PTHREADS *vs* MPI PARALLEL PERFORMANCE OF ANGULAR-DOMAIN DECOMPOSED S_n METHODS ON SMP ARCHITECTURES

Y. Y. Azmy
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6363
yya@ornl.gov

D. A. Barnett
Lockheed Martin Corporation
P. O. Box 1072
Schenectady, NY 12301

ABSTRACT

Two programming models for parallelizing the Angular Domain Decomposition (ADD) of the discrete ordinates (S_n) approximation of the neutron transport equation are examined. These are the shared memory model based on the POSIX threads (Pthreads) standard, and the message passing model based on the Message Passing Interface (MPI) standard. These standard libraries are available on most multiprocessor platforms thus making the resulting parallel codes widely portable. The question is: on a fixed platform, and for a particular code solving a given test problem, which of the two programming models delivers better parallel performance? Such comparison is possible on Symmetric Multi-Processors (SMP) architectures in which several CPUs physically share a common memory, and in addition are capable of emulating message passing functionality. Implementation of the two-dimensional, S_n, Arbitrarily High Order Transport (AHOT) code for solving neutron transport problems using these two parallelization models is described. Measured parallel performance of each model on the COMPAQ AlphaServer 8400 and the SGI Origin 2000 platforms is described, and comparison of the observed speedup for the two programming models is reported. For the case presented in this paper it appears that the MPI implementation scales better than the Pthreads implementation on both platforms.

1. INTRODUCTION

The Symmetric Multi-Processors (SMP) architecture has been gaining popularity, both on its own and as the building block of larger multiprocessor computers. An SMP is comprised of a collection of powerful CPUs with equal access to a common, or shared, memory in addition to a hierarchy of cache(s) local to each processor. As such, it supports the shared-memory programming model in which interprocessor communication is accomplished via storage in common areas of memory. A standard, portable library of routines designed to facilitate the shared-memory programming model, *Pthreads*, that allows a process to start other light-weight processes, or threads, is available on most SMPs. At the same time, the Message Passing Interface (MPI) standard that supports the message-passing, or distributed-memory, programming model is also available on some SMPs. While an optimal implementation of MPI will not communicate between processors by physically passing messages on SMPs, it will provide additional portability to this architecture by simulating such a communication mode using the shared memory. The additional flexibility afforded to the programmer in terms of selecting the programming model presents an opportunity for achieving better parallel performance. In this work we compare the parallel performance of each programming model using the AHOT code. AHOT solves the two-dimensional discrete ordinates (S_n) equations using the Arbitrarily High Order Transport method of the Nodal type. Parallel AHOT is tested on both a COMPAQ AlphaServer 8400 and an SGI Origin 2000.

In Sec. 2 we briefly review the Angular Domain Decomposition (ADD) that we employ to parallelize the mesh sweep, which is the most time consuming component of the S_n method. We describe the implementation of the parallelized algorithm using both the shared memory and the message passing programming models in Secs. 3, and 4, respectively. We report measured parallel performance on the COMPAQ AlphaServer 8400 and SGI Origin 2000 as a function of the number of participating processes for both models, and compare the scaling of the speedup factors.

2. ANGULAR DOMAIN DECOMPOSITION

The most general discrete ordinates calculation that is based on the first order form of the transport equation can be decomposed into a sequence of mesh sweeps along a single angular direction at a time. Due to the large number of times this procedure is repeated in a typical calculation (once per quadrature angle, per iteration) it has become the focus of efforts to parallelize S_n methods. Among the three primary domain decompositions that have been attempted for S_n methods: energy, angle, and space,[1] Angular Domain Decomposition (ADD) possesses the following desirable features:

1. Intrinsic domain decomposition in non-curvilinear geometry: The angular fluxes are coupled

primarily through the scattering/multiplication source which is fixed during a mesh sweep. (Additional coupling can result from reflective or periodic boundary conditions; at a minimum, the directions within an octant in angular space are mutually independent within a mesh sweep. Moreover, if all boundary conditions are fixed-value, including vacuum, then all angular directions are mutually independent.) Hence, identical arithmetic operations are performed (perhaps not in the same order) in the parallel algorithm as in the sequential mesh sweep, yielding identical (within arithmetic precision) intermediate and final results. This implies that the number of iterations required to achieve convergence is independent of the number of subdomains, or participating processes. This is important for achieving high parallel speedup, relative to the serial code.

2. Perfect load balance: All mesh sweeps comprise about the same computational load, and thus consume the same amount of time to execute even on irregular grids. This results in a perfect load balance among the processes participating in the calculation (assuming the number of processes divides the number of discrete ordinates that are mutually independent) thereby eliminating process idleness. Furthermore, this enables static scheduling of angles to participating processes, which is more efficient than dynamic scheduling in the absence of process idleness.

3. Medium parallel grain size: The amount of useful computation performed between two synchronization points defines the parallel grain size. ADD's granularity lies somewhere between that of energy (coarse) and spatial (fine) domain decompositions. Compared to energy and spatial domain decomposition, ADD incurs a relatively smaller parallelization penalty, but produces relatively fewer independent processes that can be executed simultaneously. The limited parallelization penalty enhances parallel efficiency, a desirable effect. The limited number of independent processes limits the potential for speedup, and is an undesirable constraint. This forces a compromise between the two effects that is guided by the target architecture (e.g. shared memory), the class of applications (e.g. typical problem size), and additional performance measures beyond the hardware utilization (e.g. turnaround time).

Parallel algorithms based on ADD have been developed, implemented and analyzed on a wide variety of multiprocessor platforms encompassing shared and distributed memory architectures.[1] Two examples are the production neutral particle transport code TORT[2] and the Arbitrarily High Order Transport (AHOT) test code.[3] TORT has been available for many years on CRAY UNICOS platforms and is capable of execution in multitasking mode, which is a shared memory environment. Its performance, measured and modeled, on a variety of CRAY models, e.g. Y/MP, C90, and J90, has been reported previously.[4] While significant speedup was achieved, the lack of portability of the multitasked code, and the limited potential for scaling to a large number of processes led us to explore other alternatives as summarized below.

Recently, TORT was converted to run under the POSIX threads, *Pthreads,* standard, a portable library of routines for SMP architectures. This version was installed and tested on several platforms including an 8-CPU COMPAQ AlphaServer 8400, a 2-CPU Sun Sparc Ultra-2, a

32-CPU SGI Origin 2000, and a 2-CPU Ppro PC running the GNU/Linux operating system. The main purpose of this exercise was to illustrate the portability of the resulting code. In addition, performance measurements on a few test problems exhibited reasonable speedup but at rapidly deteriorating parallel efficiency as the number of processes increased. Perhaps more impressive than the parallel performance of the Pthreaded-TORT is its almost perfect portability; only minor modifications were necessary to install and execute the code on the several platforms listed above. This is important because the parallel inefficiency observed on a given system may be a consequence of the implementation of *Pthreads*, not a property of the parallel algorithm in general.

The complex production nature of TORT, and particularly its memory management functionality, made it difficult to examine the scaling potential of ADD because it was difficult to distinguish between the causes of loss of efficiency that were genuine to the parallel algorithm, and those that were a consequence of its implementation in TORT. For this reason we shifted our attention to the two-dimensional test code AHOT which is far less sophisticated in its capabilities, but much cleaner in its programming, thus more faithful to the solution algorithm in its performance. In the remainder of this paper we focus on the parallel implementation of AHOT based on ADD using two programming models via fully portable standard libraries, Pthreads and MPI.

3. SHARED MEMORY PROGRAMMING MODEL

The main advantage of the shared memory programming model is the simplicity of programming in it, especially if the programmer starts from a serial code that is to be parallelized. Basically, the programmer needs to identify the data that must be shared among the participating processes and place them in common locations in memory. Libraries that implement multiprocessing on shared memory platforms possess instructions to synchronize access to these shared data. Data that are not shared by the participating processes are stored in private areas of the memory, and typically hold intermediate variable values, loop indices, etc. The similarity between this programming model and standard modular programming on traditional serial computers is evident: variables (scalars or arrays) declared in common blocks, or accessed by address location are shared between processes, otherwise variables are private.

Synchronization of the participating processes on shared memory architectures is accomplished by a variety of mechanisms. The two that were employed in earlier parallelizations of transport calculations are:

1. Mutual exclusion (mutex) locks that permit execution of a section of code enclosed in the lock by only one process at a time.

2. Barriers that suspend execution at a specific point until all participating processes reach that point.

In addition to simplifying the parallelization process, shared memory architectures provide an inexpensive means to implement dynamic scheduling. This mode of assigning concurrent processes to an active processor can reduce the adverse effects of load imbalance in time sharing environments. This is achieved by listing the identifying index of all concurrent processes in a queue stored in the common area of memory. Each active process, under lock, grabs the next index from the queue and increments the queue position by one for the next process to grab; it then conducts the mesh sweep for the angle corresponding to the index it grabbed.

The parallelization penalty on shared memory architectures results from the overhead of starting the participating processes, initializing the synchronization mechanisms, and the spin time. The first two can be reduced by the programmer to an insignificant level in large computations by performing these activities only once at the outset of execution, and avoiding their repetition. The spin time refers to the amount of time the CPUs spend doing no useful computations while waiting at a synchronization point, thus spin time depends on run time conditions that are beyond the control of the programmer. Furthermore, most operating systems implement sophisticated algorithms that adjust the length of spin time according to the dominant computational load history at a given time. Another common reason for loss of parallel efficiency on shared memory architectures is memory contention which is typically hard to diagnose, and sometimes impossible to eliminate due to safety features of the operating system.

To enable the comparison with the MPI version described in the following section, a static scheduling policy is implemented in the Pthreads version of AHOT. Two domain decompositions are applied: the mesh sweep is decomposed along the angular variables, ADD, while the computation of the iteration residual is conducted along a spatial domain decomposition. The angles are assigned to threads with a stride equal to the number of threads requested by the user at run time. The spatial domain decomposition is implemented in blocks of rows and are assigned to threads with a stride computed from the number of blocks of rows available in the problem. Since the mesh sweep dominates the computational time this implementation is termed ADD in spite of the fact that it is a hybrid domain decomposition.

The code is organized such that the creation of the requested threads and initialization of the mutex lock and four barriers are completed outside the inner iterations module. For each inner iteration the loop over angles that implements the mesh sweeps is executed by each thread for angles statically assigned to it as described above. The angular flux spatial moments computed in the process of a mesh sweep are accumulated by each thread in a private array in order to avoid synchronization points that would result in finer granularity and a higher potential for loss of parallel efficiency. At the conclusion of a mesh sweep along all angles in a thread's domain, the thread accumulates, in a shared array, the contribution to the new iterate of the scalar flux spatial moments. The accumulation is performed under a serializing lock. This is followed by a barrier point in order to ensure that the convergence test stage does not commence until all contributions to the new iterate have been accumulated.

The convergence test stage starts with each thread computing the largest pointwise iteration residual in its assigned block of rows and then storing this value and its location index in private locations. Once all threads have completed this activity, they synchronize at the second barrier, then under lock they compute the largest magnitude iteration residual and save it in a shared location. This is followed by the third barrier to guarantee that no thread will proceed before a final determination of iterative convergence is reached. At this point all threads have access to the global iteration residual. If the user specified convergence criterion is satisfied, then all threads exit the inner iterations module and join the parent thread which successfully terminates the computation. If convergence has not been achieved but a new iteration is to be started, the threads update in parallel the shared arrays of the old and new iterates of the scalar flux spatial moments, synchronize at the fourth barrier, and then proceed to perform the next iteration. If convergence is not achieved and the user specified number of iterations has been exhausted, the threads exit the inner iterations module and join the parent thread which then terminates the execution and warns the user of the lack of convergence.

The algorithm as summarized above was implemented in AHOT using the standard Pthreads instructions, and was successfully installed and executed on an 8-CPU COMPAQ AlphaServer 8400, a 2-CPU Sun Sparc Ultra-2, and a 32-CPU SGI Origin 2000. Sample measured parallel performance is reported in Sec. 5.

4. MESSAGE PASSING PROGRAMMING MODEL

The programming model for message passing architectures is complicated by the fact that it has no immediate analogy in traditional serial programming. In this case the CPUs comprising the target platform possess (physically or logically) uncoupled memory spaces and data exchanges among participating processors must be accomplished via explicit message exchange. Another difficulty of the message passing programming model arises from the way messages must be exchanged. At specific points in the instruction flow in different processes the programmer must anticipate the need for data exchange, and the program must coordinate both sending and receiving processes to conduct this exchange. This requires complete definition of the message contents from type and length to location in memory on both processes involved. Among the various types of message exchange methods available, blocking exchanges are the safest because they protect the contents of the message from being overwritten by subsequent instructions until they have been copied elsewhere, either in the send buffer or sent to the receiving process. For this reason, message exchange is the primary method of synchronization on message passing platforms.

The main advantage of message passing architectures is the potential for better parallel scaling to a larger number of processes. Also, in the message passing programming model the programmer has better control on the conduct of the computation, via explicit decisions to communicate, by virtue of the lower possibility of interference by the operating system. The primary source of parallelization penalty here is the cost of message exchange which is typically characterized by a latency component to initialize a message and a volume component characterizing the communication bandwidth. Hence, it is crucial in message passing programming to reduce the number of messages and the volume of traffic to the absolute minimum, and to overlap computation with communication as much as possible. Earlier studies that attempt to evaluate latency and data traffic volume as a function of platform characteristics for transport methods have identified algorithms, e.g. Bucket Algorithm, that perform better on computers with particularly low latency.

In the context of the AHOT code three communication strategies aimed at improving the performance of global reduce operations and reducing the memory requirement on physically distributed memory architectures were examined, and their performance measured and modeled.^{5} That study implemented the parallel instructions using the PVM (Parallel Virtual Machine) library. In the present work, the three communication strategies investigated earlier are briefly reviewed below. In the interest of cross-platform portability, they were implemented using the standard MPI library.

The parallelization of the AHOT code on message passing architectures is also based primarily on ADD. Thus the master process reads the input data which specifies the problem configuration and selects various options, constructs arrays representing the fixed source, angular quadrature, material composition and cross section data, etc., and broadcasts all necessary input to the slave processes. Each process, master or slave, proceeds with mesh sweeps along the angles statically assigned to it, and accumulates the contribution to the scalar flux spatial moments in a local array. At the conclusion of all mesh sweeps all participating processes accumulate their local arrays, which contain partial contributions to the scalar flux spatial moments, into the new iterate array. The array accumulation is based on global reduce operations that are implemented differently in each of the three considered strategies.

In the first strategy, termed MPI, the native MPI global reduce operation is used. This of course can vary from one platform to another, but often it is based on a spanning tree connection topology among the participating processes. In such a scheme, half the participating processes send their partial contributions to the other half, that performs the sum, then half the remaining processes send again, and so on. This is followed by a broadcast stage in which the new iterate is sent to all participating processes along the same spanning tree topology described above. The main advantage of this scheme is that it reduces the number of messages exchanged to a minimum, a crucial benefit on platforms that possess high communication latency. On the other hand, its disadvantages include a substantial idleness as the number of active processes is cut by half in each step of each of the two stages, and the additional idleness that results if the number of processes is not a power of 2.

The second of the three communication strategies is based on the Bucket Algorithm which performs the global reduce operation on a monodirectional ring topology. In this scheme, each of the P participating processes starts by passing a *bucket* containing the P-th subvector of its local contribution to the scalar flux spatial moments to its neighbor along the monodirectional ring. Each process then sums the subvector it just received into the corresponding local subvector which contains its own contribution, then sends the bucket to its neighbor, and so on. By the time each bucket has circumnavigated the entire ring, each process will contain the completely updated iterate for its subvector. In the broadcast stage, each process sends the new iterate subvector to its neighbor again along the monodirectional ring. The bucket algorithm typically results in a larger number of messages than the spanning tree scheme, however, it produces a smaller volume of data traffic that is not concurrent, hence outperforming the native global reduce on platforms with small communication latency. The main advantage of the Bucket Algorithm is that it does not constrain the number of participating processes to powers of 2, thereby all but eliminating idleness if the number of processes divides the number of angles. Its disadvantages include a greater number of arithmetic operations, and of messages exchanged. In addition, it requires implementation by the programmer since it is not available in standard MPI libraries. Clearly this can result in suboptimal performance since it is programmed at a relatively high level and, for the purposes of cross-platform portability, will not take into account individual features of the target platforms that impact parallel performance.

The third communication strategy is very similar to the Bucket Algorithm except it attempts to simulate a production environment by distributing the large flux iterates arrays and the fixed source array over the memories of the participating processes. In the implementation of this scheme in AHOT, termed Distributed Memory, these arrays are blocked by rows in the y-direction. This arrangement implies a significantly larger communication penalty as each sweep of a block of rows must start with the *owner* process broadcasting the fixed scattering plus fixed source subarray in its local memory to all participating processes. The advantage here is the great reduction in the required memory since little duplication of these large arrays is necessary. Communication in this scheme is conducted along a bi-directional ring, and some overlapping of computation and communication is introduced for the sake of higher parallel efficiency. Since messages collide in a bi-directional ring comprised of only two processors, it is necessary to execute this algorithm on three or more CPUs.

In each of the above communication strategies, the convergence test section of the code was conducted in a way compatible with that strategy. For the MPI scheme the native global maximum function was used to compute and broadcast the largest magnitude iteration residual, while for the Bucket Algorithm a spatial domain decomposition was implemented as described in the shared memory implementation. In the Distributed Memory scheme, the native global maximum function was employed. In all cases the small, almost negligible cost, of the convergence test section made the choice of parallel scheme immaterial.

The AHOT code implementing these three communication strategies was translated to employ MPI for communication, and was successfully installed and tested on an 8-CPU COMPAQ AlphaServer 8400, a 32-CPU SGI Origin 2000, and a 24-CPU IBM SP2. In this case, also, the MPI code was almost perfectly portable to these different platforms. Measured parallel performance for the three strategies outlined above is presented in Sec. 5 and compared to the performance of the Pthreads version of AHOT.

5. NUMERICAL RESULTS AND CONCLUSIONS

The Pthreads and three MPI versions of AHOT were used to solve a simple test problem on a COMPAQ AlphaServer 8400 and an SGI Origin 2000. The test problem is a one group, two region configuration based on a 16 x 16 uniform mesh with all vacuum boundary conditions and an S_{16} angular quadrature. For the purpose of this test we used a third order spatial expansion of the dependent variables in Legendre polynomials, and required all computed spatial moments to converge to a relative criterion of 10^{-4} in all runs, which resulted in convergence in 27 unaccelerated inner iterations. Correctness of the parallel code was verified by comparing the converged scalar flux with the values from the serial run. Agreement was verified to within the convergence criterion.

The measured elapsed time (sec) for the runs with the Pthreads version and the three message passing schemes versions of AHOT *vs* the number of participating processes is shown in Table I for the COMPAQ AlphaServer 8400 and in Table II for the SGI Origin 2000 platforms. Note that in the Pthreads implementation the number of requested threads must evenly divide the number of angles in the quadrature. (The S_{16} quadrature contains 36 angles per octant.) Also, the Distributed Memory version cannot be executed with fewer than 3 processes, and results are not available for 5 or 7 CPUs because in these cases the blocking of rows results in one process that owns an empty block.

Table I. Performance (Elapsed sec) of Parallel AHOT on COMPAQ AlphaServer 8400

	Serial	1	2	3	4	5	6	7	8
Pthreads	40.8	45.6	27.3	20.0	15.9		10.5		
MPI		44.5	22.5	15.2	11.9	10.7	8.1	8.1	7.1
Bucket		46.1	24.9	15.5	12.8	10.6	8.1	8.2	7.2
Dist Memory				16.6	12.0		8.0		7.7

In general these results exhibit reasonable speedup as the number of participating processes increases. The AlphaServer displays slightly better scaling, but the difference is probably not significant given the vagaries of system load on time sharing computers. Comparison of the results shown in Tables I and II illustrates the far better performance of message passing versions over the Pthreads version for the two platforms considered. This is not surprising since earlier experience suggests that, even though it is harder to develop a message passing version from a sequential code, message passing typically possesses higher parallel efficiency. This is largely due to the ease of identifying the sources of parallel inefficiency, which enhances the programmer's ability to optimize performance on a target platform. Also, the measured results demonstrate insensitivity of the parallel performance to the MPI communication strategy; each of the message passing implementations show approximately equal scaling. This is a hopeful sign for production codes where distributing the memory requirement across participating processes might be necessary. Of course it still remains to be verified that this observation, namely the insensitivity of the parallel performance to the MPI communication strategy, holds when CPUs outside of a single box of an SMP are employed.

Table II. Performance (Elapsed sec) of Parallel AHOT on SGI Origin 2000

	Serial	1	2	3	4	5	6	7	8
Pthreads	74.2	74.5	74.1	37.6	32.8		27.8		
MPI		80.8	39.7	29.4	21.6	19.5	16.9	15.9	13.1
Bucket		81.2	41.1	28.1	22.4	19.6	15.8	15.1	13.5
Dist Memory				28.1	21.8		16.2		15.7

REFERENCES

1. Y. Y. Azmy, "Multiprocessing for Neutron Diffusion and Deterministic Transport Methods," *Progress in Nuclear Energy* **31**, 317 (1997).

2. W. A. Rhoades and D. B. Simpson, "The TORT Three-Dimensional Discrete Ordinates Neutron/Photon Transport Code," *ORNL/TM-13221* (1997).

3. Y. Y. Azmy, "Performance and Performance Modeling of a Parallel Algorithm for Solving the Neutron Transport Equation," *J. Supercomputing* **6**, 211 (1992).

4. Allen Barnett and Yousry Y. Azmy, "Parallel Performance of TORT on the CRAY J90: Model and Measurement," *International Conference on Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Sept. 27-30, 1999, Madrid, Spain, Vol. 1, p. 422, American Nuclear Society, La Grange Park, IL (1999).

5. Y. Y. Azmy, "Communication Strategies for Angular Domain Decomposition of Transport Calculations on Message Passing Multiprocessors," *Joint International Conference on Mathematical Methods and Supercomputing for Nuclear Applications*, Oct. 5-9, 1997, Saratoga Springs, NY, Vol. 1, p. 404 (1997).