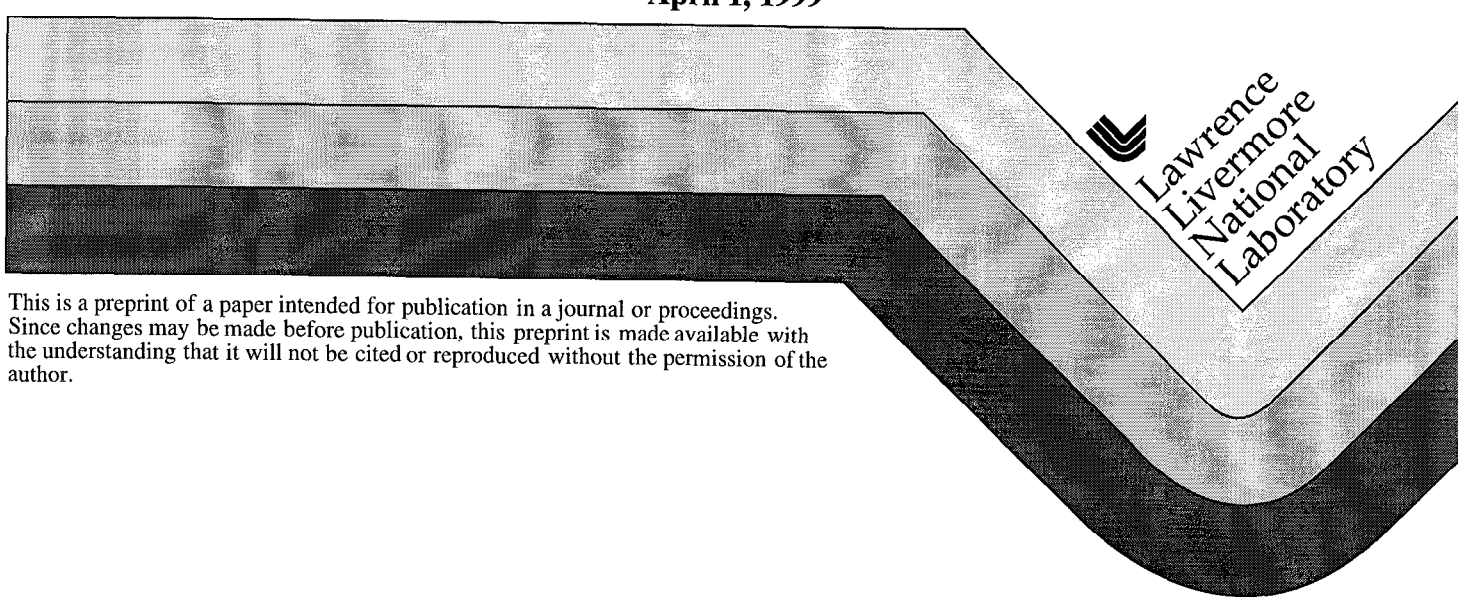
UCRL-JC-133213
PREPRINT

Experience with Mixed MPI/Threaded Programming Models

J. May
B.R. de Supinski

This paper was prepared for submittal to the
High Performance Scientific Computation with Applications
Las Vegas, NV
June 28-July 1, 1999

April 1, 1999



Experience with Mixed MPI/Threaded Programming Models*

John M. May
johnmay@llnl.gov

Bronis R. de Supinski
bronis@llnl.gov

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
7000 East Avenue, L-561
Livermore, CA 94550

1 Introduction

A shared memory cluster is a parallel computer that consists of multiple nodes connected through an interconnection network. Each node is a symmetric multiprocessor (SMP) unit in which multiple CPUs share uniform access to a pool of main memory. The SGI Origin 2000, Compaq (formerly DEC) AlphaServer Cluster, and recent IBM RS6000/SP systems are all variants of this architecture.

The SGI Origin 2000 has hardware that allows tasks running on any processor to access any main memory location in the system, so all the memory in the nodes forms a single shared address space. This is called a nonuniform memory access (NUMA) architecture because it gives programs a single shared address space, but the access time to different memory locations varies. In the IBM and Compaq systems, each node's memory forms a separate address space, and tasks communicate between nodes by passing messages or using other explicit mechanisms.

Many large parallel codes use standard MPI calls to exchange data between tasks in a parallel job, and this is a natural programming model for distributed memory architectures. On a shared memory architecture, message passing is unnecessary if the code is written to use multithreading: threads run in parallel on different processors, and they exchange data simply by reading and writing shared memory locations.

Shared memory clusters combine architectural elements of both distributed memory and shared memory systems, and they support both message passing and multithreaded programming models. Application developers are now trying to determine which programming model is best for these machines. This paper presents initial results of a study aimed at answering that question. We interviewed developers representing nine scientific code groups at Lawrence Livermore National Laboratory (LLNL). All of these groups are attempting to optimize their codes to run on shared memory clusters, specifically the IBM and DEC platforms at LLNL. This paper will focus on ease-of-use issues. We plan in a future paper to

*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. UCRL-JC-133213-ext-abs.

analyze the performance of various programming models.

Section 2 describes the common programming models available on shared memory clusters. In Section 3 we briefly describe the architectures of the IBM and DEC machines at LLNL. Section 4 describes the codes we surveyed and the parallel programming models they use. We conclude in Section 5 with a summary of the lessons we have learned so far about multilevel parallelism.

2 Parallel models

MPI [9] is widely used on distributed memory parallel computers, and it also works on shared memory machines. A simple way to run a parallel program on an SMP cluster is to treat each processor as if it were a separate compute node with its own private pool of memory. Processes exchange data by passing messages even when they could share memory directly. We call this the "MPI-everywhere" model.

For applications running on a shared memory system, programmers can choose among several kinds of multithreading. Pthreads [6] and other explicit threading models offer maximum flexibility, but they require programmers to create and synchronize threads explicitly. Although the Pthreads library is available on most parallel computers, minor differences between versions of the standard sometimes cause compatibility problems. Parallelization directives, like those in OpenMP [8], focus on loop-level parallelization and are easier to use than Pthreads. Easiest of all are parallel math libraries, which use threading internally to implement common operations like the Fourier transform.

In an SMP cluster, programmers can use MPI alone, or they can combine it with multithreading: the code within a node can use thread-based parallelism, and nodes can communicate with each other using MPI. We call this combination a mixed programming model. In NUMA machines, applications can use multithreading alone and avoid MPI entirely. There are other models for multithreading and for combining threads with message passing; however none of the codes we surveyed uses them, so we won't describe them here.

3 SMP clusters at LLNL

The codes described in this paper run mainly on two architectures at LLNL. One of these is the IBM Tech Refresh machine built for the Accelerated Strategic Computing Initiative (ASCI). At the time these codes were surveyed, the system included 168 four-way SMP nodes. The current system has twice as many nodes and is called the Combined Tech Refresh (CTR) system. IBM supports two forms of message passing in this system: User Space (US) communication offers high communication bandwidth and low latency, but at the time of our initial study only one MPI process could run on each node in User Space. Internet Protocol (IP) communication allows all the processors in the system to run separate MPI tasks, but the communication latency and bandwidth are poorer than they are in US mode. Thus, users had to choose between a simpler MPI-everywhere communication model with poor communication performance and a mixed MPI/threads model with better communication performance. New IBM system software eliminates this dilemma by allowing four MPI tasks per node to communicate in US mode. However the total number of MPI processes in a US mode job cannot exceed 1024. Therefore, the very largest jobs on the CTR system cannot run MPI everywhere in US mode.

The other kind of system is a cluster of AlphaServers. LLNL has several of these clusters. A typical configuration has eight nodes, and each node has eight to twelve Alpha pro-

cessors, for a total of 80 processors. The nodes are connected by Memory Channel, and MPI jobs can run over the entire system.

4 Mixed models

We surveyed nine mixed-model codes for this study. They all use MPI for message passing. Eight of the nine also use multithreading. The ninth is still experimenting with various techniques for threading. The threading techniques fall into three categories: loop parallelization directives, explicit threading, and multithreaded libraries. The following subsections describe how each technique was used.

4.1 Parallelization directives

Three of the codes we surveyed use directives. These are statements added to a program that have the format of comments but that some compilers can recognize as requests to parallelize the code in a certain way. Typically, directives appear just before loop statements, and if the compiler detects no dependencies between loop iterations, it will automatically generate code to parallelize the loop. Directives can also tell the compiler how to share data between threads and what portions of the loop should *not* be parallelized. Compiler vendors have offered directives for many years. Although the form of directives varies among compilers, their functionality is similar. Switching between directive types is relatively easy. Many parallelizing compilers are now adopting a standard set of directives for C, C++, and Fortran called OpenMP.

The Ares hydrodynamics code, written in C, is typical of how directives are used. It works on regular block-structured meshes, and it uses MPI message passing. The code was designed to use domain overloading, a technique that decomposes the mesh into more domains than there are processors. Each processor computes on several domains, one after another. The technique was developed to improve cache utilization and load balancing, but it also simplified thread parallelization. The code has computation phases separated by communication phases, and within each computation phase, an outer loop iterates over the domains assigned to each process. The first multithreaded implementation used Pthreads. Domains were assigned to threads within a process, and computation could proceed on separate domains in parallel. After each computation phase, a single thread would manage MPI communication to exchange data between the nodes.

Since the number of threads remained the same from one compute phase to the next, the developers initially implemented a thread pool, in which a set of threads was spawned in each task at the beginning of the program, and work was assigned to the same collection of threads throughout the execution of the program. The purpose of the thread pool was to avoid the cost of repeated thread creation and termination. However, managing this pool was complicated, and there was little performance benefit because the cost of creating a thread was small compared to the cost of computation. Overall, the Ares developers found the Pthreads model awkward. When compilers supporting parallelization directives became available, it was a simple matter to replace the Pthreads calls with a parallelized loop over the domains. Using directives greatly simplified the code, making it easier to maintain with no significant loss of performance.

Another C-language hydrodynamics code, this one with an unstructured mesh, also uses parallelization directives for domain loops. In addition, it uses directives to parallelize loops that compute the interactions between contact surfaces. These loops do not iterate over the domains but rather over large-grained subsets of the contact surface work. Several such loops were parallelized individually. The group developing this code noted the Ares experience

with Pthreads and moved directly from MPI-everywhere parallelism to MPI with directives.

A third directive-based code, written in Fortran 90, computes photon transport on an unstructured mesh. Like Ares, this code initially used Pthreads but later switched to directives. The computation can be decomposed for parallelism (singly or in combinations) over the three spatial dimensions of the transport medium, over energy groups, or over sweep angles. The spatial domain is distributed over nodes and uses MPI communication. Within each process, the code is parallelized on the outer loop, which iterates over sweep angles. This loop encapsulates more than 90% of the CPU time for the code. The computational work can be done in parallel, but for certain classes of problems the order of evaluation of individual angles is important—an incorrect ordering can lead to more iterations per code cycle before the convergence criteria are met. The code formerly distributed energy groups over MPI tasks, but it was restructured to place a loop over energy groups within the outer loop that computes over each angle. This greatly improved the cache performance, since the energy groups for each angle could be batched together. The developers of this code worked hard to eliminate dependencies between loop iterations. The code has only one Open MP directive placed on the angle loop, but restructuring the code to safely expose this parallelism required significant effort.

4.2 Explicit threading

Directives work well for codes with loop-level parallelism, but many threaded codes have a task-parallel structure that doesn't fit the directive model. These applications require the flexibility of explicit threading. Four codes we surveyed use explicit threading; of these, three use Pthreads and a fourth uses a custom thread library.

Ardra [3] is a C-language three-dimensional neutral particle transport code that uses a block structured mesh. The mesh is statically mapped to a 3-D processor layout. The compute time is divided nearly evenly between a harmonic projection phase that is easily parallelized and a more complicated sweep phase. A sweep phase is a sequential pass through the mesh, and the code makes these passes in several directions (e.g., 80 directions for an S_8 angular approximation). Boundary conditions may create dependencies between sweeps, but at least some sweeps can proceed in parallel.

To maximize parallelism, Ardra uses a data driven model for both message passing and multithreading. Each MPI process posts nonblocking receive requests. When data from another process arrives, it is added to a queue. When the process is ready for more work, it finds the next angular direction whose dependencies have all been satisfied and carries out that sweep. The program uses Pthreads within a process to replicate this data-driven model on a smaller scale. Each thread works autonomously, scanning an incoming work queue and carrying out computations. The Ardra group initially tried using a thread pool, but they removed it because it didn't improve performance significantly.

The sPPM [1] code simulates three-dimensional gas dynamics in a regular grid. It is written in C and Fortran. The grid is decomposed in three dimensions over MPI processes. Each thread updates a number of subdomains, which are managed on a work queue. One thread in each process exchanges large slabs of data with its nearest neighbors using nonblocking message passing. This communication is overlapped with the computation that all the threads are doing. By starting the exchanges as early as possible and waiting for their completion as late as possible, the code avoids most synchronization delays for large problem sizes.

sPPM was carefully optimized, and programming ease was secondary to performance. The code could potentially use compiler direc-

tives, but the developers believe that explicit threads give slightly better performance. They used low-level synchronization primitives in place of Pthread locks for the same reason. The code also uses a thread pool; unlike other codes groups, the sPPM project is willing to tolerate the extra complexity to reap any available performance gain.

The Semi-Coarsening Multigrid (SMG) [4] solver is a linear solver written in C. It initially used Pthreads and has since abandoned that model. In SMG, the problem domain is decomposed over MPI processes, and the threaded version of the code further decomposed each domain for a series of one- to three-level loops. Although the decomposition was static, the subdomains were assigned to threads dynamically as the threads completed preceding tasks. The code used a pool of threads that were spawned once and reused for each nest of loops. The performance of this model was disappointing, and the code was difficult to debug. The developers are currently experimenting with simpler threading models that they hope will also yield better performance.

The final project that uses explicit threading is Overture [2]. Overture is not a specific application but rather a C++ code framework designed for problems that involve adaptive mesh refinement, moving meshes, or overlapping grids used for complex geometries. The framework consists of many parts, and the Overture team has experimented with threads at several levels. The most effective use of threads has been for task-level parallelism. Multiple threads can independently work on separate patches of a grid. Overture uses the thread interface in the Tulip class library [10]. The Overture developers prefer Tulip for its object-oriented programming model and better performance than Pthreads. Tulip can encapsulate many different thread libraries, including Pthreads and system-specific thread libraries. The Overture team found that Tulip threads calling system-specific thread libraries were more efficient than Pthreads.

4.3 Threaded libraries

The only code we surveyed that uses a threaded math library is JEEP [5]. JEEP is a C and C++ molecular dynamics code. It computes molecular interactions at a quantum-mechanical level, solving the Schrödinger equation at each time step. JEEP combines message passing parallelism with calls to a threaded math library (ESSL [7]) on the IBM platform. The message passing portion of the code is parallelized over electronic states, with each process assigned one or more states. Computing each state requires several Fourier transforms, and ESSL uses multithreading to parallelize this part of the computation. Adding multithreading to a code in this way is very simple, but Amdahl's Law limits the available gains.

5 Results and future work

We have presented three general techniques for combining threaded programming with message passing. For codes that can use loop-level parallelization, directives are much easier to use than explicit threading. Only one surveyed code, which demanded the very highest performance, uses Pthreads in a situation where directives would also work. Although directives are easy to use, there were problems with some OpenMP compilers. Not all of them initially supported the full OpenMP standard, and some users had trouble determining the appropriate compile-time options.

The codes that use explicit threading have a task-oriented model of parallelism rather than a loop-oriented model. A task-oriented model is a poor match for directives because it builds a work queue incrementally, whereas directives generally require a complete description of the work upon entry to the loop. The sPPM code could be adapted to either model because the

tasks all take about the same amount of time and can be done in a predictable order. In Ardra, the programming model is clearly a task queue because the optimal order of execution is less predictable.

The code teams that experimented with thread pools have almost all been disappointed. The pools did not greatly improve performance, and only the sPPM project was willing to incur the extra complexity.

The time needed to adapt codes to use threads varied from a day or so to many months. The variation reflects several things: the time needed adapt the code to use threading (codes with loops that had independent iterations were easy to adapt); the threading model (directive-based codes were easier to adapt than Pthread codes, and JEEP was easiest of all); and the maturity of the tools (OpenMP compilers gave some teams problems).

We have focused here on ease-of-use rather than performance. Our initial information suggests little difference in performance between explicit threading and directives for the same code. We plan to examine performance differences between the different models in the near future.

Acknowledgements

We thank the code developers who shared their experiences with us and reviewed this paper for accuracy: Bruce Curtis, Noah Elliott, François Gygi, Ulf Hannebutte, Marty Marinak, Mike Nemanic, Tim Pierce, Brian Pudliner, Dan Quinlan, and Debbie Walker.

References

[1] *The ASCI sPPM benchmark code.* `http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/asci_sppm.html`.

[2] D. Brown, W. Henshaw, and D. Quinlan, *Overture: An object-oriented framework for solving partial differential equations*, in Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference, December 1997.

[3] P. N. Brown, B. Chang, M. R. Dorr, U. R. Hanebutte, and C. S. Woodward, *Performing three-dimensional neutral particle transport calculations on tera scale computers*, in High Performance Computing '99 (part of 1999 Advanced Simulation Technologies Conferece), San Diego, CA, April 1999.

[4] P. N. Brown, R. D. Falgout, and J. E. Jones, *Semicoarsening multigrid on distributed memory machines.* To appear in the *SIAM Journal on Scientific Computing special issue on the* Fifth Copper Mountain Conference on Iterative Methods. Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-130720.

[5] F. Gygi, *The JEEP manual*, Lawrence Livermore National Laboratory, Livermore, CA, August 1998. email: gygi1@llnl.gov.

[6] Institute of Electrical and Electronics Engineers, *IEEE standard for information technology : portable operating system interface (POSIX)—Part 1: System Application Program Interface (API)*, New York, 1996.

[7] International Business Machines Corportation, *Parallel ESSL Version 2 Release 1.1 Guide and Reference*, Poughkeepsie, New York, November 1998. `http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/`.

[8] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, October 1997. http://www.openmp.org/.

[9] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, MPI Press, Cambridge, Mass., 1996.

[10] *Tulip: A portable parallel run-time class library.* http://www.acl.lanl.gov/tulip/.