Portable software for multiprocessor systems

In this article we describe Prelude, a programming language and accompanying system support for writing portable parallel programs for multiprocessor architectures. Prelude allows 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. This makes it easier to tune the performance of a program on a particular machine and also simplifies porting a program to new architectures.

by Adrian Colbrook, William E. Weihl, Eric A. Brewer, Chrysanthos N. Dellarocas, Wilson C. Hsieh, Anthony D. Joseph, Carl A. Waldspurger and Paul Wang

Large-Scale Parallel Software Research Group,
MIT Laboratory for Computer Science

Introduction

The mainstream use of computers has increased in sophistication to a point where single processor systems are unable to cope with performance demands. Multiprocessor systems, in which several processing elements execute in parallel, are now the most feasible means of achieving the required performance improvements. Multiprocessors are already in use in scientific applications, including aerodynamics, astrophysics, biology, computer science, chemistry, engineering, geophysics, material science, nuclear physics and plasma physics. They also have a range of industrial applications such as in the oil industry, automobile manufacturing and pharmaceuticals.

The enabling technology for multiprocessor systems arises from the remarkable progress made in microelectronics. The increasing density and corresponding functionality of chips has allowed the cost-effective construction of printed circuit computers incorporating several megabytes of memory and the ability to execute millions of instructions per second.

The advances in microelectronics have given rise to a number of commercial and research-based multiprocessor systems. Existing multiprocessor systems can be classified into two broad architectural areas: *single instruction stream multiple data* (SIMD) and *multiple instruction stream multiple data* (MIMD). In SIMD architectures the same program instructions are executed simultaneously by every processor on different data. In MIMD architectures, each node executes a separate instruction stream.

Given the variety of architectures and the high cost of developing large programs for these architectures, there is a clear need for support for writing high-performance programs that are portable and scalable across a broad range of multiprocessor architectures. Even on a single multiprocessor machine, better support is needed for writing parallel programs that are both correct and efficient.

In this article we describe Prelude, a programming language and accompanying system support for writing portable programs for MIMD multiprocessor systems. Prelude supports a methodology for designing and organising parallel programs that makes them easier to tune for a particular architecture or to port to a new architecture. We present a high-level description of some of the novel mechanisms provided by Prelude rather than giving a detailed description of the language syntax and semantics. Ultimately, we expect to integrate the mechanisms we are developing into parallel versions of existing languages such as C and Fortran.

MIMD multiprocessors can be divided into two broad categories; *shared-memory multiprocessors* and *distributed-memory multiprocessors*. In shared-memory multiprocessors, the processors share the primary memory via a connection network (which is often a common bus). Interprocessor communication and synchronisation in these architectures are usually via shared-memory operations. In distributed-memory architectures, each processor has its own local memory that is not shared. Interprocessor communication and synchronisation in these architectures is achieved via explicit message passing between the processors. However, the distinctions between these two categories are becoming increasingly vague, with shared-memory architectures supporting message passing and distributed-memory architectures supporting virtual shared memory.

In this article we are particularly concerned with the implementation of Prelude on distributed memory MIMD architectures. Prelude programs are usually implemented in a shared memory style (with no explicit message passing). It is the concern of the language compiler and runtime system to map these programs efficiently onto distributed-memory architectures.

Multiprocessors differ in a number of characteristics that affect the performance of parallel programs, including: the relative costs of communication, computation and synchronisation; 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 of 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 a variety of machines.

Efficiently mapping a program onto a multiprocessor involves: choosing an appropriate size for the concurrent tasks; determining where to place tasks and data; determining when and where to migrate tasks and data; scheduling the execution of tasks; managing communication among tasks; and determining how to cache, replicate and partition data structures. For example, in a distributed-memory message-passing multiprocessor, 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.

The performance of a program on a particular machine can depend on many details of the machine, and can be difficult to predict

Prelude provides high-level annotations that allow the programmer to control the mapping of a program onto 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 in the multiprocessor 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. 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 processors in the machine.

Parallel programs are difficult to test, debug and tune. To accompany Prelude, we have built a retargetable simulator, Proteus,[1] that provides extremely efficient instruction-level simulation for a wide range of 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 multiprocessor configurations.

In the next section of this article we describe the Prelude language. The following section describes how the annotations supported by Prelude provide flexible control over the mapping of a program onto a particular machine.

The Prelude language

Prelude is an object-oriented language with linguistic support for parallel computing. Prelude provides the programmer with a computational model based on objects and threads that abstracts away from the underlying architecture. An object is a self-contained entity that encapsulates state. For example, in a banking system the objects might be the different customer accounts in the system and the 'state' of each account would be the current balance and a record of recent transactions.

A thread is simply a sequence of statements that are executed sequentially. Concurrent programs contain some number of threads that execute in parallel. Threads can invoke methods on objects, create new objects and fork new threads. For example, account objects in the banking system may provide 'deposit' and 'withdraw' methods that can be invoked on an account object by a thread. Prelude also provides mechanisms that allow threads to communicate and synchronise.

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[2] 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 synchronisation explicitly using locks; in addition, the required synchronisation 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 whether an object is single-threaded; the compiler then automatically generates the necessary synchronisation 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,[3] BLAZE,[4] occam[5] and SISAL[6]):

- 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.
- The *pipe* construct[7] 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.

```
x: account      % declares  x to be an account object
balance: int
success: bool

balance := x. deposit(50)
......           % an arbitrary code segment represented simply as K
success := x.withdraw(30)
```

Fig. 1 Prelude code that synchronously invokes methods on an account object

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,[8-15] 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 variety of mapping mechanisms efficiently, and must provide flexible support for choosing among the different mechanisms.

Emerald[8] and Amber[9] 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,[10] Sloop,[11] Ivy,[12] DEMOS/MP,[13] Par,[14] and Comandos[15] 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. 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. We provide additional annotations for *computation* migration. These annotations allow us to move the execution of code from one processor to another. In this section we illustrate the different invocation techniques for remote data supported by Prelude for distributed-memory multiprocessor systems. We also describe the other annotations that Prelude provides.

We begin by considering a simple piece of Prelude code, shown in Fig. 1, that performs synchronous invocations on an account object. The code first deposits $50 into the account x by invoking the method x.deposit. This method returns the current balance of the account after the deposit has been made. We then execute some code segment containing only invocations on local objects (represented by K) and then perform a second invocation, this time x.withdraw. This invocation attempts to withdraw $30 from the account. If there are sufficient funds in the account then the withdrawal is made and the invocation returns True. Otherwise, no changes are made to the account balance and the invocation returns False.

Assume that a thread on processor P_0 executes the code segment in Fig. 1 and that the object x is located on processor P_1 in a distributed memory MIMD architecture. A processor gains access to remote objects through explicit message passing. Prelude invocations are location independent, it is the responsibility of the Prelude compiler to generate the code that determines the relative location of threads and objects and performs the appropriate local or remote invocations. In this case, the compiler and runtime system can choose to perform the remote invocations using remote procedure calls, data migration or computation migration. We now consider each of these alternatives in turn.

Invocations using remote procedure calls

Fig. 2 represents the execution of

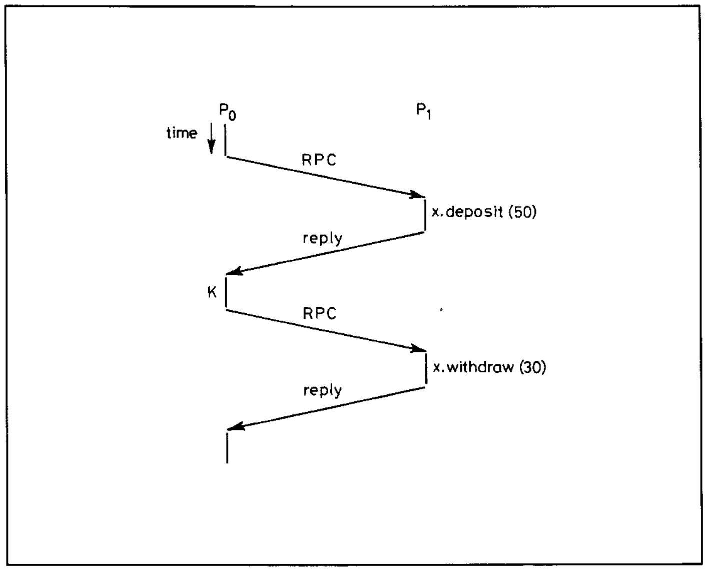


Fig. 2 Execution of the code segment using remote procedure calls (RPCs)

the code segment in Fig. 1 using remote procedure calls. Processor P_0 sends an invocation message to processor P_1. The code for x.deposit(50) is executed on P_1 and the reply message containing the new account balance is sent back to P_0. The code segment K on P_0 is executed and then a remote invocation is performed for x.withdraw(30).

In this example, there are two remote procedure calls each of which requires two messages. Each invocation message contains references to the object (x in this case) and the method to be invoked, the arguments for the invocation and an address for the reply value.

Invocations using data migration

Fig. 3 represents the execution of the code segment in Fig. 1 using data migration. In this case the object x moves (or *migrates*) from P_1 to P_0. Processor P_0 first sends a migration request message to processor P_1. P_1 then migrates x to P_0 and then x.deposit, the code segment K and x.withdraw are executed locally by P_0.

In this example, object migration requires two messages, the first to request migration and the second to migrate the object itself. The size of the message containing the migrating object is proportional to the amount of state that must be migrated in order to reconstruct the object at the destination. Object migration also involves address translation on architectures without a global address space. A description of the implementation of this translation process is beyond the scope of this article; the interested reader is referred to Reference 16.

Invocations using computation migration

When a thread attempts to invoke a method on a remote object, the execution of the thread could be moved to the object's location (so that subsequent invocations on the same object become local). We term this *computation migration* and this corresponds in the implementation to moving the top frame on the calling thread's stack to the location of the called object.

Fig. 4 represents the execution of the code segment in Fig. 1 using computation migration. Processor P_0 executes migration code that moves the executing thread from P_0 to processor P_1; x.deposit, the code segment K and x.withdraw are then executed by P_1.

The size of the migration message depends on the amount of state required by the thread. Note that only the top stack frame is migrated in this case so that control eventually returns to the thread on P_0 once the invocation associated with the top frame has completed. We have assumed that the application code is replicated on every processor in the system. If this is not the case then the code associated with top stack frame must also be migrated.

We now consider an extended example to highlight the differences between these implementations for remote invocations.

Extended example: a concurrent B-tree

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

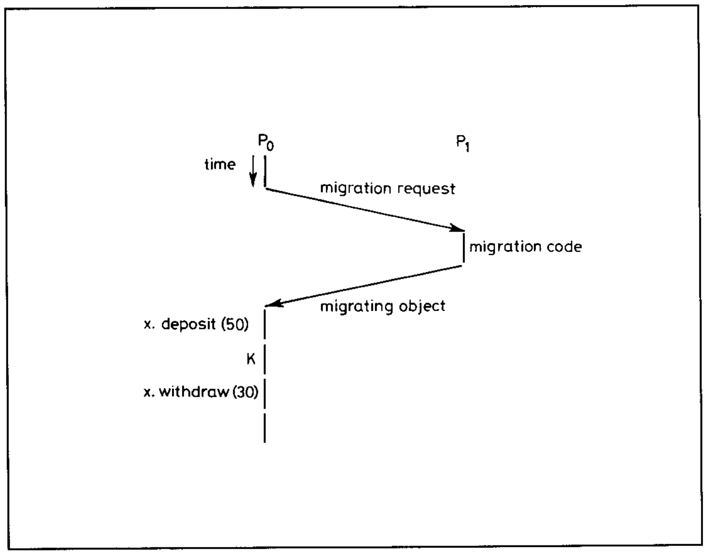


Fig. 3 Execution of the code segment using data migration

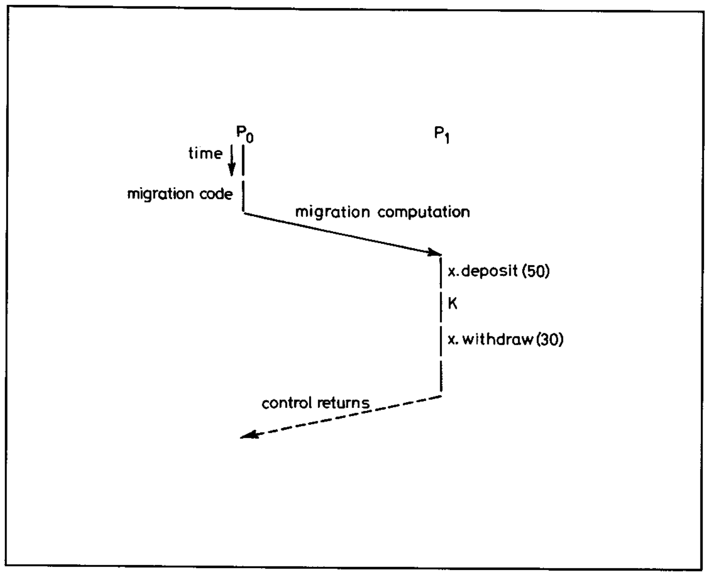


Fig. 4 Execution of the code segment using computation migration

programs with explicit message-passing constructs.

Search trees are data structures that support many dynamic-set operations, including search, insert and delete. A B-tree is a balanced search tree in which every branch node of the tree has many children. The interested reader is referred to References 17 and 18 for a complete description of the B-tree algorithm used in this example.

Operations on the B-tree 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 synchronise concurrent operations. Independent operations may concurrently acquire the same lock-in read mode. However, a thread can acquire a lock-in write mode only if no other thread has acquired the lock in either read or write mode.

Prelude code to implement the locate phase for an insert operation on a B-tree is given in Fig. 5. The code traverses the tree from the root using read-locks, until a leaf node is reached.

One way of achieving high throughput for a concurrent B-tree is to store different nodes of the B-tree on different processors. Mapping the data structure in this way allows operations on nodes to run concurrently.

Given this mapping of the tree's data structure onto a machine, how should a thread executing a B-tree operation be mapped onto processors? We could choose to run the operation on a single processor, and execute each synchronous invocation of a method as a remote procedure call to the processor that stores the appropriate node. For example, the invocations of the is_leaf, read_lock, successor and read_unlock methods on a remote node could be remote procedure calls (commonly referred to as function-shipping). Alternatively, we could choose to use data migration and move each B-tree node object accessed 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 in this case. Using function-shipping, the number of messages is very high. For example, the fragment of the insert method shown in Fig. 5 makes four invocations to access an interior node of the tree. Each invocation requires two messages, giving a total of eight messages per node accessed. Using data-shipping, the number of messages could be as few as two per node accessed; however, the amount of data sent in the messages will usually be high, since the entire contents of each node must be transferred between processors.

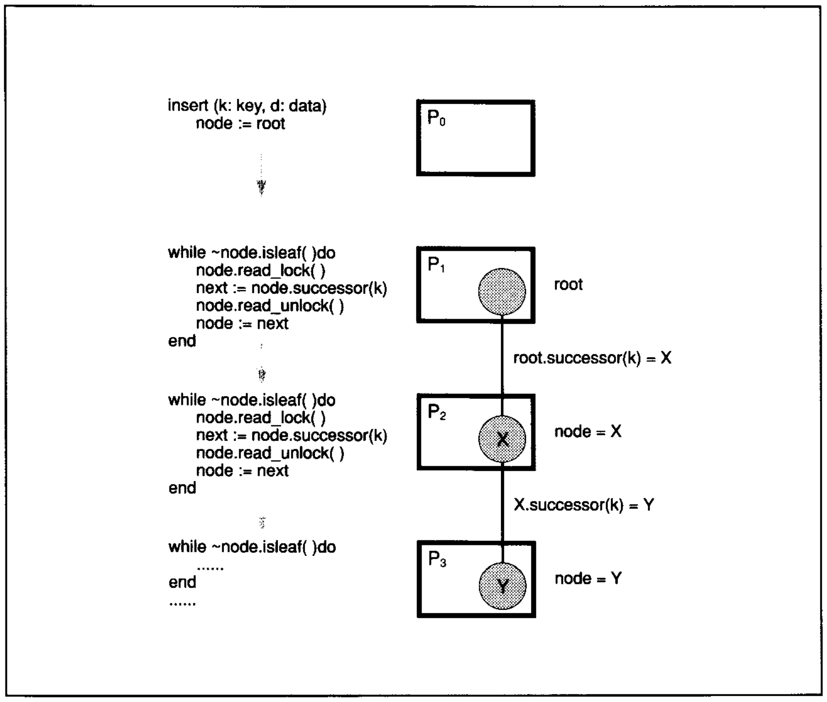
Alternatively, we could choose to use computation migration. Fig. 6 shows an example execution of the code given in Fig. 5 using computation migration. In this example, the thread executing the code begins on processor P_0 and the root node is stored on processor P_1. When an invocation is made on a node object that is not local the computation moves to the location of the node object and continues execution. In this example, the computation moves from P_0 through to P_3 as the tree is traversed. One message is required for each node accessed and it contains the key and data values together with the reference for the node to be accessed. This leads to an implementation with fewer messages than one using function-shipping and shorter messages than one using data-shipping.

Programs written explicitly to use this kind of 'computation migration' style (often referred to as continuation passing) can be very efficient on distributed-memory machines. Indeed, many proponents

```
insert (k: key, d: data)
  ......
  node := root
  while ~node.is_leaf( ) do
      node.read_lock( )
      next := node.successor(k)
      node.read_unlock( )
      node := next
  end
  ......  % decisive and update phases follow
end insert
```

Fig. 5 The locate phase for the insert method for the B-tree

Fig. 6 Execution of the insert method using computation migration



of Actor-based languages advocate programming in this style, in part to reduce the amount of communication required.[2] 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 machine using computation migration messages, a *move* annotation can added to the code of Fig. 5 as follows:

```
insert (k: key, d: data) move
```

The *move* annotation instructs the Prelude system that each call in the body of the insert method should be implemented by computation migration. 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 insert method continues running on that processor.

Thus Prelude allows programs to be coded using computation migration while still preserving the clarity of code that uses remote procedure calls. Furthermore, the *move* annotation has no effect on the correctness of the program. If the program is moved to an architecture on which remote procedure calls are more efficient than computation migration, the annotation is simply removed and invocations are then performed by remote procedure calls.

Annotations in Prelude

The ability to express computation migration in Prelude through simple annotations promotes portable programming styles. In other object-oriented languages the movement of computation would have to be encoded explicitly in the methods. In addition to the move annotation, Prelude provides annotations for controlling parameter passing, resource management and the movement and placement of objects.

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.[19,20] In a message-passing architecture such semantics may 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 on 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 an annotation.

We are currently exploring additional annotations, based on ideas in the Munin system[21] and on directives for data placement in Fortran D,[22] 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 these mechanisms efficiently.

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.

Conclusions

In this article we have described Prelude, a programming language and accompanying system support for writing portable programs for MIMD multiprocessor systems. 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. Since annotations affect performance, but not functionality, the annotations attached to a program can be freely changed without introducing errors into the program. 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.

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, Prelude includes novel migration mechanisms for computations. Programs can be coded using computation migration while still preserving the clarity of code that uses remote procedure calls.

We are experimenting with our current implementation to evaluate the effectiveness of our suite of mapping mechanisms and to understand what other mechanisms or changes to our current mechanisms are needed. Ultimately, we expect to integrate many of these mechanisms into versions of existing languages such as C and Fortran.

Acknowledgments

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 a Science and Engineering Research Council Postdoctoral Fellowship, an Office of Naval Research Graduate Fellowship, National Science Foundation Graduate Fellowships, an IBM Graduate Fellowship and an AT&T Graduate Fellowship.

References

1 BREWER, E. A., DELLAROCAS, C. N., COLBROOK, A., and WEIHL, W. E.: 'Proteus: a high-performance parallel-architecture simulator', Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, 1991

2 AGHA, G.: 'Actors: a model of concurrent computation in distributed systems' (MIT Press, Cambridge, MA, 1986)

3 Parallel Computing Forum: PCF Fortran Proposed Standard, 1990. Version 3

4 MEHROTRA, P., and VAN ROSEDALE, J.: 'The BLAZE language: a parallel language for scientific programming', *Parallel Computing*, November 1987, **5**, pp.339-361

5 INMOS Ltd.: 'Occam programming manual (Prentice Hall, Englewood Cliffs, New Jersey, 1984)

6 McGRAW, J., SKEDZIELEWSKI, S., ALLAN, S., OLDEHOEFT, R., GLAUERT, J., KIRKHAM, C., NOYCE, W., and THOMAS, R.: 'SISAL language reference manual', Technical report, Lawrence Livermore National Laboratory, March 1985.

7 COLBROOK, A., BREWER, E. A., HSIEH, W. C., WANG, P., and WEIHL, W. E.: 'Pipes: linguistic support for ordered asynchronous invocations', Technical Report MIT/LCS/TR-539, MIT Laboratory for Computer Science, 1992

8 JUL, E., HUTCHINSON, N. and BLACK, A.: 'Fine-grained mobility in the Emerald system', *ACM Transactions on Computer Systems*, 1988, **6**, (1), pp.109-133

9 CHASE, J. S., AMADOR, F. G., LAZOWSKA, E. D., LEVY, H. M., and LITTLEFIELD, R. J.: 'The Amber system: Parallel programming on a network of multiprocessors', Technical Report 89-04-01, Department of Computer Science, University of Washington, April 1989

10 BENNETT, J. K.: 'The design and implementation of distributed smalltalk', *in* Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference, 1987, pp.318-330

11 LUCCO, S. E.: 'Parallel programming in a virtual object space', *in* proceedings of the Object-Oriented Programming Systems Languages and Applications Conference, 1987, pp.26-33

12 LI, K.: 'Ivy: A shared virtual memory system for parallel computing', *in* Proceedings of the International Conference on Parallel Processing, 1988, pp.1178-86

13 POWELL, M. L., and MILLER, B. P.: 'Process migration in DEMOS/MP', *in* Proceedings of the Ninth ACM Symposium on Operating System Principles, 1983, pp.110-119

14 COFFIN, M. D., and ANDREWS, G. R.: 'Towards architecture-independent parallel programming', Technical Report 89-21a, Department of Computer Science, University of Arizona, December 1989

15 MARQUES, J. A., and GUEDES, P.: 'Extending the operating system to support an object-oriented environment', *in* Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference, 1989, pp.113-122

16 WEIHL, W., BREWER, E., COLBROOK, A., DELLAROCAS, C., HSIEH, W., JOSEPH, A., WALDSPURGER, C., and WANG, P.: 'Prelude: a system for portable parallel softare', Technical Report MIT/LCS/TR-519, MIT Laboratory for Computer Science, 1991

17 WANG, P.: 'An in-depth analysis of concurrent B-tree algorithms', Technical Report MIT/LCS/TR-496, MIT Laboratory for Computer Science, January 1991

18 WEIHL, W. E., and WANG, P.: 'Multi-version memory: software cache management for concurrent B-trees', in Proceedings of the 2nd IEEE Symposium on Parallel and distributed Processing, 1990, pp.650-655

19 LISKOV, B., and GUTTAG, J.: 'Abstraction and specification in program development' (MIT Press, 1986)

20 GOLDBERG, A., and ROBSON, D.: 'Smalltalk80: the language and its implementation' (Addison-Wesley, Reading, MA, 1983)

21 BENNETT, J., CARTER, J., and ZWAENEPOEL, W.: '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

22 FOX, G. *et al.*: 'Fortran D language specification', Technical Report COMP TR90-141, Rice University, Department of Computer Science, December 1990

At the time this article was written, all the authors were with the Large Scale Parallel Research Group, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, USA.