters and return results. As in nearly all imperative programming languages, we require that argument evaluation precede operation invocation; that is, `f(g())` implies `g` $\rightarrow$ `f`. This requirement enables sequential evaluation of expressions, without limiting the potential for parallelism in control flow.

4.3: First-class closures

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

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

In our notation, the definition of a closure consists of a parameter list within parentheses followed by a sequence of statements within braces. One of these statements may be the `reply` statement containing a return value expression. (A `reply` statement without an expression simply returns control.) Using this notation, a closure that accepts an integer parameter and returns twice its value would be written as follows:

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

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

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

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

The invocation of a closure precedes the first statement in the closure. In addition, the evaluation of the reply value precedes the closure reply. For example, the closure definition

$\texttt{(p: integer)\{ s}_1\texttt{; ...; s}_i\texttt{; reply f() \}}$

and the statements calling the closure

$\texttt{...; s}_x\texttt{; } \mathit{closure}\texttt{(g()); s}_y\texttt{(); ...}$

result in the following partial order of execution:

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

As an example of the use of closures in defining a control construct, consider the Pascal `if` statement described in §3. This construct takes two parameters, a boolean value and a compound statement. In our notation, the `if` statement becomes an operation invocation, which accepts the compound statement as a closure parameter. The definition of the construct is:

```
if ( cond: boolean; stmt: 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 a valueless reply is the last statement in a closure, we omit the reply statement.

Closures are, in essence, the in-line definition of a nested operation. Operations are simply named closures. 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

When an invocation of an operation (or closure) returns a result, it may continue executing in parallel with the caller. This mechanism, called *early reply*, is the sole source of parallelism in Matroshka.[1] This mechanism is not new [2, 10, 13], but its expressive power does not appear to be widely recognized.

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

$\texttt{(p: integer)\{ s}_1\texttt{; ...; s}_i\texttt{; reply f();}$
$\qquad\texttt{s}_j\texttt{; ...; s}_n\texttt{ \}}$

and the statements calling the closure

$\texttt{...; s}_x\texttt{; } \mathit{closure}\texttt{(g()); s}_y\texttt{; ...}$

result in the following partial order of execution:

$$\cdots \rightarrow \texttt{s}_x \rightarrow \texttt{g} \rightarrow \downarrow \mathit{closure} \rightarrow \texttt{s}_1 \rightarrow \cdots$$
$$\rightarrow \texttt{s}_i \rightarrow \texttt{f} \rightarrow \uparrow \mathit{closure} \rightarrow \texttt{s}_y \rightarrow \cdots$$
$$\searrow \texttt{s}_j \rightarrow \cdots \rightarrow \texttt{s}_n$$

The statements $\texttt{s}_j \ldots \texttt{s}_n$ may execute in parallel with statement $\texttt{s}_y$ 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 $\downarrow \texttt{f}_j \parallel \downarrow \texttt{f}_y$.

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

4.5: Conditional execution

For conditional execution, we adopt the Smalltalk approach [5] and assume 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.

```
if( cond: boolean; work: closure() )
```

$\downarrow \texttt{if(}\ \mathit{true}\texttt{, work)} \rightarrow \texttt{work} \rightarrow \uparrow \texttt{if}$

$\downarrow \texttt{if(}\ \mathit{false}\texttt{, work)} \rightarrow \uparrow \texttt{if}$

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. For the purposes of our examples, we will use simple wait-free condition variables; in practice, any reasonable primitive would suffice.

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

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

$\uparrow \texttt{pending:}\mathit{true} \rightarrow \downarrow \texttt{signal}$

$\downarrow \texttt{signal} \rightarrow \uparrow \texttt{pending:}\mathit{false}$

The first rule says that any reply from `pending` that returns *true* must precede the call to `signal`. The second rule says that any reply from `pending` that returns *false* must occur after the call to `signal`.

Note that this condition variable is not capable of achieving mutual exclusion, and hence is not adequate as a general-purpose synchronization primitive. We assume that an imperative parallel programming language more powerful synchronization primitives, such as compare-and-swap.

These mechanisms are sufficient

To design new parallel control constructs, programmers need to be able to specify an arbitrary partial order of events. The control mechanisms presented above can generate arbitrary (computable) partial orders. Sequencing, conditional execution, and recursion can be

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

Figure 1: Implementation of `cobegin`

used 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. Given a synchronization primitive like compare-and-swap, we can also synchronize among arbitrary existing processes [7]. With compare-and-swap, the mechanisms we presented can generate arbitrary partial orders and coordinate among an arbitrary number of existing processes.

5: Building control constructs

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

5.1: Wait on condition

Given the condition variables defined in §4.6, we will construct a `wait` operation that does not reply until a condition has been signaled. Our implementation will use busy-waiting; alternative implementations based on blocking synchronization are also possible.

`wait( var cond: condition )`

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

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

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

We use induction and the partial orders of the primitive operations `pending` and `signal` to verify that this implementation satisfies the partial order. The base case occurs when `signal` has already been invoked:

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

Precedence 1 derives from the definition of the primitive operations on condition variables, §4.6. Precedence 2 derives from the evaluation of `pending` as an argument before invoking `if`, §4.2. Precedence 3 derives from the definition of `if` with a false argument, §4.5. And finally, precedence 4 derives from the definition of closures (with an implicit reply), §4.3.

The induction step occurs when `signal` has not yet been invoked. Assuming $\downarrow \mathtt{signal} \rightarrow \uparrow \mathtt{wait}_{recursive}$,

$\mathtt{pending}{:}\mathit{true} \xrightarrow{5} \downarrow \mathtt{if(}\ \mathit{true},\ \mathit{stmt}\mathtt{)}$
$\xrightarrow{6} \mathtt{wait}_{recursive} \xrightarrow{7} \uparrow \mathtt{if} \xrightarrow{8} \uparrow \mathtt{wait}$

Precedence 5 derives from the evaluation of `pending` as an argument before invoking `if`, §4.2. Precedences 6 and 7 derive from the definition of `if` with a true argument, 4.5. Finally, precedence 8 derives from the definition of closures (with an implicit reply), 4.3. And by transitivity, $\downarrow$ `signal` $\rightarrow$ $\uparrow$ `wait`.

5.2: Cobegin

Our next example is a `cobegin` construct that can execute two closures concurrently, replying only when both have completed execution. Its syntax and semantics are:

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

$\downarrow$ `cobegin` $\rightarrow$ `work1` $\rightarrow$ $\uparrow$ `cobegin`

$\downarrow$ `cobegin` $\rightarrow$ `work2` $\rightarrow$ $\uparrow$ `cobegin`

$\downarrow$ `work2` $\not\rightarrow$ $\downarrow$ `work1`

These rules state, respectively, that both closures start after the `cobegin`, both closures reply before the

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

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

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

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

Figure 2: Definition of `forall`

```
implement forall( lower, upper: integer; work: closure( iteration: integer ) )
{ if( lower = upper, { work( lower ) } );   --- the base case, one element in the range
  if( lower < upper, { middle := (lower + upper) div 2;   -- the inductive case, two or more
                       cobegin( { forall( lower, middle, work ) },    --- do the two halves
                                { forall( middle+1, upper, work ) } ) } ) }   --- in parallel
```

Figure 3: Implementation of `forall`

`cobegin` replies, and `work1` does not wait on `work2`.[2]

The above rules permit but do not guarantee concurrent[3] execution. The rule that guarantees concurrent execution, $\downarrow$ `work1` $\not\rightarrow$ $\downarrow$ `work2`, states that `work2` does not wait on `work1`. With the third rule above, $\downarrow$ `work1` $\|$ $\downarrow$ `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.

One possible parallel implementation of `cobegin` appears in figure 1. It uses only the mechanisms defined in §4 and the `wait` operation defined in §5.1. We use early reply as the source of parallelism and our condition variable as the source of synchronization. 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, §4.1. 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, anti-precedence 10 derives from the concurrent execution of the statement after a reply and the statement after a call to a closure, §4.4. Except the calls to `signal` and `wait`, the derivation of precedences is straightforward.

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

[3] We use the term concurrent to include those implementations that may execute correctly on uniprocessors, in addition to truly parallel implementations. Such implementations need some form of blocking thread.

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 [11]. Its syntax and semantics appear in figure 2. 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.[4] Again, we omit the rule that guarantees concurrency:

$\downarrow$ `work(` i `)` $\|$ $\downarrow$ `work(` j `)` $[i, j : i \neq j \wedge$ `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` (figure 3). 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` with a sequential `for` operation.

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

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

$$\begin{array}{ll}
\texttt{triangulate(size: integer; work: closure(pivot, reduce: integer))} & \\
\downarrow \texttt{triangulate(size, work)} \rightarrow \downarrow \texttt{work(} \ i, \ j \ \texttt{)} & [i, j : 1 \leq i < j \leq \texttt{size}] \\
\uparrow \texttt{work(} \ i, \ j \ \texttt{)} \rightarrow \downarrow \texttt{work(} \ k, \ j \ \texttt{)} & [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}] \\
\uparrow \texttt{work(} \ i, \ j \ \texttt{)} \rightarrow \downarrow \texttt{work(} \ j, \ k \ \texttt{)} & [i, j, k : 1 \leq i < j \leq \texttt{size} \wedge i < k \leq \texttt{size}]
\end{array}$$

Figure 4: Definition of `triangulate`

```
var system: array [ 1..SIZE ] of array [ 1..SIZE ] of float;
triangulate( SIZE, ( pivot, reduce: integer )   --- for each pivot and reduction
 pair
  { var fraction := system[reduce][pivot] / system[pivot][pivot];
    forall( pivot, SIZE, ( variable: integer )   --- for each variable in the eq
uation
      { system[reduce][variable] -:= fraction * system[pivot][variable] } ) } )
```

Figure 5: Implementation of Gaussian elimination

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, recursion, 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 abstraction

We programmed several parallel applications using control abstraction in the Matroshka language. Our experiences 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 [4].

6.1: Multiple implementations

In implementing a parallel algorithm, programmers have 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.

Using control abstraction, it is both possible and desirable to base the specification of control on the parallelism and synchronization constraints inherent in the algorithm. In Gaussian elimination, the primary source of parallelism is in computing 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.

We represent the partial order of Gaussian elimination by defining a new control construct, `triangulate` (figure 4), that encapsulates precisely the partial order required. It takes two parameters: the number of equations in the system, and the statements to be executed for each pivot and reduction equation pair. The statements that reduce an equation become a closure parameter; 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. This construct has several different implementations, from sequential to maximally parallel, each embedding only the synchronization it needs. Because synchronization is embedded in the implemen-

```
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 ]
```

Figure 6: Definition of `remove_element_cond`

```
implement distances( curr_small, curr_large: integer; var node: tree_node )
{ --- for all vertices 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] } ) } ) }
```

Figure 7: Implementation of `distances`

tation of the `triangulate` control construct, and not in the statements that reduce an equation, we were able to select parallelism and synchronization simultaneously by choosing an implementation for `triangulate`.

Written with `triangulate`, the code to form the upper triangular matrix[5] appears in figure 5. 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 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. The Butterfly can take advantage of the several parallel implementations of `triangulate`, while the Alliant can take advantage of both a vector implementation of `forall` and a parallel implementation of `triangulate`. We can adapt this program to a wide variety of architectures simply by changing the implementation annotations for each control construct.

6.2: Data abstraction

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 vertices in the small graph must be no larger than the distances from the current vertex in the large graph to other vertices in the large graph. We remove from the set of possible mappings those that 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` (figure 6), that removes those elements of a set that meet a given test. Because this iterator combines the specification of parallelism across elements 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 (figure 7) expresses our algorithmic intent precisely, while leaving much latitude in the possible implementations of

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

`remove_element_cond`.

We cannot reasonably expect language designers to include 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.

Seven straightforward optimizations reduce the execution cost of our mechanisms to that comparable with compiled procedural 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. The disadvantage of delaying replies is that one loses parallelism. This is an engineering tradeoff.

In-line substitution: By statically identifying the procedure that implements an operation (through static typing or type analysis), 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 [9].

Direct scheduler access: 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 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 [4] 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 8 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.

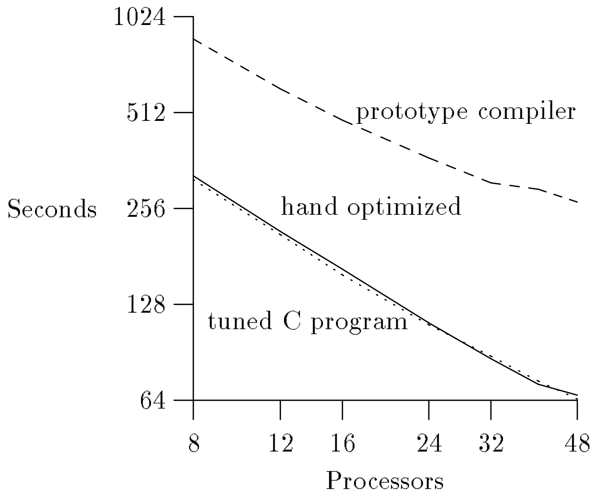

Figure 8: Performance of Gaussian Elimination

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 parallelism 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 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 procedural languages. We conclude that the enormous benefits and reasonable costs of control abstraction argue for its inclusion in explicitly parallel programming languages.

Acknowledgements We thank Michael Quinn and César Quiroz for their comments on drafts of this paper.

References

[1] G. A. Alverson and D. Notkin. Abstracting data-representation and partition-scheduling in parallel progams. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 138–151, April 1991.

[2] G. R. Andrews, R. A. Olsson, M. H. Coffin, I. J. P. Elshoff, K. Nilsen, T. 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.

[3] M. H. Coffin and G. R. Andrews. Towards architecture-independent parallel programming. Technical Report 89–21a, Computer Science Department, University of Arizona, September 1989.

[4] L. A. Crowl. Architectural adaptability in parallel programming. Technical Report 381, Computer Science Department, University of Rochester, May 1991. Ph.D. Dissertation.

[5] A. Goldberg and D. Robson. *Smalltalk-80, The Language and Its Implementation.* Addison-Wesley, 1983.

[6] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[7] M. 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.

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

[9] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[10] B. H. Liskov, M. P. Herlihy, and L. 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.

[11] B. H. Liskov, A. Snyder, R. R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[12] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[13] M. L. Scott. Language support for loosely-coupled distributed programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.