Application based Evaluation of Distributed Shared Memory Versus Message Passing*

Sumit Roy, Vipin Chaudary and Shi Jia
Parallel and Distributed Computing Laboratory
Wayne Sate University
Detroit, MI 48202

Padmanabhan Menon
Enterprise Systems Group
Hewlett-Packard Company
Cupertino, CA 95014

Abstract

Clusters of Symmetrical Multiprocessors (SMPs) have recently become very popular as low cost, high performance computing solutions. While some programs can be automatically parallelized for use on a single SMP node, using multiple nodes in a cluster requires the programmer to rewrite the sequential code and employ explicit message passing. This paper explores an alternate approach, the use of a multithreaded Distributed Shared Memory (DSM) system, *Strings*. Though shared memory programs are easier to write, a DSM system may not perform as well as a message passing library. The performance of both approaches is evaluated using two applications from the field of medical computing. The first application is a program for deblurring images obtained from Magnetic Resonance Imaging. The other program is part of a system for radiation treatment planning. Each program was initially parallelized using a message passing approach. The programs were then rewritten to use a multithreaded approach over the DSM. Our results show that current implementations of the standard message passing libraries PVM and MPI are not able to effectively exploit multiple processors on an SMP node. Since *Strings* is multithreaded, it provides very good speed-up for both the programs in this environment. It is also seen that the DSM code is as good as the message passing version for one program, and nearly as good in the other program.

1 Introduction

Though current microprocessors are getting faster at a very rapid rate, there are still some very large and complex problems that can only be solved by using multiple cooperating processors. These problems include the so-called *Grand Challenge Problems*, such as Fuel combustion, Ocean modeling, Image understanding, and Rational drug design. Recently many vendors of traditional workstations have adopted a design strategy wherein multiple state-of-the-art microprocessors are used to build high performance shared-memory parallel workstations. These symmetrical multiprocessors (SMPs) are then connected through high speed networks or switches to form a scalable computing cluster.

Using multiple nodes on such SMP clusters requires the programmer to either write explicit message passing programs, using libraries like MPI or PVM; or to rewrite the code using a new language with parallel constructs eg. HPF and Fortran 90. Message passing programs are cumbersome to write and may have to be tuned for each individual architecture to get the best possible performance. Parallel languages work well with code that has regular data access patterns. In both cases the programmer has to be intimately familiar with the application program as well as the target architecture. The shared memory model on the other hand, is easier to program since the programmer does not have to worry about sending data explicitly from one process to another. Hence, an alternate approach to using compute clusters is to provide an illusion of logically shared memory over physically distributed memory, known as a Distributed Shared Memory (DSM) or Shared Virtual Memory (SVM).

This paper evaluates the performance of two large applications which are parallelized for execution on a cluster of symmetrical multiprocessors. The first program (MRI) is used for deblurring of images obtained from magnetic resonance imaging at the Department of Radiology, Wayne State University. The other application (RTTP) forms part of the system routinely used for radiation therapy treatment planning for patients at the Gershenson Radiation Oncology Center at Harper Hospital, Detroit. The parallel versions of the code were written using the standard message passing libraries PVM and MPI, and also implemented with a shared memory model over a DSM, *Strings*. Our results show that implementations of the *Strings* DSM ef-

*This research was supported in part by NSF grants MIP-9309489, EIA-9729828, US Army Contract DAEA 32-93D004 and Ford Motor Company grants 96-136R and 96-628R

fectively exploits multiple processors on SMP nodes, since it is a multithreaded runtime. Good speed-up and scalability in this environment are obtained in this environment. The DSM version of the MRI program performs as good as, and sometimes better than the message passing code, depending on the algorithm that is implemented. Though the RTTP program is more challenging, the *Strings* code comes within 12 – 20 % of the performance of the original MPI program.

The next section provides some background about the different runtime environments used in this evaluation. The two applications are introduced in Section 3. The performance results are shown and discussed in Section 4. Section 5 concludes this paper.

2 Programming Environments

The programming environments used in this work include two standard message passing libraries, PVM and MPI, and a multithreaded Distributed Shared Memory system, *Strings* [1].

2.1 Parallel Virtual Machine

The Parallel Virtual Machine (PVM) [2] is a runtime system composed of two parts. The first part is a daemon that resides on all the computers making up the virtual machine. When a user wishes to run a PVM application, a virtual machine is created first by starting up the PVM daemon on each host in the physical distributed machine. The PVM application can then be started from any of the hosts.

The second part of the system is a library of PVM interface routines. It contains a fairly complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains routines for passing messages, spawning processes, coordinating tasks, and modifying the virtual machine.

2.2 Message Passing Interface

The Message Passing Interface (MPI) [3] is another industry standard message passing library that includes point-to-point communication and collective operations, all scoped to a user-specified group of processes. MPI allows the construction of virtual topologies for graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient, efficient way. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

MPI also provides services for environmental inquiry, basic timing information for application performance measurement, and a profiling interface for external performance monitoring. The performance measurements in this paper were done with the MPICH implementation from Argonne National Laboratory [4].

2.3 *Strings*: Distributed Shared Memory

Strings [1] is a fully multithreaded DSM that incorporates Posix1.c threads multiplexed on kernel lightweight processes. The kernel can schedule multiple threads across all the processors on an SMP node, using these lightweight processes. *Strings* is implemented as a library that is linked with the application programs. At initialization of the runtime, a separate `communication` thread is created. This thread handles all incoming messages by issuing a blocking `recvmsg(3N)` on a *UDP* socket. Hence, messages do not generate explicit interrupts that would have to be serviced. Requests from other nodes for synchronization objects or shared memory pages are handled by short lived threads. Thus multiple independent requests can be served in parallel.

Portable global pointers are implemented across nodes in the DSM program by mapping the shared regions to fixed addresses in the memory space. The system uses Release Consistency with an update protocol, derived from *Quarks* [5]. Changes to shared pages are only propagated to other nodes when there is a synchronization release ie. unlock or a barrier arrival. The changes are obtained by *diffing* the current version of the page with the contents after the previous release operation. A distributed queue is implemented for locks, requests from the local node preempt remote requests. While this policy is not fair, this optimization works well for the programs tested.

3 Application Details

The two applications evaluated are deblurring of images obtained from magnetic resonance imaging and genetic optimization of radiation treatment planning.

3.1 Deblurring of MRI Images

Magnetic resonance imaging (MRI) is a relatively new technique for obtaining images of the human body. It offers many advantages over conventional techniques such as X-ray and γ-ray imaging, since it does not use radiation and has no known side-effects. MRI images are obtained by placing the sample to be imaged in a very strong, uniform magnetic field.

Images generated by MRI may suffer a loss of clarity due to inhomogeneities in the magnetic field. One of the algorithms proposed for deblurring uses a map of the magnetic field to correct the image for local inhomogeneities [6]. First, the local field map data is acquired and a set of frequency points is determined for demodulation and image reconstruction. The collected data is corrected for nonlinear sampling density and then gridded onto a Cartesian scale [7, 8]. Next, a two dimensional Fast Fourier Transform (FFT) is used to get the image in Cartesian space. This process is carried out for each of the selected frequency points. This is a very computation intensive operation and appreciable gains could be provided through parallelization. A set of images is produced, each deblurred for a particular frequency component of the local magnetic field. The final image is obtained by a consolidation of these images. For each pixel, the value of the local frequency map corresponding to its Cartesian coordinates is taken and the two closest selected frequencies are determined. Then a linear interpolation of the pixel values of the images from these frequencies is done to get the pixel value of the final image.

The parallel implementation was done using two approaches, message passing using PVM and MPI, and a Distributed Shared Memory (DSM) version using *Strings*. Three different parallel algorithms were evaluated.

3.1.1 One-to-one Task Mapping

The program is divided into three main phases. In the distribution phase, the master program reads in the input data and calculates which frequencies are to be demodulated. After the number and value of frequency points to be demodulated has been determined, a corresponding number of slave processes is spawned and the relevant data is distributed to the slaves. This is the distribution phase of the program. At the end of this phase, each slave program possesses its own copy of the data on which it performs computations.

Each of the slave programs then proceeds to do the demodulation, gridding, and Fourier transform for its chosen frequency. This phase of the program is executed in parallel and it is where the bulk of the computation is carried out. At the end of the computation, the local results are sent back to the master which then consolidates all the data by calculating the best possible value for each pixel in the final image. This was the initial algorithm described in [9], and was implemented using PVM and MPI.

3.1.2 Workpile Model

The DSM implementation of the algorithm used the workpile model of computation. A master thread starts first, allocates memory for shared variables and initializes data. It then reads the input data and determines the frequencies that should be demodulated. Child threads are then created to perform computation on the data. These threads obtain one frequency to be deblurred from a shared memory region and carry out the computation on that frequency. When the process is done, the child updates the final image with the data from this frequency. It then takes another frequency, until the workpile is empty. Note that this kind of computational model would be difficult to implement using explicit message passing, since one would have to fork a special handler process to serve incoming requests from the child processes.

3.1.3 Static Task Distribution

The third implementation statically divides the number of frequency points that need to be demodulated across the total number of tasks that are created. In this approach, the MPI and PVM versions can use less tasks to handle a larger number of points. Compared to the original DSM version, the slave threads do not have to obtain new tasks from the master and this save on communication. If a task handles more than one frequency, the results are initially accumulated in a local copy of the image. These partial results are then passed to the master process before the child task terminates. This reduces the amount of communication compared to the One-to-one task mapping and the Workpile models.

3.2 Radiation Therapy Treatment Planning

Radiation therapy using external photon beams is an integral part of the treatment of the majority of cancerous tumors. In this modality, beams of photons are directed at the tumor in the patient from different directions, thereby concentrating radiation dose in the tumor. The maximum dose that can be delivered to a tumor is limited by its neighboring normal organs' tolerance to radiation damage. Conventionally, desired dose distributions within the patient are calculated by dosimetrists/physicists using computers and represent attempts to maximize the dose to the tumor while minimizing the dose to normal tissues. Treatment plans acceptable to the radiation oncologist are obtained by iterative interaction between the dosimetrist and the oncologist. This treatment planning process is fairly time consuming for three-dimensional treatment planning.

Computerized treatment planning was limited for many years to producing dose distribution on one or a few planes of a patient's body. In addition, the beams that could be modeled were limited to those coplanar with the patient planes. These limitations were primarily caused by the computing resources generally available and the need to produce evaluable plans in time frames relevant to clinical practice, which is to say in minutes, not hours or days. A detailed description of the genetic algorithm used in this implementation is available in [10]. The purpose of the code is to select a set of beams, each with a corresponding weight, used to treat the patient. A beam's weight is its relative intensity. This set of beams and weights collectively is called a plan. Each beam is a description of a particular radiation field, its orientation in space, energy, shape, wedge, etc. Each beam will produce a particular dose distribution within a patient, and this is characterized by calculating the dose per unit beam weight at many points within the body. A plan is the total dose distribution obtained by summing, for each point being considered, the dose from each beam in proportion to its weight. The quality of the plan is judged by evaluating this dose distribution in relation to the particular constraints imposed by the physician. The point of the optimization is to select the best beams and weights.

3.2.1 Parallel Implementation of RTTP

The genetic optimization algorithm has a dependency from each generation to the former generation, except for the first random generation. Hence, the main loop cannot be parallelized. However, each function call within the main loop has many simple uniform loops which can be executed in parallel. The parallel algorithm has a fork-join implementation and a large number of synchronization points when compared to the MRI application. The master program reads in the data and creates an initial population of beams. This population data is distributed across the slaves. Each slave task determines the dose distribution in the patient's body due to the beams in its part of the population. The master collects the results, and uses genetic optimization to determine a new population. This process is repeated until the change between two iterations is below some threshold value. In an actual clinical environment the same procedure would be repeated multiple times so that the oncologist can select a suitable plan from the ones generated.

4 Performance Analysis

The performance analysis of both applications was done on a cluster of four SUN UltraEnterprise Servers. One machine is a six processor UltraEnterprise 4000 with 1.5 Gbyte memory. The master process in each case was started on this machine. The other machines are four processor UltraEnterprise 3000s, with 0.5 Gbyte memory each. All machines use 250 MHz UltraSparcII processors, with 4 Mb external cache. Two networks are used to interconnect the machines in the cluster, 155 Mbps ATM with a ForeRunnerLE 155 ATM switch, and 100 Mbps FastEthernet with a BayStack FastEthernet Hub. The results presented in this paper are obtained on the FastEthernet network. Based on previous work, we would not expect the ATM network to provide a very large performance improvement [11].

4.1 Deblurring of MRI Images

The performance evaluation of this code was done using a 256x256 pixels test image. First, the One-to-one task mapping implementations is compared against the serial code. Then the workpile model versions of the DSM is compared against the One-to-one task mapping. Finally we show results for the Static Load Allocation approach using MPI, PVM, and *Strings*.

4.1.1 One-to-one Task Mapping

Figure 1 shows the performance comparison between the serial code and the MPI as well as PVM implementations. This test was conducted using two of the four processor SMP machines.

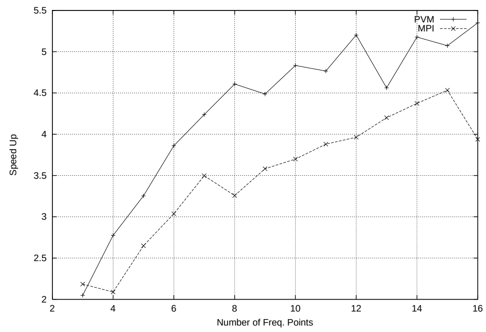

Figure 1: Speed-Up with up to 8 Processors

From the figure, very good initial speedups are observed for both PVM and MPI versions but the PVM version does consistently better than MPI. The code was further analyzed and it was found that the main difference was in the efficiency of collective communication. There are two broadcasts/multicasts in the master program. The second broadcast consists of *3.667 MBytes*.

The MPI version takes a very large amount of time for the broadcast as compared to PVM. The reason is that PVM uses a multi-level broadcast system. The master *pvmd* daemon sends the broadcast message to all the daemons running on the other hosts in the virtual machine. Each daemon then forwards the message locally to all the tasks running on its host. In MPI, on the other hand, there is no daemon process and the broadcast is implemented as a tree structured multiple unicast. Hence, the broadcast consumes more time and bandwidth, leading to an overall performance degradation.

It is also seen that the performance degrades when the number of tasks is more than the number of processors.

4.1.2 Workpile Model

The One-to-one model using message passing was then compared to the Workpile model on *Strings*, using 16 CPUs. .

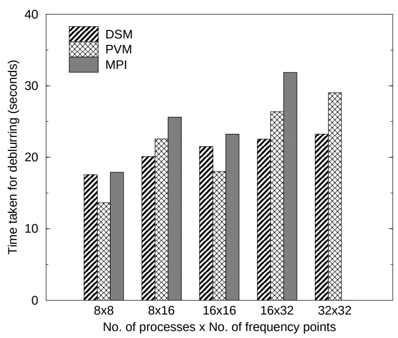

Figure 2: MRI on *Strings*, PVM and MPI (16 CPUs)

Figure 2 shows that the DSM version of the code outperforms the MPI version for all the configurations of problem and machine size that were used. The MPI code could not be used for the problem size of 32 frequency points being deblurred by 32 processes, because the MPI master program was being contacted by too many slave processes leading to a hot spot in communications. This caused a communication timeout and a consequent shutdown of the MPI environment.

The PVM program also performs better than MPI for all program and network sizes. This is due to the extra overhead incurred by MPI during the broadcast phase, as explained earlier. The comparison between PVM and *Strings* shows that for small problem sizes, PVM performs better than *Strings* but as the problem size is increased, ie. for a greater number of frequency points chosen, *Strings* performance is better. Therefore, *Strings* exhibits better scalability than PVM.

4.1.3 Static Load Allocation

Figure 3 shows the speed-ups obtained using 48 frequency points and the Static Load Allocation algorithm. Compared to Figure 1, the message passing codes scale much better, since the number of processes has been limited to the physical number of CPUs. The *Strings* version of the code performs marginally better, since at the end of the computation, it only sends data that was changed in the code. The message passing programs on the other hand send the entire local copy of the partially deblurred image.

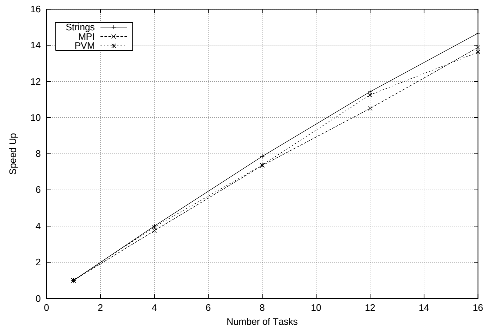

Figure 3: MRI on *Strings*, PVM and MPI (Static Load Allocation)

4.2 Genetic Optimization

The genetic optimization program was tested using the MPI implementation and the *Strings* version. Two beam sets are generated using a population of 100. Figure 4 shows the speed-up obtained using the two approaches. It can be seen that the *Strings* version of the program performs almost as well as the MPI code.

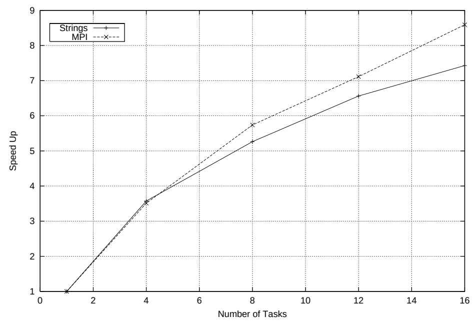

Figure 4: Speed-Up for Genetic Optimization

In the MPI version of the program, there is a reduction operation at the end of the parallel part of the loop when the child tasks inform the parent process about the resultant beam dose due to their part of the population. At the beginning of each iteration, the parent broadcasts the new population obtained by genetic optimization of the current generation.

In the shared memory version, these updates are achieved by using barrier synchronization. Since *Strings* uses release consistency, this is the only way that updates to globally shared memory can be made visible to other tasks. The barrier synchronization requires more messages than the broadcast, since all the processes have to contact the barrier manager when they arrive at the barrier, and the manager in turn has to send out the *barrier crossed* message to each participating node. Hence the *Strings* version performs a little worse than the MPI implementation.

5 Conclusion

In this paper the performance of two message passing systems, PVM and MPI was compared to a Distributed Shared Memory, *Strings*, using two large applications. The performance figures obtained show that the *Strings* can provide nearly equivalent performance compared to the message passing platform, depending on the algorithm selected. This is particularly true for a compute cluster consisting of multiprocessors nodes, since the multithreaded design of the *Strings* system provides a light-weight approach to exploiting all the processors on each node.

Related research compared the performance of TreadMarks, an early generation DSM, with PVM using mostly benchmark codes and computational kernels [12]. The DSM performed worse than PVM due to separation of synchronization and data transfer, and additional messages due to the invalidate protocol and false sharing. In contrast, in this paper we have shown performance results with actual programs used in medical computing. *Strings* uses a multithreaded approach to allow efficient use of an update protocol, and reduces the effect of false-sharing by allowing multiple writers to a page.

In conclusion, this paper shows that a Distributed Shared Memory system can provide performance close to that of message passing code, while allowing a simpler and more intuitive programming model. Our future work includes trying to reduce the barrier synchronization overhead further, and to attempt to exploit even more parallelism in the RTTP application.

References

[1] S. Roy and V. Chaudhary, "*Strings:* A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, (Chicago, IL), pp. 90 –97, July 1998.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. 55 Hayward Street, Cambridge, MA 02142: MIT Press, 1994.

[3] "Message passing interface forum." http://www.mpi-forum.org/.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, pp. 789–828, Sept. 1996.

[5] D. Khandekar, *Quarks: Portable Distributed Shared Memory on Unix*. Computer Systems Laboratory, University of Utah, beta ed., 1995.

[6] D. C. Noll, C. H. Meyer, J. M. Pauly, D. G. Nishimura, and A. Macovski, "A Homogeneity Correction Method for Magnetic Resonance Imaging with Time-Varying Gradients," *IEEE Transactions on Medical Imaging*, vol. 10, pp. 629 – 637, 1991.

[7] J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a Convolution Function for Fourier Inversion Using Gridding," *IEEE Transactions on Medical Imaging*, vol. 10, pp. 473 – 478, 1991.

[8] C. H. Meyer, B. S. Hu, D. G. Nishimura, and A. Macovski, "Fast Spiral Coronary Artery Imaging," *Magnetic Resonance in Medicine*, vol. 28, pp. 202 – 213, 1992.

[9] P. Menon, V. Chaudhary, and J. G. Pipe, "Parallel Algorithms for deblurring MR images," in *Proceedings of ISCA 13th International Conference on Computers and Their Applications*, March 1998.

[10] V. Chaudhary, C. Z. Xu, S. Jia, G. A. Ezzell, and C. Kota, "Parallelization of Radiation Therapy Treatment Planning (RTTP): A Case Study," in *Proceedings of ISCA 12th Intl. Conference on Parallel and Distributed Computing Systems*, August 1999.

[11] S. Roy and V. Chaudhary, "Evaluation of Cluster Interconnects for a Distributed Shared Memory," in *Proceedings of the 1999 IEEE Intl. Performance, Computing, and Communications Conference*, pp. 1 – 7, February 1999.

[12] H. Lu, "Message Passing Versus Distributed Shared Memory on Networks of Workstations," Master's thesis, Rice University, 1995.