A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters

W. W. Gropp and E. L. Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Abstract

The basic processing element, from PCs to large systems, is rapidly becoming a symmetric multiprocessor (SMP). As a result, the nodes of a parallel computer will often be an SMP. The resulting mixed hardware models (combining shared-memory and distributed memory) provide a challenge to system software developers to provide users with programming models that are portable, understandable, and efficient. This paper describes and compares a variety of programming models for a parallel computer made up of SMP nodes.

1 Introduction

The first commercial parallel computers were shared-memory machines. As time passed, it was found that the cost benefits of large-scale parallelism provided an advantage to architectures that were more scalable than the bus-based shared-memory machines, and high-performance computing came to be dominated by message-passing machines like the Intel iPSC/860 and Paragon, the IBM SP1 and SP2, the TMC CM-5, and the Cray T3D. Now, however, parallel versions of RISC-based workstations are giving the shared-memory paradigm a new life, with scalability provided by interconnecting multiple such shared-memory machines (symmetric multiprocessors, or SMP's) with high-speed (or even low-speed) networks and switches. Thus we are approaching a situation in which the hardware model presented to a programmer consists of a cluster of SMP's, with shared memory provided for inter-process communication on a single SMP, and message-passing provided for inter-process communication between SMP's. This same model is being presented from three different directions:

- High-speed networks connecting large SMP's. Such clusters compete with traditional MPP's (massively parallel processors).
- Small SMP nodes connected by high-speed switches. These are the next generation of MPP's.
- Small SMP workstations and multiprocessor PC's on inexpensive networks. These are the next generation of today's workstation networks.

Although different, all of these platforms provide the same system model combining shared memory with message passing. Their operating systems may or may not provide aspects of advanced programming environments: virtual shared-memory across the cluster, threads within a cluster, remote memory operations such as `put/get`, but are very likely to provide the minimum components in some form: processes, memory accessible by more than one process, locks to protect variables in shared memory, messages to provide data transfer from one process's private memory to another's. We focus here on what can be accomplished with this basic set of tools, and consider higher-level constructs to be built on these.

The idea of combining the shared-memory and message-passing models is not new. It has been described in [1] and implemented in a widely-available programming system [2]. However, the current computing environment, with this programming model becoming available in so many different ways, gives the topic renewed importance.

In Section 2, we define the terms we will use in our discussion of programming models. Section 3 briefly describes the criteria we will use in evaluating programming models, and Section 4 lists components of the models we will examine. Section 5 contains a rough classification of programming models for SMP clusters and some commentary on each model with regard to the criteria discussed in section 3. Section 6 draws some conclusions.

2 Common Programming Models

Since many of the terms we use in this discussion mean different things to different people, in this section we define our basic vocabulary.

2.1 Processors

In the following discussion, a *processor* is a CPU, capable of executing a program. A processor may or may not be individually addressable or controllable. An example of such a processor might be a CPU in a symmetric multiprocessor. That is, we will say that a single symmetric multiprocessor with four CPU's, capable of running four Unix processes simultaneously, has four processors.

2.2 Processes and Threads

A *process* has its own address space, together with a single program counter and stack. A Unix process is a good example. A single process may contain multiple *threads*. A thread has its own stack and program counter, but shares the rest of memory with other threads in the same process. On shared-memory machines, there is often a way for processes to obtain a pointer to a block of memory that is shared with other processes, although such mechanisms are highly nonstandard. Conversely, there is a way for threads to acquire private memory. The Posix "pthread" standard provides a standard, portable way to do this, although not all thread systems are Posix-compliant.

Processes are scheduled for execution on a single *processor* by the operating system. The cost of switching execution on that processor from one process to another is generally considered to be large, because of the cost of changing address spaces. Threads are scheduled within a single process with a variety of methods, which differ in cost on different systems, depending on whether they are scheduled by the operating system or not. Thread scheduling may be entirely under user control, whereas process scheduling is in general not under user control. Since the address space need not be changed, the cost of switching threads within a process is generally considered to be low.

An important thread-scheduling issue is whether a system call blocks only the calling thread or the entire process. (Of course it may block all threads on some system calls but not others.) This is a crucial issue for the interaction of message-passing libraries and thread libraries. A convenient programming technique is to fork a thread to perform a blocking receive while other threads continue to execute. For this to work, it is essential that the blocking receive operation not block the entire process. In addition, the thread must become dispatchable when the receive completes. Thus, useful thread libraries cannot be completely independent of the message-passing system, I/O system, or other parts of the operating system, such as some "user threads" packages are. Unfortunately, a thread library can be POSIX compliant without the important feature that system calls block only the calling thread, not all threads in the calling process. Therefore it is often difficult to tell from system documentation whether this feature is present or not, and experimentation is required.

Any comparison of processes with threads must take into account both the different memory models and the different scheduling mechanisms.

Processes are typically scheduled independently. *Gang scheduling* refers to scheduling sets of processes simultaneously. Gang scheduling is important for performance if synchronization among the processes occurs. Without it, message-passing latency or the delay required to obtain a lock may be "unnaturally" increased because the other process has been swapped out in order to execute another user's process. Threads belonging to a single process are in some sense automatically gang-scheduled when the number of threads is less than the number of processors, since the unit of scheduling by the operating system is normally the process.

2.3 Message Passing

By *message passing* we mean the transfer of data between processes (and in some cases, between threads) by `send/receive` operations, in which both processes must participate in the transfer. Thus we explicitly do not include "active" messages or remote-memory copy operations, which can be carried out by a single process. We will discuss these operations, which are likely to be provided in some form on SMP clusters of the near future, but we do not consider them (yet) to be part of the message-passing model. They are not part of the Message Passing Interface (MPI) standard, as defined by [3]. They may be included, however, in an extended version of MPI [4].

2.4 Shared Memory

By *shared memory* we mean the capability of multiple processes to access the same memory location. Under ordinary circumstances, for example, Unix processes have completely separate address spaces. However, many systems with shared-memory hardware (and some without) provide a mechanism whereby certain addresses are accessible to more than one process. A relatively cumbersome and incomplete mechanism has long been provided in the Unix context via System V shared-memory operations (`shmget`, `shmat`) or by `mmap`. Various vendors have also provided their own mechanisms (such as SGI's "shared arenas").

It is also possible to provide a single address space shared by all processes. An example is provided by the SGI `sproc` system call, which can "fork" a set of processes that share all memory. The normal Unix `fork` copies memory, which is then not shared[1]. Processes in this situation behave much like threads, since they share all memory. There is little agreement on standards for this model, with renders it not portable. By *virtual shared memory* we mean the provision of this model on distributed memory machines. A variety of techniques have been proposed for doing this efficiently.

In order to coordinate access to shared memory by multiple processes, some form of *lock*s must be provided. These also are typically non-portable. Locks can come in a variety of types (spinlocks, exponential back-off, etc.), and can be used to construct higher-level synchronization mechanisms, such as semaphores, barriers, and monitors. Little standardization has occurred on the form of these operations, although the `msemaphores` defined by OSF are available on more than one platform (at least KSR, Convex, and IBM). Locks may involve system calls (with the resulting expense) or may be achieved with low-level memory operations.

For some programmers, the "shared memory" model means that compiler directives control the parallelism available in certain loop, particularly if data locality has been controlled by other compiler directives. This is particularly attractive in Fortran, which has no explicit memory model containing the notion of address. Such compiler directives have only recently

[1] A variant is `vfork`, which is expected to copy only on use. In some Unix's, however, such as IBM's AIX, `vfork` is exactly the same as `fork`.

achieved any type of standardization, with data distribution directives being specified by HPF [5].

2.5 "One-sided" Message Passing

A characteristic feature of what we define here as the message-passing model is that data can only be moved from the local address space of one process to the local address space of another process through the cooperation of both processes. This can cause unnecessary synchronization. Some systems offer "one-sided" data transfer operations that offer some of the advantages of the shared-memory paradigm in a message-passing environment.

The first widely-used such system was the active message library on the CM-5, where messages could trigger actions, including node-to-node memory transfers, on remote nodes. A more recent version of this is the Cray T3D Shmem library, which provides `put/get` operations that can be used by a process on one node of the machine to transfer data to/from the memory of another node. In such schemes, a critical issue is how addresses on other nodes are represented, communicated, validated, and protected.

For examples, on the Cray T3D, physical addresses are communicated, and validation of the address for a `put` operation is done at the *sending* node. This is adequate for SPMD programs with uniform data layouts on all the nodes, but not for more sophisticated message-passing programs. On the Meiko CS-2, on the other hand, shadow page tables are maintained in the communication processor, so that virtual addresses can be communicated and addresses can be validated on the nodes where they are to be used.

3 Criteria

Here we describe the criteria that we will use to evaluate programming models that can be supported on clusters of SMP's.

The first one is *performance*. For some users of such machines, no programming model will be acceptable if it prevents them from obtaining a very large percentage of the maximum available performance. They are likely to evaluate the performance of applications in terms of floating point operations per second, with message-passing latency and bandwidth treated as known obstacles to obtaining peak megaflop rates. Since peak megaflop rates of today's RISC processors depend on having operands in cache, cache effects are an important part of the model.

The second major consideration is *portability*. There are at least three types of portability. Firstly, there are already "dusty deck", or "legacy", parallel programs. The ease with which these programs, often currently running on distributed-memory MPP's, can be moved to SMP clusters will be a factor in the market acceptance of the SMP's. Secondly, users increasingly have access to multiple computers of the same programming model, and will want their code to be portable from one SMP cluster to another. Standards such as MPI will be helpful in creating high-performance portable code. And finally, as software life cycles get longer and hardware life cycles get shorter, portability is necessary to be able to continue to develop and run the same application on multiple generations of the same manufacturer's products.

It is crucial that the models support *correctness* in that users cannot corrupt the memory of other processes. The Elan hardware in the Meiko CS-2, which maintains copies of page tables on the communications processor, is an example of a correctness-promoting design.

Ease of use relates to how cumbersome it is to write high-performance, portable, correct code at the level we are addressing here. Some may consider this irrelevant, assuming that "users" will interface to these systems through high-level parallel languages (e.g., Fortran M), parallel versions of existing languages (e.g., HPF), or "distributed objects". Nonetheless, someone has to provide these higher-level systems, and for them ease of use translates into more robust, less expensive, more maintainable user systems.

4 Components of the model

There are several components of any programming model for SMP's. These components are (almost) independent. We will consider all combinations of them.

4.1 Address space

The first part of the programming model to consider is whether or not the user sees a single address space (shared memory) or separate and distributed address spaces.

There is also an intermediate position: the address spaces are mostly distinct, but it is possible to have some space that is shared. This model is typified by the Unix System V shared memory operations. In this model, users can allocate shared memory, often with a special `malloc` (in C) or compiler directives (e.g., shared common for Fortran).

4.2 Process scheduling

The next part of the programming model concerns how the threads of control are handled. On each SMP, a user may have one or more processes, each of which may have one or more threads of control. There are a number of cases:

1. One process per thread of control, no more processes than processors.
2. One process on the SMP, with many threads of control (but no more than the number of processors)
3. Many processes (more than the number of processors), one thread per process
4. Many threads in each process
5. Combinations of these.

Each of these can be combined with the different memory models. Specifically, we do not require that processes have disjoint address spaces.

In this model, a key feature of a thread is the user-control over the scheduling of threads. This is the only major distinction between threads and processes that share an address space. (But recall the discussion in Section 2 on whether system calls block threads or processes.)

One issue that affects the choice of threads or processes is how they are scheduled. For many applications, there are three important choices. They are

- All processes/threads scheduled independently
- All processes/threads within a single SMP can be scheduled together (gang scheduling)
- All processes/threads can be scheduled together (grand gang scheduling)

In the discussion above, we distinguished between at most one user process per processor and more than one user process per processor without discussing how the number of processors is characterized. The obvious characterization that takes the number of processors may be misleading, since we are referring, particularly in the case of gang scheduling, to processors that the user has effectively exclusive access to. On an SMP running a full-featured operating system such as some variant of Unix, the operating system and its demons can consume enough CPU resources to cause performance problems for applications that use static load balancing (such demons include network demons, file system, accounting, message logging, etc). Thus, on an SMP with p processors, a user may want to use only $p-1$, leaving the last for the operating system. This is expensive for the user, of course.

4.3 Heterogeneity of model

The last component to consider is whether the user sees a uniform model (e.g., all shared memory or all message-passing) or not.

A uniform programming effectively hides some aspect of the cluster of SMPs. One model hides the shared memory of the SMPs by using message-passing everywhere. The other hides the interconnections between the clusters by using shared memory everywhere (that is, virtual shared memory between SMP's).

5 Commentary

There are eight basic options (2 choices along each of 3 axes). Two of these, non-shared address space threads with either model, are irrelevant and will not be discussed further.

The remaining six are shown in Table 1. In this section we examine each of the six possible models and discuss them with respect to the criteria described in Section 3.

Type	Address Space	Processes or Threads	Uniform Model
1.	shared	process	no
2.	non-shared	process	no
3.	shared	thread	no
4.	shared	process	yes
5.	non-shared	process	yes
6.	shared	thread	yes

Table 1: Possible Models

1. **Shared-memory processes, non-uniform programming model.** This model has the potential to provide high-performance on SMP clusters, due to explicit exploitation of shared memory on single SMP's. It is most likely to be realized in compiler-based parallelism for exploiting shared memory with explicit message passing between nodes (individual SMP's). Particularly in the case of large SMP's, explicit shared-memory programming, coupled with a small amount of message-passing, will deliver high-performance. Peak performance will depend on the hardware (and software!) that connects the nodes.

2. **Non-shared-memory processes, non-uniform.** This is not really an option, since if the processes don't share memory the model is uniform and occurs below.

3. **Shared-memory, threads, non-uniform model.** Compilers for systems that support threads are very likely to produce code that exploits multiple processors sharing all memory by compiling for threads rather than separate processes. The model will be non-uniform if the programmer adds explicit message-passing calls to his code. Also, programmers may write explicitly multi-threaded programs that use message passing. Note that this is only possible with a thread-safe message-passing library like MPI [3].

4. **Shared-memory, processes, uniform model.** This is the virtual shared memory programming model, in which all memory is shared, It is likely to be less efficient on SMP clusters (and on pure distributed-memory machines) than on SMP's, but useful in porting codes developed on single SMP's.

5. **Non-shared, process, uniform model.** This is the most *portable* of all the programming models considered here, because it consists of pure message passing, with the shared memory in each SMP used to provide particularly efficient message passing among processes on that node (See Section 5.2). One could expect MPI programs, for example, to port transparently from MPP's to SMP's and SMP clusters, with good performance.

6. **Shared-memory, uniform, with threads.** This is the same as the preceding, but with threads instead of processes.

5.1 A Note on Message Passing among Threads

For large SMP's, some have advocated porting message-passing codes to the SMP's by replacing processes by a single multithreaded process in order to gain the advantages of fast thread scheduling along with the sharing of memory that can make the message passing particularly efficient. This can be an effective approach, but one must be sure to understand the issues involved. The port is unlikely to be transparent, because of the very different memory models involved. In particular, when replacing processes with threads, variables that were by default (and by necessity) local to each process will now become shared by default. One can expect that changes will have to be made to a message-passing code for it to run with threads on an SMP, and that the changes will not port back to the distributed-memory machine, where threads sharing memory on different nodes is unlikely to be supported. It is also the case that threads are not the only way to obtain shared memory on most SMP's; there is always some kind of explicit shared memory available.

The issue is really scheduling. The main motivation for replacing processes by threads without replacing the message passing with explicit shared-memory operations is to get all threads scheduled on multiple processes simultaneously. If the operating system supports gang scheduling of processes (as is possible with SGI's `sproc`) then the need to address this model is eliminated.

5.2 Is message-passing slow?

One assumption that many researchers make is that message-passing is slow and that a shared memory model is faster. This view is based on the overhead of message-passing operations (i.e., matching message tags, synchronizing senders and receivers, and data buffer management). While there is some truth to this, the actual situation is more complex.

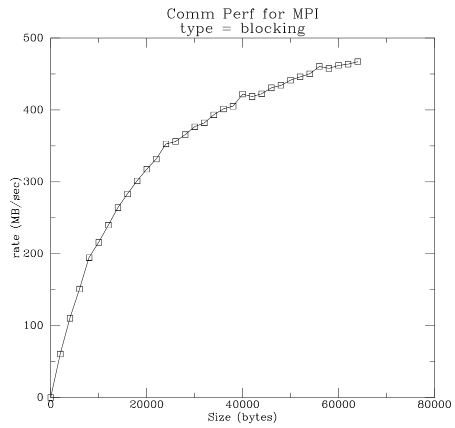

Figure 1: Performance for message-passing on shared memory hardware (preliminary figures)

To see why, first consider a shared memory program. Each access to shared data must be guarded, either by an explicit lock or some sort of barrier synchronization. Depending on the amount of hardware support for managing locks (or monitors or critical sections), this can be relatively expensive; in SMPs providing multiprocessing, even hardware support will probably need to be managed by the operating system, causing locks to require a system call (locks based

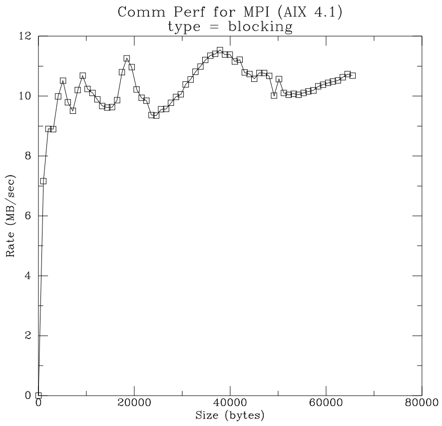

Figure 2: Performance for message-passing on shared memory hardware with two copies (preliminary figures)

purely on atomic reads and writes require many operations and are also not inexpensive).

Message passing requires many of the same operations. In one implementation, sending a message involves two locks and and a data copy, as well as some logic. This is not much higher than the locks required by shared memory.

Now consider the cost to access data. In shared memory, the data may be read directly. In message-passing, it must be copied from one buffer to another. This suggests that message-passing involves extra data-motion. But this need not be so. If there is one thread of control per processor, and each processor has a separate data cache, then even in the shared-memory case, the data will need to be copied into the cache of the processor that is referencing it. Thus, shared-memory may also need to copy the data, although this happens implicitly. The advantage over message-passing is thus primarily one of being able to access all of memory directly, without synchronizing with any other processor. But is this such an advantage?

Users must protect all memory reference operations (safe code must lock everything); locks are removed (implicitly) by assertions that data is not subject to multiple simultaneous writers or non-atomic read operations. This programming cost is often not taken into account.

Experiments on an SGI Power Challenge bear this out. The maximum obtainable bandwidth on this particularly system is 600 MB/sec; the asymptote is 586 MB/sec for the message-passing model (See Figure 1). Latency for short messages is close to locking cost. Note that these figures are for cache-to-cache copies.

Note that it is shared-memory hardware that makes this performance so good; our argument is that writing effective programs with shared memory is harder than it might seem, and in many situations any advantage over message passing is slight. Of course, if the programmer can guarantee that many operations can be performed while a single lock is held, then the cost of the locks can be amortized over many accesses.

On an IBM PowerPC-based SMP, preliminary data shows bandwidth given in Figure 2. In this implementation, shared memory is used for transferring messages, but with a "two-copy" mechanism, in which data is copied into shared memory from the sending process, and then from the shared memory into the receiving process's address space. Here we see lower bandwidth than the machine is capable of delivering, based on its underlying memory-to-memory copy speeds.

5.3 Latency Hiding

By latency-hiding we mean any programming strategy that allows useful work to be done while the process is waiting for an operation to complete. One approach to latency hiding is to have multiple threads per processor. For example, if a major component of latency in message passing is the cost of handshakes between source and destination, a second thread could run while the first thread is waiting for the handshake to occur. (This approach is often used for I/O and even for memory references, as on the Tera computer and the Alewife.) The downside is the additional cost of providing multi-threading (no thread may block another) to the single resource of the high-speed interconnect.

5.4 Obstacles to the Development of Performance Models

It would be advantageous to be able to analytically model the various programming models in order to determine their merits and costs. Unfortunately, an effective model is difficult to develop. Some of the complications include:

- the costs of switching threads (including locks, data structures, etc.)
- the effects on cache utilization (Do threads in same process have separate cache mappings? What about conflicts?)
- scheduling effects

6 Conclusions

The advent of SMP's and particularly clusters of them challenges existing programming models, and provides for a wealth of possibilities to be supported by vendors. Here we draw attention to three particular models that merit support from vendors, each for different reasons.

- The uniform, process-based, non-shared memory model is a crucial one because it offers the greatest *portability*. If an SMP and associated SMP cluster supports MPI, for example, then existing applications, programmed in MPI and running on MPP's, will port transparently.
- The non-uniform, threads-based model with shared-addresses local to a node but not local, is an important model because it offers greatest *performance* by exploiting the hardware fully. On each SMP, shared memory will be used to avoid message passing overheads and controlled by explicit POSIX-style thread primitives or a compiler, either generating message-passing calls internally or allowing the user to do so.
- The uniform shared-memory process model (virtual shared memory) offers the greatest *ease of use* in parallelizing codes that exist in serial form. compiler-based parallelism can often aid in getting such codes running quickly although not at peak performance. The transition to a high-performance version of the code can then proceed incrementally.

Other models are likely to be demanded by users, particularly those writing libraries that deliver parallelism in more abstract, object-oriented ways. But these three are essential.

Acknowledgements

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

[2] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994. (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[3] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[4] The MPI Forum. Extensions to the message-passing interface, 1996.

[5] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993.