

# Final Report: Programming Models for Shared Memory Clusters

*J. May, B. de Supinski, B. Pudliner, S. Taylor, S. Baden*

**January 4, 2000**

*U.S. Department of Energy*

Lawrence  
Livermore  
National  
Laboratory



## DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Work performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

This report has been reproduced  
directly from the best available copy.

Available to DOE and DOE contractors from the  
Office of Scientific and Technical Information  
P.O. Box 62, Oak Ridge, TN 37831  
Prices available from (423) 576-8401  
<http://apollo.osti.gov/bridge/>

Available to the public from the  
National Technical Information Service  
U.S. Department of Commerce  
5285 Port Royal Rd.,  
Springfield, VA 22161  
<http://www.ntis.gov/>

OR

Lawrence Livermore National Laboratory  
Technical Information Department's Digital Library  
<http://www.llnl.gov/tid/Library.html>



*Final Report*  
Programming Models for  
Shared Memory Clusters  
99-ERD-009\*

John May, *Principal investigator*  
Bronis de Supinski, *Coinvestigator*  
Brian Pudliner, *Coinvestigator*  
Scott Taylor, *Coinvestigator*  
Scott Baden, UCSD, *External Collaborator*

January 4, 2000

## 1 Introduction

Most large parallel computers now built use a hybrid architecture called a *shared memory cluster*. In this design, a computer consists of several nodes connected by an interconnection network. Each node contains a pool of memory and multiple processors that share direct access to it. Because shared memory clusters combine architectural features of shared memory computers and distributed memory computers, they support several different styles of parallel programming or *programming models*. (Further information on the design of these systems and their programming models appears in Section 2.) The purpose of this project was to investigate the programming models available on these systems and to answer three questions:

- How easy to use are the different programming models in real applications?
- How do the hardware and system software on different computers affect the performance of these programming models?
- What are the performance characteristics of different programming models for typical LLNL applications on various shared memory clusters?

---

\*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

The project was originally planned to run for two years, but it ended after its first year. Nevertheless, we were able to make significant progress toward answering all three questions. Many details of this work are described in two attached papers. Section 4 contains full citations and synopses of these papers.

The remainder of this report describes the background to our work, the results we obtained after the two papers were published, and the continuing influence of the project on other work after it ended on September 30, 1999.

## 2 Background

Developments in parallel computer architecture have brought about the need for research into shared memory clusters. We begin by describing these developments, and then we discuss techniques for programming these machines.

### 2.1 Parallel computer architectures

Until recently, parallel computer architectures fell into two categories: shared memory and message-passing.

Shared memory computers allow all the processors in the machine to read or store data directly at any location in memory. These machines are sometimes called *symmetric multiprocessors* or SMPs because every processor can access any memory location in an equal amount of time; no memory location is “farther away” from a given processor than any other location, so access to all data is symmetric.

In message-passing computers, the memory is distributed among the processing nodes, and each processor can directly access only its local portion of the memory. To access data in another node, a process must send or receive the data in a message over the computer’s internal network. While this network may be very fast, sending a message takes longer and requires more programming effort than accessing local memory.

Building very large SMPs is difficult because the memory system must efficiently handle simultaneous accesses from many processors. With current technology, SMPs can be built with up to 32 or 64 processors. In larger SMPs, contention among the processors for access to the common pool of memory would cause an unacceptable delay or *latency* between the time a request is issued and the time it completes. It is also difficult to build in sufficient *memory bandwidth* to handle the quantity of data being moved. Memory bandwidth is less of a problem in pure message-passing machines because they have only one processor per node, so there is essentially no contention.

A message-passing computer includes an interconnection network to transfer data between nodes. Such a network has fewer demands on it than the connection between processors and memory in an SMP because it carries less data and because the data can travel over independent paths. Also, programs can be designed to tolerate the latency on this network. As a result, message-passing machines are built with hundreds or thousands of processors. The ability to build

large machines using a given technology is known as *scalability*, and message-passing machines have long been considered more scalable than SMPs. For this reason, the term *massively parallel processor* or MPP has become synonymous with message-passing.

However, extremely large message-passing systems face two important problems. First, large interconnection networks are expensive. A second and more subtle problem is memory fragmentation: increasing the number of nodes in an MPP also increases the number of separate memory regions. The total amount of memory may grow in proportion to the number of processors, but the *fraction* of the total memory that each processor can access directly will diminish. The result of this *surface-to-volume ratio* problem is that as applications are scaled up to use more processors and memory, their performance may not increase proportionately because a larger share of their data accesses require message passing.

In many newer computers, vendors have combined message-passing with shared memory architectures to build shared memory clusters. The idea is to combine the scalability of message-passing with the good surface-to-volume ratio of SMPs. For a given amount of memory and number of processors, reducing the number of nodes allows each processor to access more data without passing messages. The development of shared memory clusters has coincided with the growing use of shared memory parallelism in mainstream computers. As a result, small SMPs have become an inexpensive building block for larger systems. Vendors developing or selling shared memory clusters include Compaq (formerly DEC), HP/Convex, IBM, SGI, and Sun.

In the IBM ASCI SST system at LLNL, each node has four processors that share access to local memory. Communication between nodes uses explicit message-passing calls. (The message-passing network is itself a multilevel system, but that doesn't directly affect the programming models we are studying here.) The SGI ASCI system at LANL uses a collection of Origin 2000 machines connected over a message passing network. A single Origin 2000 "box" can be programmed like a shared memory computer, and the LANL system uses message passing to communicate between the boxes. Each box has 128 processors organized as 64 two-processor nodes. Processors in the same node share memory in the usual way. Between nodes in the same box, processors communicate over an internal network that allows the 64 nodes to access each other's memory. While this avoids the need for message-passing calls within a box, the system requires additional hardware and software to translate local memory requests on a node into remote accesses on other nodes. The hardware must also ensure that data residing in each processor's cache memory remains consistent with main memory on remote nodes. Furthermore, unlike an SMP, this type of machine incurs different latencies for accesses to different memory locations. Because of this nonuniformity, the SGI Origin architecture is known as CC-NUMA [3] (*cache-coherent nonuniform memory access*.) A variant of CC-NUMA called S-COMA [5] (*simple cache-only memory architecture*) automatically moves data to the node where it is accessed most frequently. This technique can improve performance by reducing the frequency of off-node mem-

ory accesses, but it requires additional hardware and software, and it works poorly in some circumstances.

## 2.2 Programming models

Message-passing and shared memory architectures lend themselves to different parallel programming models. In message-passing machines, the tasks on each node do not share memory with each other, so they must exchange data by sending and receiving messages. Although various libraries exist for this purpose, most parallel programs at LLNL and elsewhere now use a library called MPI [6] (*message-passing interface*), which is widely available on parallel computers.

Shared memory systems typically use *threads* running on different processors to implement parallelism. Each thread executes an independent sequence of instructions within a program, but all the threads can access the same data. Conceptually, this model is simpler than message-passing, because threads can exchange data by reading and writing the same memory locations, rather than calling subroutines to pass messages. Furthermore, since all the data is equally accessible to all threads, a program can easily reallocate work among the processors to balance the computational load. In a message-passing system, on the other hand, reallocating work among the processors requires moving data between nodes.

Managing threads does require more skill than writing an ordinary sequential program, and the programmer must be careful to avoid synchronization problems, such as having one thread update series of a values that another thread is using. Some application developers have avoided explicit multithreading because it is tedious to program and because subroutines for manipulating threads are not entirely standard. Another option in shared memory systems is to use parallelizing compilers, which automatically distribute work (such as independent iterations of a loop) to different processors. Parallel language extensions such as HPF [2] and OpenMP [4] allow programmers to direct the compiler's attempts to parallelize the code. At LLNL, OpenMP has become increasingly popular as good compiler implementations have become available.

Unlike MPPs and SMPs, shared memory clusters do not have a single natural programming model. Programmers can use multithreading, message-passing, or a combination of both. Each approach has pros and cons. Pure shared memory techniques can be used on systems that implement CC-NUMA or S-COMA. Other shared memory clusters can simulate pure shared memory by using Distributed Shared Memory (DSM) software. With any of these methods, however, accesses to different memory locations can take different amounts of time. The variations are especially large in systems that use DSM because it is a software-only solution and because the interconnection networks on machines that use DSM software typically have longer latency than CC-NUMA or S-COMA networks. The problem with using pure shared memory programming on a shared memory cluster is that the programmer may not know which data accesses are local and which require communication with a remote node. Ignoring the distinction will not produce incorrect results, but it can reduce performance,

much as ignoring cache behavior can. Therefore, the simplicity of a pure shared memory programming model can hide performance pitfalls from the user.

A major advantage of message-passing is that it is highly portable. The broad availability of MPI lets developers write parallel programs that run almost unchanged on many different systems. Since it is relatively easy to implement message-passing on SMP architectures, programmers often use MPI on these systems to avoid making extensive changes to their message-passing codes, even though multithreaded code may be more efficient. The “MPI-everywhere” option is also viable on shared memory clusters, and it retains the advantage of offering maximum portability. Of course, this option foregoes some of the benefits of shared memory: efficient access to shared data and easier load balancing.

Finally, programmers can use a hybrid model, in which programs use shared memory techniques within a node and message-passing between nodes. The use of shared memory within a node exploits the system’s efficient access to shared data and allows a portion of the load balancing benefits available from a pure shared memory model. Passing messages makes it obvious to programmers when they are performing expensive remote data accesses. Using two kinds of parallelism within the same program therefore offers some of the benefits of both approaches, but it increases the program’s complexity.

Some researchers are trying to address the unique features of shared memory clusters by developing new programming models. These models allow applications to manage parallelism using a single set of subroutines, but they internally account for the differences in the time needed to access memory within and between nodes. One example of such a model is KeLP [1], developed by Baden and his students at the University of California, San Diego. This system allows an application to describe complex data structures and how they will be shared among tasks, and then it arranges to move data as necessary using the most efficient means available.

### 3 Project goals and activities

We initiated this project to determine more specifically the pros and cons of the different programming models on a variety of shared memory clusters. As noted in the introduction, we wanted to answer three main questions concerning the ease of use of the programming models, the performance effects of hardware and system software, and the high-level performance of applications using different models.

Our purpose in investigating these questions was to give users specific information to help them choose a programming model for their applications and to give computer center staff comparative data to help them evaluate current and proposed new architectures. Although several groups at LLNL had already begun experimenting with mixed programming models by the time we began this work in October 1998, there had been little systematic investigation of the different models.

This section describes the goals of our work and our activities during Fiscal

Year 1999. The next section and the papers that accompany this report describe the specific results of the project.

Five people (in addition to the many developers we spoke with) contributed to the project, although only three received direct funding from it: May, de Supinski, and Taylor. Each worked 50% time on the project, for a total of 1.5 FTE. Pudliner and Baden served as advisors and received no funding from the project.

We began our work by investigating the first main question of our project, concerning the programming models' ease of use. Since several LLNL code groups had already experimented with hybrid programming models, we conducted interviews with developers from these groups to learn what issues they faced as they migrated their codes to shared memory clusters. In particular, we wanted to know what programming models they used, how easy they were to program, and what performance they observed. This survey gave us some important insights into the ease of use of different models. We had planned to supplement this information with our own observations on the ease of use of different models in a later part of the project, when we adapted some sample codes to use different models.

Once we understood the current state of work on mixed programming models at LLNL, we turned to the second main question, how hardware and system software affect the low-level performance of different programming models. To answer this question, we measured a series of basic operations on different computers using a software benchmark program. This program is an extensively modified version of an existing performance analysis tool called SKaMPI, developed at the University of Karlsruhe in Germany. SKaMPI was written to measure the performance of MPI libraries. Our version of this code, called Sphinx, measures operations in Pthreads libraries as well as MPI. It is the first tool we know of that measures Pthreads performance. In response to requests from colleagues outside LLNL, we have released a version of this software to the public.

We completed measurements of Pthreads libraries on four different hardware platforms. We did not carry out similar measurements for OpenMP constructs before the project ended, but similar work is underway as part of a different project, and we expect to use Sphinx in that work.

To answer the final question, we planned to collect a small number of representative LLNL applications and investigate their performance using different programming models. This work was underway when the project ended.

## 4 Results

As noted earlier, we published two papers describing our work on this project.

The first, "Experience with mixed MPI/threaded programming models," by John May and Bronis de Supinski (UCRL-JC-133213), was presented as an invited paper at the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), which was held June 28–July

1, 1999. It presents the results of our survey of nine LLNL code projects that use mixed programming models. We found three combinations of techniques in use: Pthreads with MPI, OpenMP with MPI, and a threaded vendor library with MPI. Several groups had initially written code to use Pthreads with MPI but later changed to OpenMP with MPI, which they found easier to use. We found that some codes are well-suited to parallelization with OpenMP, while others are not. Those that work well with OpenMP have loops with fixed iteration bounds that are visible to the compiler. For these programs, versions compiled with OpenMP directives were about as efficient as versions using Pthreads calls. Some applications use a model in which small packets of work are created, placed on a queue, and then completed in an unpredictable order. For these codes, Pthreads offers a more natural programming model. The one code that used a threaded library with MPI achieved good threaded performance, but only portions of the code were able to use this library, so the code was not able to exploit the full parallelism of the machine for all of its run time.

A peripheral result of our survey was the creation of a Threads Working Group at LLNL. Many of the people we spoke with during our survey wanted better channels of communication with other developers, so we established a series of occasional informal talks and discussions on multithreaded programming.

The second paper, "Benchmarking Pthreads performance," by Bronis de Supinski and John May (UCRL-JC-133263), was a refereed paper also presented at PDPTA. It describes our techniques for measuring the performance of Pthreads basic operations and the design of the Sphinx tool. We found noticeable variations in the performance of certain operations across the tested platforms (DEC, IBM, SGI, and Sun) and for similar operations under different conditions. For example, synchronizing two threads through a "mutual exclusion lock" (mutex) took three times as long on the IBM as it did on the DEC when the two threads were forced to run on the same CPU. When we removed this constraint, the performance difference reversed and the IBM completed the synchronization in less than 1/15 the time that the DEC required. These results suggest that hardware and software differences on the two platforms strongly affect the performance of basic operations, and application developers need to consider these differences when they evaluate the performance of their codes.

After these papers were published, we focused on two areas: extending the capabilities of Sphinx and investigating the performance of representative LLNL codes using different programming models.

Our plan for Sphinx was to develop it into a general purpose tool for measuring and reporting a variety of performance data on MPI, Pthreads, and OpenMP primitives. We also planned to measure operation that used multiple programming models, such as barriers involving all threads on all processes in a job. SKaMPI, the code on which Sphinx is based, was developed to measure MPI performance only. It included a number of useful features for specifying and automatically performing a series of tests on basic MPI operations. The tool was designed to conduct its measurements in a structured way and to collect data in a form that is easy to manage. These features made SKaMPI a good

choice to be the foundation of our own work, but the range of measurements that SKaMPI could perform and the combinations of tests that could be specified were too limited for our purposes. In particular, we wanted to be able to carry out tests that varied more than one parameter and to have more flexibility in the kinds of parameters we could vary. We made some of the necessary modifications for our Pthreads tests, but we needed to make further changes before we could begin the OpenMP tests. These changes were nearly complete when the project ended.

To measure the performance differences between programming models in the same application, we planned to gather a small group of LLNL codes that used mixed models and test them using MPI only, MPI with Pthreads, and MPI with OpenMP. If time permitted, we planned to test versions of the codes using only OpenMP or Pthreads under DSM implementations and on CC-NUMA or S-COMA machines. We began by measuring a hydrodynamics code designed to work on block-structured meshes. The code had been parallelized using OpenMP, and we compared an MPI-only version to a version that used OpenMP with MPI on the unclassified ASCI Blue Pacific machine. Initial results showed that MPI-only version performed far worse than versions using OpenMP. Although we expected better performance from the versions using OpenMP, the magnitude of the result was puzzling, since it appeared that the OpenMP version was more than four times as fast as the MPI-only version for the same number of CPUs. We eventually determined that the OpenMP compiler we were using was generating inefficient code when we disabled (but did not remove) the OpenMP directives. When we repeated the tests using a different compiler, we found that the MPI-only version of this particular code was up to 15% faster than the version that also used OpenMP. These results apply only to the IBM compiler (and MPI library) running on the IBM system. Faster still was an MPI-only code that ran on just one CPU per node instead of all four. The additional improvement of about 5% compared to the other MPI-only version was due, we believe, to the lack of contention for memory bandwidth in the runs that used only one CPU per node. Of course, running a program using only one CPU per node wastes the other three CPUs in the node, but it also suggests that on this architecture, memory bandwidth limits performance when all four CPUs are in use (for this code). Unfortunately, we were not able to repeat these tests for different codes or different machines.

## 5 Ongoing benefits

After this project ended, several of its members began working on a related set of problems under the auspices of the ASCI program. This new work is focused on developing tools for understanding the performance of massively parallel applications. We have been able to continue (though at a slower pace) development of Sphinx, and we plan to carry out some low-level OpenMP performance measurements in the coming year.

We have also applied our experience measuring the performance of mixed-

model codes to the design of new performance analysis tools. Our work on the LDRD project has given us valuable insight into the questions that developers typically need answered when they investigate the performance of their applications.

Finally, we have been able to use our knowledge of various approaches to mixed model programming to advise other LLNL developers as they optimize their codes to run on hybrid architectures.

## References

- [1] S. J. FINK AND S. B. BADEN, *Runtime support for multi-tier programming of block-structured applications on SMP clusters*, in Proc. 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97), December 1997, pp. 1–8.
- [2] C. H. KOELBEL, D. B. LOVEMAN, R. S. SCHREIBER, G. L. STEELE, JR., AND M. E. ZOSEL, *The High Performance Fortran Handbook*, The MIT Press, 1994.
- [3] J. KUSKIN, D. OFELT, M. MEINRICH, J. HEINLEIN, R. SIMONI, K. GHARACHORLOO, J. CHAPIN, D. NAKAHIRA, J. BAXTER, M. HOROWITZ, A. GUPTA, M. ROSENBLUM, AND J. HENNESSY, *The Stanford FLASH multicomputer*, in Proc. 21st International Symposium on High Performance Computer Architecture, April 1994, pp. 302–313.
- [4] OPENMP ARCHITECTURE REVIEW BOARD, *OpenMP Fortran Application Program Interface*, October 1997.
- [5] A. SAULSBURY, T. WILKINSON, J. CARTER, AND A. LANDIN, *An argument for simple COMA*, in First IEEE Symposium on High Performance Computer Architecture, January 1995, pp. 276–285.
- [6] M. SNIR, S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA, *MPI: The Complete Reference*, The MIT Press, 1996.

