PRELUDE: A System for Portable Parallel Software

by

William Weihl Eric Brewer Adrian Colbrook
Chrysanthos Dellarocas Wilson Hsieh Anthony Joseph
Carl Waldspurger Paul Wang

October 1991

Abstract

In this paper we describe PRELUDE, a programming language and accompanying system support for writing portable MIMD parallel programs. PRELUDE supports a methodology for designing and organizing parallel programs that makes them easier to tune for particular architectures and to port to new architectures. It builds on earlier work on Emerald, Amber, and various Fortran extensions to allow the programmer to divide programs into architecture-dependent and architecture-independent parts, and then to change the architecture-dependent parts to port the program to a new machine or to tune its performance on a single machine. The architecture-dependent parts of a program are specified by annotations that describe the mapping of a program onto a machine. PRELUDE provides a variety of mapping mechanisms similar to those in other systems, including remote procedure call, object migration, and data replication and partitioning. In addition, PRELUDE includes novel migration mechanisms for computations based on a form of continuation passing. The implementation of object migration in PRELUDE uses a novel approach based on *fixup blocks* that is more efficient than previous approaches, and amortizes the cost of each migration so that the cost per migration drops as the frequency of migrations increases.

The current implementation of PRELUDE is built on top of PROTEUS, a configurable simulator that provides both fast and accurate simulations of a wide range of MIMD architectures. PROTEUS itself is a useful tool for developing parallel applications, since it provides powerful non-intrusive debugging and performance monitoring capabilities that are difficult or impossible to obtain on a real machine. Much of the testing, debugging, and initial testing of an application can be accomplished using PROTEUS, typically with less effort than would be required on a real machine. In addition, PROTEUS allows the programmer to test the scalability and portability of a program, including on a range of machine sizes and architectures not supported by available machines. We are using PROTEUS to develop our initial prototype of PRELUDE, and plan to port the implementation of PRELUDE to commercial and research multiprocessors in the near future.

Keywords: Portability, Performance tuning, Annotations, Computation migration, Data migration.

This work was supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988 and by an equipment grant from Digital Equipment Corporation. Individual authors were supported by an Office of Naval Research Graduate Fellowship, a Science and Engineering Research Council Postdoctoral Fellowship, National Science Foundation Graduate Fellowships, an IBM Graduate Fellowship and an AT&T Graduate Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

A number of MIMD multiprocessors are now commercially available, and new large-scale machines are being developed. Given the variety of architectures and the cost of developing large programs, there is a clear need for support for writing high-performance programs that are portable and scalable across a broad range of MIMD architectures. Even on a single multiprocessor machine, better support is needed for writing parallel programs that are both correct and efficient.

In this paper we describe PRELUDE, a programming language and accompanying system support for writing portable MIMD programs. PRELUDE supports a methodology for designing and organizing parallel programs that makes them easier to tune for a particular architecture or to port to a new architecture. Ultimately, we expect to integrate the mechanisms we are developing into parallel versions of existing languages such as C and Fortran. As part of this project, we are also developing debugging and performance monitoring tools to help the programmer debug and tune programs; these are also described briefly in this paper.

Multiprocessors differ in a number of characteristics that affect the performance of parallel programs, including the relative costs of communication, computation, and synchronization; the number of processors; the network topology; and the support provided for shared memory. The problem in achieving reasonable portability is to allow a single program to be mapped onto many different machines without requiring the programmer to make significant changes to the program for each machine.

Portability is related to the problem of performance tuning. The performance of a program on a particular machine can depend on many details of the machine, and can be difficult to predict. Thus, significant tuning may be required to achieve good performance. The mechanisms we propose allow the programmer to separate the description of the computation to be performed by a program from the description of how that computation is to be mapped onto a machine, thus making it easier to tune the performance of a program on a particular machine. This also simplifies porting a program to new architectures. As described in more detail below, our mechanisms integrate and extend the mapping mechanisms proposed in previous systems. Our goal is to provide a comprehensive suite of mapping mechanisms that together give the programmer the flexibility and control needed to map programs efficiently onto particular machines.

Efficiently mapping a program onto a machine involves choosing an appropriate grain size for tasks; determining where to place tasks and data; determining when and where to migrate tasks and data; scheduling tasks; managing communication among tasks; and determining how to cache, replicate and partition data structures. In a distributed-memory message-passing machine, decisions about the placement of data and tasks have a strong impact on the amount of communication required to run a program. Since the cost of sending a message in such machines is typically significantly greater than the cost of accessing local memory, placement decisions can make a large difference in the performance of a program. This is also true in a shared-memory system; a poor job of placement for tasks can result in a large number of cache misses, which also reduces performance.

Existing approaches to managing these issues fall into three classes: those that provide direct, low-level control; those that completely relegate decisions to the compiler and runtime system; and those that allow the programmer to provide directives to the compiler and runtime system, but leave the details of decomposing data structures and tasks to the compiler and runtime system. The first approach is extremely difficult to use, and leads to programs that are difficult to port, precisely because so many architecture-specific decisions are encoded in the program. The second approach is easy to use, but its application to date has been limited to relatively small programs with regular communication patterns, task sizes, and data structures. For numerical programs with irregular data sets and for symbolic programs, purely automatic approaches have not worked well. As a result, we believe that the third approach is the most promising.

Prelude provides the programmer with a computational model based on objects and threads that abstracts away from the underlying architecture, together with high-level annotations that allow the programmer to control the mapping of a program onto a particular machine. Concurrency is expressed explicitly in Prelude, and the programmer is encouraged to write programs with as much concurrency as possible. The Prelude compiler and runtime system then generate an appropriate number of physical threads for the program to run efficiently on a particular machine. The annotations attached to the program are used to describe and control the performance of the program, not its functionality. For example, annotations can be used to control the migration of objects and computation between processors in distributed memory architectures; such migration can yield a significant reduction in message traffic, with a resulting improvement in program performance. Since annotations affect performance but not functionality, the annotations attached to a program can be freely changed without introducing errors into the program; this makes it easy to experiment with different mappings to determine which provides the best performance. This separation of architecture-specific performance-related concerns from the rest of a Prelude program makes it relatively easy to port a program, or to tune its performance.

The Prelude runtime system incorporates novel mechanisms for migrating data and computation in a distributed-memory multiprocessor. We also incorporate flexible mappings of the logical program threads onto the actual physical threads to produce efficient message passing. Existing systems have provided reasonable flexibility in mapping data onto parallel machines (via partitioning, replication, and migration), but have provided only simple mechanisms such as remote procedure calls for mapping logical threads. As described in more detail in Section 4, Prelude is designed to provide flexible control over the *migration* of computation, which allows a logical thread to be mapped onto a number of different physical threads as the computation represented by the logical thread migrates around the machine. In addition, the runtime system supports domain-specific scheduling and load balancing mechanisms that allow the granularity and distribution of tasks to be chosen adaptively at run time based on the characteristics of the architecture and the application load.

Parallel programs are difficult to test, debug, and tune. To accompany PRELUDE, we have built a retargetable simulator, PROTEUS, that provides extremely efficient instruction-level simulation for a wide range of MIMD multiprocessors. Because of its efficiency, accuracy and flexibility, PROTEUS has shown itself to be a useful tool for prototyping, testing, and tuning parallel programs. We have built prototypes of the PRELUDE compiler and runtime system using PROTEUS to evaluate the efficiency of our mechanisms for a variety of MIMD configurations.

In Section 2 of this paper we describe the PRELUDE language. In Section 3 we describe how the annotations supported by PRELUDE provide flexible control over the mapping of a program onto a particular machine, and in Section 4 we discuss the techniques we have developed for implementing the mappings described by the annotations. In Section 5 we describe PROTEUS and its support for language development and prototyping. Finally, in Section 6 we conclude with a discussion of the current status of the project and our plans for future work.

2 The PRELUDE Language

PRELUDE is a statically typed, class-based, object-oriented language with linguistic support for parallel computing. It is lexically scoped and statement based. PRELUDE's computational model is based on two concepts: *objects* and *threads*. Objects contain state and reside in the heap. Each thread maintains a stack and performs sequential computation. Threads can access and modify existing objects, create new objects, and fork new threads. Mechanisms are provided to allow threads to communicate and synchronize.

A PRELUDE object can be *single-threaded* or *multi-threaded*. A multi-threaded object can have multiple active threads performing method invocations on it; a single-threaded object can support only one such thread at a time. Some systems, particularly those based on Actors [Agh86], support only single-threaded objects. We believe that multi-threaded objects are natural and efficient to use in many programs, and that to provide adequate generality and expressive power the system should not restrict the programmer to using single-threaded objects, which forces him to use complex and awkward program structures to achieve the benefits of multi-threaded objects. At the same time, when the programmer intends an object to be single-threaded, the source program is simpler if he does not have to code the required synchronization explicitly using locks; in addition, the required synchronization and scheduling can be implemented more efficiently if the compiler and runtime system know that the object is single-threaded. Thus, we allow the programmer to indicate explicitly as part of a class definition that the class's objects are single-threaded; the compiler then automatically generates the necessary synchronization code.

PRELUDE supports the following constructs for thread creation (variants of the **parfor**, **parbegin** and **fork** constructs have been introduced by other languages including PCF Fortran [For90], BLAZE [MvR87], Occam [Lim84] and SISAL [MSA+85]):

- The parallel **parfor** construct is syntactically similar to a sequential **for** loop. However, each iteration specified by the **parfor** loop is executed by a newly created thread in parallel with the other iterations.
- The **parbegin** construct specifies a set of sequential code blocks; newly created threads execute these blocks in parallel with each other.
- The **fork** construct is used to specify asynchronous invocations of methods and procedures. Asynchronous invocation is described in Section 2.1.
- The **pipe** construct, described in detail in Section 2.1, is used for ordered asynchronous invocations, which run in parallel with the calling thread but are run in the order in which the invocations were made by the caller.

For the **parfor**, **parbegin** and **fork** constructs, PRELUDE supports two types of thread creation: *must* and *maybe*. In *must* creation, the new thread (or threads) is necessary to ensure correctness; an example of *must* creation is an asynchronous call that may deadlock if a separate thread is not forked to perform the invocation. In *maybe* creation, the new thread may improve performance, but does not affect correctness. In PRELUDE, the default is *maybe* creation. The keyword **must** is used if creation is required.

2.1 Asynchronous Invocations

PRELUDE provides three types of invocations: synchronous, unordered asynchronous, and ordered asynchronous. In a synchronous invocation, the calling thread performs the invocation. Unlike synchronous invocations, an asynchronous invocation conceptually forks a new thread to perform the invocation. The calling thread does not necessarily wait for the invocation to finish. There are two kinds of asynchronous calls: unordered and ordered.

Unordered asynchronous invocations avoid the software overhead required to maintain order and are therefore simpler and faster than ordered invocations. However, unordered invocations often result in programs that are difficult to understand and contain subtle race conditions. In many situations a thread can run concurrently with a sequence of calls it makes but these calls must be executed sequentially. Synchronization to achieve this effect can be coded explicitly in the application program, but this leads to complex and less efficient programs. A mechanism for ordered asynchronous calls leads to programs that are both simpler to understand and more efficient than ones in which the ordering is enforced by application-level synchronization. We introduce a new mechanism, pipe objects, to support ordered asynchronous calls.

2.1.1 Unordered Asynchronous Calls

In an unordered asynchronous call, a new thread is forked without any extra synchronization with other threads. Therefore, a thread making a sequence of unordered asynchronous calls to the same receiver object cannot make any assumptions about the order in which the calls are processed. PRELUDE denotes unordered asynchronous calls by preceding the invocations with the **fork** keyword. The invocation returns a *promise*.

The parameterized class **promise**`[T]` refers to a promise for an object of type `T`. Promises [LS88] are similar to futures in MultiLisp [Hal85], except that the value of a promise must be explicitly extracted. (Promises were designed as part of extensions to Argus [LDH+87] for incorporating asynchronous remote procedure calls, as implemented in the Mercury project [LBG+88].) A promise is created by an asynchronous call. For example, an asynchronous call to a procedure that normally returns a type `T` returns the type **promise**`[T]`.

A promise, unlike a future, must be claimed explicitly. For a **promise**`[T]`, the method `claim()` **returns**`(T)` returns the value of the promise, an object of type `T`. Promises also provide a method `ready()` **returns**`(bool)` that indicates whether or not the promise has been filled (and is therefore ready to be claimed).

For example, the following PRELUDE code represents asynchronous calls of method `foo` of object `x` with arguments corresponding to the values of `arg1`, `arg2`,..., `argN`.

```
y: promise[T1] := fork x.foo(arg1, arg2,..., argN)
fork x.foo(arg1, arg2,..., argN)
z: T1 := y.claim()
```

The two calls to `foo` can be run concurrently, and the result of the first call is obtained by the calling thread via the call to `claim` in the third line, which blocks until the result is available.

2.1.2 Ordered Asynchronous Calls

In some circumstances, for example in a pipeline, a sequence of invocations must be performed in order, but can be run in parallel with the calling thread. Previous work by Gifford and Glasser [GG88] and in the Mercury system [LBG+88] has resulted in the design of remote invocation mechanisms for distributed systems in which a sequence of calls between a single sender and a single receiver are run in order, but asynchronously with respect to the caller. We have adapted these ideas for use in PRELUDE; our design provides integrated language support for such ordered asynchronous invocations, and also generalizes the previous work by allowing calls from multiple sending threads to be ordered.

Pipes are special objects used to implement ordered asynchronous calls. The parameterized class **pipe**`[T]` denotes a pipe to an object of type `T`. A pipe is created by the class method **pipe**`[T].new`. If `x` is an object of type `T`, then invoking **pipe**`[T].new(x)` creates and returns a pipe object of type **pipe**`[T]` that provides a mechanism for ordering asynchronous method invocations to object `x`. Objects of type **pipe**`[T]` provide all methods provided by type `T`. However, the return types for these methods are promises: if `T` provides a method `foo(T1)` **returns**`(T2)`, then **pipe**`[T]` provides a method `foo(T1)` **returns**`(`**promise**`[T2])`.

To perform a sequence of ordered asynchronous calls to an object, we merely perform the same sequence of calls in a synchronous manner to one of its pipes; we refer to such calls as "pipe calls". The pipe ensures that pipe calls are processed by the target object in the same order that they are sent. Abstractly, we can view a pipe to object `x` as a forwarder that queues up all calls sent to it and returns promises of the appropriate types. Semantically, it sends the queued calls sequentially to `x`; a call is sent to `x` only after `x` has finished processing the previous queued call. The implementation, described in more detail in Section 4, uses queues at both the sending and the receiving ends (if the caller and the target object are on different processors) so that the delay in the interconnection network affects the computation as little as possible.

A pipe, like any other object, can be passed on to other objects as an argument in a procedure or method invocation; multiple objects and threads can send ordered asynchronous calls through the same pipe object. Also, there can be multiple pipe objects associated with the same target object.

If a calling thread is to perform a sequence of ordered asynchronous calls to a receiving object, it must first obtain a pipe assigned to the receiving object. This can be accomplished either by accessing an existing pipe object assigned to the receiving object, or by creating a new pipe. To perform the sequence of ordered asynchronous calls, the calling thread invokes the same sequence of calls synchronously on the pipe object. For example, suppose two asynchronous method invocations with method names `foo` and `bar` are to be sent to object `x:T`, and the invocation for `foo` must occur before the invocation for `bar`. (`foo` and `bar` return values of types `T1` and `T2`, respectively.) The following PRELUDE code accomplishes this behavior:

```
p: pipe[T] := pipe[T].new(x)
y: promise[T1] := p.foo(arg1, arg2,..., argN)
z: promise[T2] := p.bar(arg1, arg2,..., argM)
```

The pipe ensures that the call to `bar` does not start running on `x` until the call to `foo` has completed. The results of the calls can be obtained by the calling thread (or some other thread that obtains the promises) by claiming the promises returned by the pipe calls.

3 Mapping Annotations

The Prelude language allows concurrency to be expressed independent of architecture-specific constraints. Annotations specify the architecture-specific implementation details that are usually necessary to achieve efficient execution. Previous projects have proposed particular mechanisms for mapping programs onto multiprocessors (e.g., [JHB88, CAL+89, Ben87, Luc87, Li88, PM83, CA89, MG89]), each of which is appropriate for particular kinds of applications and particular kinds of machines. For a system to be effective, we believe that it must support a wide variety of mapping mechanisms efficiently, and must provide flexible support for the user to choose among the different mechanisms. Thus, Prelude supports a wide range of mapping techniques that have appeared individually in other systems. In addition, looking at example applications has made it clear that the mapping techniques that have been proposed so far are inadequate. In particular, existing systems support data mapping via migration, partitioning, and replication (including caching), and support thread mapping via remote procedure call. For some applications, migrating a computation is more effective than moving or replicating the data or accessing it via a series of remote procedure calls. Thus, the design of Prelude includes flexible mechanisms for migrating computations. We also include annotations for specifying scheduling constraints.

Emerald [JHB88] and Amber [CAL+89] provide mechanisms for specifying object location (*locate* object X at node Y), object migration (*move* object X to node Y) and object-object co-location (*attach* object X to object Z). An invocation on an object in Emerald or Amber is always executed at the location of the object, using remote procedure call if the object is remote. The argument objects of a remote invocation can also be moved to the site of the invocation by specifying *call-by-move* parameter passing. Distributed Smalltalk [Ben87], Sloop [Luc87], Ivy [Li88], DEMOS/MP [PM83], Par [CA89] and Comandos [MG89] have migration mechanisms similar to those in Emerald and Amber.

In certain situations neither remote procedure call (commonly referred to as *function-shipping*) nor object migration (commonly referred to as *data-shipping*) is sufficient. We provide additional annotations for *computation* migration, which can be viewed as a form of continuation-passing. These annotations allow us to move the execution of code from one processor to another. For example, the programmer might indicate that when a procedure attempts to invoke a method on a remote object, the execution of the procedure should be moved to the object's location (so that subsequent invocations on the same object become local); this corresponds in the implementation to moving the top frame on the calling thread's stack to the location of the called object. In general, the programmer might indicate that any portion of the top of the thread's stack should be moved, ranging from a part of the top frame (representing part but not all of the remaining computation in the currently active procedure at the top of the stack) to the entire stack (representing the entire remaining computation of the entire thread).

In the rest of this section, we motivate the need for computation migration via an extended example, and then describe the annotations that appear in PRELUDE. The example also illustrates the benefits gained by allowing the mapping of a program onto a machine to be changed easily without affecting the functionality of the program. In the next section, we describe the implementation of the mapping mechanisms.

3.1 An Example: Continuation Passing

We illustrate our mechanisms with a program to implement a concurrent B-tree, an important data structure in high-throughput database systems. The goal of our mechanisms is to allow programmers to write programs in a "shared-memory" programming style (or whatever style makes it easiest to understand the programs) regardless of the physical machine's actual memory model. The resulting programs can then be mapped onto machines so that the performance of the program is comparable to programs with explicit message-passing constructs.

The most efficient concurrent B-tree algorithms known are based on the "link technique," which was introduced by Lehman and Yao [LY81]. The underlying data structure in the link technique is similar to a B+-tree (in which the actual data is stored only in the leaves of the tree), with the modification that each B-tree node contains a pointer to its right neighbor in the tree. In other words, all the nodes at a given level of the tree are linked together from left to right. The links act as "forwarding pointers," and allow processes traversing down the tree from root to leaf to lock only one node at a time.

The PRELUDE code in Figure 1 shows the outline of an implementation of a B-tree class. The representation of a `BTree` is specified by the `anchor`; it contains a reference to the root node. The class method `new` creates a new `BTree` instance. Four instance methods are provided for each class instance. The instance method `locate` is private (and thus is hidden within the scope of the class definition), while the three instance methods `insert`, `delete` and `lookup` are exported.

Wang has extended the Lehman-Yao algorithm so that a process propagating a merge (as well as a split) up the tree locks only one or two nodes at a time [Wan91, WW90]. Operations in Wang's algorithm

```
BTree = class   % link method B-tree

   slots anchor: Anchor end   % a reference to the root of the tree

   class exports new

      new() returns (BTree)
      % class method to create BTree instances
         ......
      end new
   end

   instance exports insert, delete, lookup

      insert(k: Key, d: Data)
      % adds new (Key,Data) pair to the tree
         s: Stack[Node] := locate(k)
         % update leaf and propagate split if necessary
         ......
      end insert

      delete(k: Key)
      % remove entry for Key k and propagate merge if necessary
         ......
      end delete

      lookup(k: Key) returns (Data)
      % returns the Data associated with k
         ......
      end lookup

   % private instance method

      locate(k: Key) returns (Stack[Node])
      % implemented in Figure 2
         ......
      end locate
   end

end BTree
```

Figure 1: The `BTree` class.

can be divided into three phases: the *locate* phase, which finds the appropriate leaf on which to execute the operation; the *decisive* phase, which performs the actual operation on the leaf found in the *locate* phase; and the *update* phase, which propagates any updates to the structure up the tree as needed. The algorithm uses read-write locks on individual nodes to synchronize concurrent operations. The *locate* phase (implemented by the private instance method `locate`) for an `insert` or `delete` method finds, write-locks, and returns the appropriate leaf as the top element on a stack of the nodes visited at each level; it is described by the PRELUDE code in Figure 2. In this code the `child` method of the class `Node` returns the appropriate child of the `Node` instance, while the `right_neighbor` method returns the right neighbor. The `covers` method returns true if and only if the range of keys stored in the subtree rooted at the `Node` instance includes its argument.

```
locate(k: Key) returns (Stack[Node])
   n, next : Node
   s : Stack[Node] := Stack[Node].empty()

   % access root node
   anchor.read_lock()
   n := anchor.root()
   anchor.read_unlock()

   % descend to leaf level
   n.read_lock()
   while ~n.is_leaf() do
      if n.covers(k)
          then next := n.child(k)
               s.push(n)
          else next := n.right_neighbor()
      end
      n.read_unlock()
      n := next
      n.read_lock()
   end

   % find and write-lock appropriate leaf
   n.read_unlock()
   n.write_lock()
   while ~n.covers(k) do
       next := n.right_neighbor()
       n.write_unlock()
       n := next
       n.write_lock()
   end
   s.push(n)
   return(s)
end locate
```

Figure 2: `locate` method.

The `locate` method traverses the tree using read-locks, following links to children and to right neighbors until it reaches a leaf; it then uses write-locks and follows right-links until it finds the leaf responsible for storing the desired key. The stack of nodes constructed in `locate` is used to propagate updates back up the tree during the *update* phase. The *locate* phase for the `lookup` operation is similar, except that it read-locks the leaf, and does not keep track of the nodes visited at each level.

One way of achieving high throughput for a concurrent B-tree is to store different nodes of the B-tree on different processors. To avoid a bottleneck at the root of the tree, it may be necessary to replicate the root node on several processors (a replicated object). Mapping the data structure in this way allows operations on different nodes to run concurrently, and also allows operations on the same node to run concurrently if the node is replicated and the operations only use read-locks. (Algorithms using the link technique are designed to use write-locks infrequently, so replication is likely to be effective, particularly for nodes near the root of the tree.)

Given this mapping of the tree's data structure onto a machine, how should a process executing an operation be mapped onto processors? For the code shown above for the *locate* phase of an `insert` operation, we could choose to run the operation on a single processor, and execute each synchronous invocation of a method as a remote procedure call (RPC) to the processor that stores the appropriate node (commonly referred to as function-shipping). For example, the invocations of the `read_lock`, `read_unlock`, and `child` methods could be RPCs. Systems such as Emerald [JHB88] allow the programmer to choose between this alternative and fetching the data, i.e., moving the B-tree node object temporarily to the processor executing the B-tree operation (commonly referred to as data-shipping).

Neither function-shipping nor data-shipping, however, leads to very good performance. Using function-shipping, the number of messages is very high. For example, the `locate` method shown above makes at least five invocations to access an interior node of the tree. Each invocation will require two messages, giving a total of at least ten messages per node accessed. Using data-shipping, the number of messages could be as few as three per node accessed: two to fetch the node on the first access, and one to send the node back to the processor that stores it after the last access. However, the amount of data sent in the messages will be high, since the entire contents of each node must be transferred between processors.

We could rewrite the program above to reduce the number of messages using function-shipping to two per node accessed. For example, we could rewrite `locate` to invoke a new method `find_successor`, shown in Figure 3, that returns the appropriate successor. If `find_successor` is invoked using RPC, we need only two messages for each interior node accessed. Similar restructuring can reduce the number of messages required to access the anchor and each leaf node to two.

Introducing additional methods and procedures that are called using RPC can improve performance, but has several problems: the resulting program will often be less clear than the original; the transformations appropriate for one machine may be inappropriate for another; and RPC may require more communication than necessary.

```
find_successor(k: Key) returns (Node) signals (leaf)
   next : Node

   self.read_lock()
   if self.is_leaf() then signal leaf end
   if self.covers(k)
       then next := self.child(k)
       else next := self.right_neighbor()
   end
   self.read_unlock()
   return(next)
end find_successor
```

Figure 3: `find_successor` method.

The code for the B-tree operations could be rewritten to use a "continuation-passing" style of communication so that only one message is required for each node accessed, half as many messages as are required by the program in Figure 3. The idea is to send a message to the processor that stores a node when the node is first accessed; that message is processed by calling `find_successor`, and then sending a message to the processor that stores the node returned by `find_successor`. The messages contain the node to be accessed, the key involved, and the node just accessed. The relationship between this message-passing program and the relatively simple program in Figure 2 can be understood by viewing each message as a "continuation" for the `locate` procedure. After a processor has executed the part of the `locate` procedure involved in accessing a B-tree node stored on that processor, it sends a message asking the processor that stores the next node to execute the "rest" of the procedure. The job of executing the procedure is handed from one processor to another until the appropriate leaf node is found, at which point the operation is executed and a message containing the result is sent back to the original caller.

Programs written in this kind of "continuation-passing" style can be very efficient on distributed-memory machines. Indeed, many proponents of Actor-based languages advocate programming in this style, in part to reduce the amount of communication required [Agh86]. However, such programs are usually complex and hard to understand. In addition, they are less efficient on shared-memory machines than programs that use ordinary procedure calls.

Prelude allows a programmer to write a single program that naturally expresses an algorithm and then to choose how to map it onto a machine by writing annotations that indicate how data and threads should be located and moved. Thus, the programmer can easily change the mapping simply by changing the annotations, and can easily experiment with different mappings to determine which gives the best performance. For example, to map the concurrent B-tree program described above onto a distributed-memory using machine using continuation-passing style messages, a **move** annotation can added to the code of Figure 2 as follows:

```
locate(k: Key) returns (Stack[Node]) move
```

The **move** annotation instructs the PRELUDE system that each call in the body of the `locate` method should be called by passing a continuation. A call within the method is compiled so that the call is executed using ordinary stack-based mechanisms when it is on the same processor as the caller. When it is on another processor, however, the system constructs a message containing the top frame of the local stack and sends it to the processor to run the call. When the called method completes, the `locate` method continues running on that processor.

3.2 Annotations in PRELUDE

The ability to express continuation-passing in PRELUDE through simple annotations promotes portable programming styles. In other object-oriented languages the movement of computation would have to be explicitly encoded in the methods; at every point where a continuation should be passed, a new method must be written to represent the continuation. In addition to the **move** annotation, PRELUDE provides annotations for controlling parameter passing, resource management and the movement and placement of objects. In this section, we describe other PRELUDE annotations.

3.2.1 Computation Migration

Our current prototype supports migration of single stack frames, representing the remaining computation of the currently active routine in a thread. Single-frame migration is accomplished by attaching a **move** annotation to the header of a routine or to individual calls in the body of the routine. Attaching the annotation to a particular call indicates that if the call involves a remote object at runtime, the frame for the routine should be moved to the location of the remote object. Attaching the annotation to the header of a routine, as in the B-tree example above, is equivalent to attaching it to every invocation in the body of the routine; it indicates that any call in the body of the routine to a method on a remote object should move the routine's stack frame to the location of the remote object.

In addition to single-frame migration, we are also designing mechanisms and annotations to support migration of partial and multiple frames. This will give the programmer significantly more flexible control over the mapping of the computation represented by logical threads in a program onto physical threads and messages in a machine.

3.2.2 Object Migration

We provide several kinds of annotations to control the location and movement of objects. First, the migration of arguments to invocations can be controlled via annotations. In an object-oriented system the natural parameter-passing method is call-by-object-reference [LG86, GR83]. In a message-passing architecture such semantics can potentially cause additional remote invocations for parameter access. However, PRELUDE objects are mobile. Therefore, additional remote references can be avoided by moving argument objects to the site of the remote invocation. Whether this is worthwhile depends upon the argument object size, the number of invocations of the arguments required, and the costs of mobility and local invocation. Annotations similar to those in Emerald may be used to specify *call-by-move* parameter passing. In this case the parameter object is migrated to the site of the remote invocation.

In addition to call-by-move parameter passing we also provide annotations to describe the movement and placement of objects. Objects can be initially *located at* a given processor and later *moved to* other processors. We can also specify object-object co-location using the *move to* annotation. If X and Y are objects then *move* X *to* Y results in object X being moved to the same processor as object Y.

We are currently exploring additional annotations, based on ideas in the Munin system [BCZ90] and on directives for data placement in Fortran D [F$^+$90], to provide control over replication and partitioning of data. We plan to experiment with these kinds of annotations to understand how they interact with annotations for controlling the migration of objects and computations, and then to build a prototype to explore the problems involved in constructing an integrated system that supports all of these mechanisms efficiently.

3.2.3 Resource Management

In order to develop high-performance concurrent applications, programmers must be able to exert control over resource-management at the application level. We are developing new dynamic resource management mechanisms for a variety of parallel program structures on both shared-memory and distributed-memory multiprocessors.

Our resource management framework is based on a new abstract priority mechanism that encapsulates resource rights. This is related to earlier work with allocation mechanisms that was motivated by microeconomics [WHH$^+$92]. It is coupled with a stable, scalable, hierarchical propagation mechanism that efficiently disseminates information about resource usage and availability. Together, these mechanisms provide an efficient substrate that supports both load balancing and programs that adaptively vary their granularity based on application loads and machine configurations. For example, a task's priority, together with information about loads on various processing resources, can be used by a task management package to make dynamic cost-benefit decisions about whether to fork a subtask or run it inline. The same information can also be used to decide where tasks should be created and executed. Annotations are used to express the priority of tasks to enable the runtime system to make the cost-benefit decisions.

3.3 Summary

A useful system for mapping parallel programs onto parallel machines should provide the programmer with a comprehensive suite of mapping mechanisms. PRELUDE has been designed to integrate mechanisms provided individually in previous systems. In addition, we have designed new mechanisms for controlling the migration of computations. As illustrated by the B-tree example above, these new mechanisms give the programmer important additional expressive power. We are experimenting with our prototype implementation to evaluate the effectiveness of this suite of mapping mechanisms and to understand what additional mechanisms or changes to these mechanisms are needed.

4 Implementation

In this section we discuss the implementation of the annotations and mapping mechanisms described above. We begin by outlining our current implementation of single-frame migration for computations. Then we discuss our implementation of object migration, which uses a novel approach based on *fixup blocks* to achieve better performance than previous approaches. Finally, we discuss the implementation of pipes, and in particular how it interacts with object migration.

4.1 Computation Migration

Our current implementation of the PRELUDE compiler, which is targeted to the PROTEUS simulator, generates C code. As a result, we do not have control over the generated assembly code. Therefore, we treat continuation-passing as an intermediate-language transformation, where we simply create a new method that encapsulates the continuation. Although this is not as efficient (particularly in terms of the size of the generated code) as implementing continuation-passing at a lower level, it allows us to explore the benefits of different continuation-passing annotations.

For a given continuation-passing invocation, we compute the live variables at the invocation; we generate a new method that takes these variables as arguments and represents the execution of the rest of the stack frame. At the point of call, the new method is called if the called object is non-local; otherwise, execution continues locally. Later versions of the PRELUDE compiler will generate assembly language directly. We will then implement continuation-passing by sending the relevant portions of the stack frame and the program counter, which should lead to improved performance and smaller object code.

To date we have implemented migration of single stack frames. However, the programmer may wish to migrate multiple or partial stack frames. For example, in the B-tree implementation the leaf node returned by the `locate` method will have an update method applied to it (either `insert` or `delete`). It may be beneficial to migrate the top two frames on the stack; that for the `locate` method and that for the update method. We are currently designing annotations and the compiler mechanisms needed to implement migration of partial and multiple frames.

4.2 Object Migration

Object migration provides a mechanism to improve performance by improving load balance and reducing communication costs. Migration mechanisms are traditionally expensive and usually reduce the performance of threads even when they do not migrate. In PRELUDE, we follow the philosophy used in Emerald [JHB88]: the *ability* to migrate should have little or no effect on the performance of threads that do not migrate.

Object migration mechanisms fall into two classes: those that depend on location independence of names, and those that translate location-dependent names during migration. Location-independent naming essentially requires a global naming scheme, either through a global address space [CAL+89, Li88] or though higher-level naming mechanisms, which use some form of indirection [PM83, MG89]. On architectures with shared memory, PRELUDE can exploit the global address space to avoid translation. For architectures without a global address space, PRELUDE must use either a high-level naming scheme or translate names during migration. We have chosen the latter, since even though translation increases the cost of migration and message-passing, it significantly improves the performance of threads that access local data [JHB88].

Translation of addresses in objects is straightforward in PRELUDE; the translation of addresses in thread stacks is more challenging. A stack may have frames from several different objects; thus, stack migration in PRELUDE is designed around the migration of individual frames, and a single stack may be spread across several processors. We use stub frames to handle transitions between processors. The stub frames use messages to move arguments and return values between processors.

A useful invariant for performance is the assumption that an executing frame and the object whose method it represents always reside on the same processor. This allows methods to access the object's data directly using pointers, which increases performance. During migration (across processors), these pointers must be found and translated. We are also investigating an alternative technique for the case of large objects, where co-location may be expensive. This relies upon the introduction of invalid addresses for migrated instance variable references and corresponding traps that perform remote accesses.

A second invariant is that objects only migrate when all of their active methods are at *migration points*. This allows a method to assume that its object will not migrate between migration points, which leads to higher performance. Migration points are generally associated with invocation boundaries, so methods must check for migration upon return from an invocation.

To avoid the migration checks at migration points, we have developed a novel approach based on a mechanism called a *fixup block*. A fixup block is a short piece of code (generated by the compiler) that performs the translation for a particular frame; every migration point has a corresponding fixup block. After an object migrates, its active methods, which must be at migration points, resume execution in the fixup block instead of at the instruction following the migration point. The block performs any needed translations and then jumps to the instruction following the migration point. Thus, the migration check is avoided except when an object migrates. Note that fixup blocks are generated automatically and are completely hidden from the user.

To use fixup blocks to translate addresses when objects migrate, we need to be able to find the active methods for a migrating object when the object migrates, so that the return program counters can be set to point to the appropriate fixup blocks. There are (at least) two ways to find the active methods: scan the stacks of all threads, or keep track of the threads (and stack frames) with active methods in each object. Keeping track of the active methods imposes a significant cost even when objects do not migrate, which violates our design goals. Scanning the stacks at migration time imposes a cost only when an object migrates, but the cost could be significant (proportional to the total number of frames on all stacks).

The following technique allows the stacks to be scanned for active methods more lazily. This spreads the cost out over subsequent computation that occurs after the migration, rather than incurring it all when the migration occurs. In addition, it allows the cost of scanning the stacks to be amortized over several migrations; if another object migrates before the stack scans have been completed, the unscanned frames will only be scanned once even though more than one object has migrated. The idea is to change the top frame on each stack so it resumes at its fixup block; after performing any necessary translations, that fixup block changes the return program counter for the previous frame on the stack so it will resume at its fixup block when the top frame returns. The cost when an object migrates is proportional to the number of threads, not the total number of stack frames, and if several objects migrate before a frame is scanned, the frame will be translated only once.

4.3 Pipes

This section describes our implementation of pipe objects. A pipe is made up of two objects, a head and a tail. Typically the head of a pipe will be located on the same processor as the thread making calls on the pipe, and the tail will be located on the same processor as the target object of the pipe. (If either the calling thread or the target object migrates, the corresponding end of the pipe should also be moved.) The head and tail objects both contain queues of pending invocations; buffering invocations at both ends of the pipe allows the communication delay of the interconnection network to be largely hidden from the computations using the pipe.

To preserve the order of the invocations sent from the head to the tail we adopt one of two approaches depending upon the underlying network architecture. In networks that do not preserve message order we simply use sequence numbers in the messages for the invocations. In networks where the message order between processors is maintained, we can avoid the sequence numbers and the associated overhead of maintaining them and checking them. This is done as follows. As long as neither the head nor the tail of a pipe migrates, invocations can be streamed from the head to the tail and enqueued at the tail as they are received, and order will be preserved. If the head migrates, we send a special marker message from the old head to the tail after the last message; the tail waits until the marker has been received before accepting messages from the new head. Similarly, if the tail migrates, the old tail forwards messages received from the head to the new tail, and informs the head that the tail has migrated. The head then sends a marker to the old tail and starts sending messages to the new tail, which delays accepting messages from the head until the marker is forwarded from the old tail.

5 Support for Prototyping

Critical to the success of PRELUDE as a vehicle for studying languages and runtime systems is the ability to experiment on a wide variety of MIMD architectures. To facilitate such experiments we have built a retargetable simulator, PROTEUS [Del91, Bre91, BDCW91], that simulates MIMD architectures and provides support for sophisticated data collection and display.

PROTEUS simulates MIMD multiprocessors in which independent processor nodes are connected via an interconnection medium. The interconnection medium can be either a bus, a direct network such as a k-ary n-cube, or an indirect network such as a butterfly. Each processor node consists of a processor, a network module, a cache module, and memory. Conceptually, the processor is a generic sequential processor extended with instructions for network access and cache coherence. The network module interfaces the processor with the interconnection medium. The cache module, which is optional, handles cache coherence and works with the network module for remote memory accesses.

The modules can be refined or replaced to provide more accurate simulation of a particular architecture. It is also useful to have multiple implementations of a module that provide different performance/accuracy tradeoffs. For example, for k-ary n-cubes, we use two implementations: one that is very accurate and simulates each packet hop by hop, and a second that uses an analytical model to compute the arrival time at the target. Although the model version is clearly less accurate, it is also an order of magnitude faster. When network accuracy is less important, such as during development, we use the model version to exploit the higher performance.

PROTEUS applications and modules are written in an extended version of C. Thus, our runtime system is written in C and the PRELUDE compiler generates C as its output. We then link the PROTEUS engine, the runtime system, and the compiled PRELUDE application into an executable that simulates the application on the specified architecture. The compiler and runtime system are designed so that they should be relatively easy to port to a machine with a C compiler; we plan to port them to nCUBE and Intel machines in the near future.

Since the combination of PRELUDE and PROTEUS is intended as a base for studying language and runtime-system issues, extensive support for debugging and monitoring is an important requirement. PROTEUS provides *nonintrusive* monitoring and debugging: users can add monitoring code that does not affect the behavior or timing of the simulation. It also provides *repeatability*: users can rerun simulations to pinpoint bugs. Real multiprocessors generally provide neither of these abilities.

Nonintrusive monitoring, combined with repeatability, greatly simplifies the development of concurrent programs. Real multiprocessor systems suffer from the *probe effect*: the addition of monitoring code may cause the monitored effect to disappear [Gai86]. This prevents programmers from collecting additional data for debugging. PROTEUS allows users to add arbitrary monitoring or debugging code without changing the behavior of the simulation.

Nonintrusive monitoring is only useful if the platform ensures repeatability: the whole point of nonintrusive monitoring is to allow repeatability in the presence of additional code. Nondeterministic systems, such as multiprocessors, rarely provide any form of repeatability; some bugs may occur only once every thousand (or more) executions, which makes it nearly impossible to track them down. Repeatability is perhaps the single most important feature of PROTEUS; its presence provides a debugging environment that is generally not available on real multiprocessors.

The PRELUDE/PROTEUS combination has been designed to work well with sequential debuggers. This extends the power of advanced sequential debuggers to the parallel development arena. Furthermore, PROTEUS provides an internal debugging mode that allows users to examine the state of threads, processors, locks, and memory. Using a sequential debugger such as `dbx` [Lin90] together with PROTEUS results in a very effective development environment.

While we designed PROTEUS initially as a substrate for experimenting with prototype language, compiler, and runtime system mechanisms, it has become clear that a simulator such as PROTEUS can also be a useful tool for developing parallel applications. The monitoring capabilities provided by PROTEUS can make debugging and initial performance tuning significantly easier than on a real machine. Also, PROTEUS can be run on uniprocessor workstations so that the time required on expensive and scarce multiprocessors is less. In addition, the ability of an application program to scale to large machine sizes (perhaps beyond the range provided by available machines) or to be ported effectively to a range of machines can be investigated fairly easily.

6 Conclusions

The current PRELUDE implementation supports only single-frame migration for computations. We are currently designing and implementing mechanisms and annotations for multiple- and partial-frame migration. In addition, we are studying how to add annotations for data replication and partitioning, based on ideas in Munin [BCZ90] and in Fortran D [F$^+$90].

The implementation of object migration in PRELUDE uses a novel approach based on *fixup blocks*. Fixup blocks eliminate the need to check at each migration point whether an object has migrated. Instead, at migration time the stack for each thread is modified so that frames resume at a fixup block, which does the appropriate checks and address translations. A further optimization allows the modifications to the stacks to be spread out over time as each frame returns to its caller, which reduces the cost per migration as the frequency of migrations increases.

PRELUDE is currently implemented on top of PROTEUS, a configurable simulator that provides both fast and accurate simulations of a wide range of MIMD architectures. PROTEUS itself is a useful tool for developing parallel applications, since it provides powerful non-intrusive debugging and performance monitoring capabilities that are difficult or impossible to obtain on a real machine. Much of the testing, debugging, and initial testing of an application can be accomplished using PROTEUS, typically with less effort than would be required on a real machine. In addition, PROTEUS allows the programmer to test the scalability and portability of a program, including a range of machine sizes and architectures not supported by available machines. We are using PROTEUS to develop our initial prototype of PRELUDE, and plan to port the implementation of PRELUDE to commercial and research multiprocessors in the near future. We have also used PROTEUS for a number of algorithmic and architectural studies.

PRELUDE allows the programmer to write programs using an abstract model of computation that is independent of any particular underlying architecture. A program can then be mapped onto a particular machine by attaching annotations to it that describe the mapping. Our goal in PRELUDE is to provide a comprehensive suite of mapping mechanisms that give the programmer sufficient power to implement a wide range of parallel programs efficiently on a wide variety of MIMD architectures. To this end, we have included many mapping mechanisms that have appeared in other systems, including remote procedure call, object migration, and data replication and partitioning. In addition, however, PRELUDE include novel migration mechanisms for computations based on a form of continuation passing. We are experimenting with our current implementation to evaluate the effectiveness of our current suite of mapping mechanisms and to understand what other mechanisms or changes to our current mechanisms are needed.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[BCZ90] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, March 1990.

[BDCW91] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, 1991.

[Ben87] J.K. Bennett. The design and implementation of Distributed Smalltalk. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 318–330, 1987.

[Bre91] E.A. Brewer. Aspects of a high-performance parallel-architecture simulator. Master's thesis, MIT Laboratory for Computer Science, 1991.

[CA89] M.D. Coffin and G.R. Andrews. Towards architecture-independent parallel programming. Technical Report 89-21a, Department of Computer Science, University of Arizona, December 1989.

[CAL+89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, April 1989.

[Del91] C.N. Dellarocas. A high-performance retargetable simulator for parallel architectures. Master's thesis, MIT Laboratory for Computer Science, 1991.

[F^+90] G. Fox et al. Fortran D language specification. Technical Report COMP TR90-141, Rice University, Dept. of Computer Science, December 1990.

[For90] Parallel Computing Forum. PCF Fortran Proposed Standard, 1990. Version 3.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[GG88] David K. Gifford and Nathan Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[GR83] A. Goldberg and D. Robson. *Smalltalk80: The language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[Hal85] R. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[JHB88] E. Jul, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[LBG^+88] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, January 1988. Available as MIT LCS Programming Methodology Group Memo 59.

[LDH^+87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens (editor), R. Scheifler, and W. Weihl. Argus reference manual. Technical Report MIT/LCS/TR-400, MIT Laboratory for Computer Science, November 1987.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[Li88] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, pages II78–86, 1988.

[Lim84] INMOS Limited. *Occam Programming Manual.* Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[Lin90] M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 USENIX Summer Conference*, pages 211–220, June 1990.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 260–267, 1988.

[Luc87] S.E. Lucco. Parallel programming in a virtual object space. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 26–33, 1987.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[MG89] J.A. Marques and P. Guedes. Extending the operating system to support an object-oriented environment. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, pages 113–122, 1989.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL language reference manual. Technical report, Lawrence Livermore National Laboratory, March 1985.

[MvR87] P. Mehrotra and J. van Rosedale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[PM83] M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 110–119, 1983.

[Wan91] P. Wang. An in-depth analysis of concurrent B-tree algorithms. Master's thesis, MIT, January 1991.

[WHH+92] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, February 1992. *to appear.*

[WW90] W.E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, 1990.