J78716

Performance Evaluation of Remote Memory Access (RMA) Programming on Shared Memory Parallel Computers

Haoqiang Jin and Gabriele Jost*

NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000

Abstract

The purpose of this study is to evaluate the feasibility of remote memory access (RMA) programming on shared memory parallel computers. We discuss different RMA based implementations of selected CFD application benchmark kernels and compare them to corresponding message passing based codes For the message-passing implementation we use MPI point-to-point and global communication routines. For the RMA based approach we consider two different libraries supporting this programming model. One is a shared memory parallelization library (SMPlib) developed at NASA Ames, the other is the MPI-2 extensions to the MPI Standard. We give timing comparisons for the different implementation strategies and discuss the performance.

1. Introduction

In this study we will compare different programming paradigms for the parallelization of large scientific applications on shared memory computer architectures. The applications we consider are such that they can be divided into sub-problems so that many processes can work together on different parts of the same data structure.

Parallel programming on a shared memory machine can take advantage of the globally shared address space. Compilers for shared memory architectures usually support multi-threaded execution of a program. Loop level parallelism can be exploited by using compiler directives such as those defined in the OpenMP standard [5]. Lightweight threads are automatically created for performing the work in parallel. Data transfer between threads is done by direct memory references. This approach provides a relatively easy way to develop parallel programs but has disadvantages. It is difficult to achieve scalability for a large number of processors and it is not portable to distributed memory architectures.

The programming models considered in this study assume that each process has its own local memory. The message passing programming model is a well understood programming paradigm for this situation. The computational work and the associated data are distributed between a number of processes. If a process needs to access data located in the memory of another process,

* Computer Sciences Corporation, M/S T27A-2, NASA Ames Research Center.

it has to be communicated via the exchange of messages. The data transfer requires cooperative operations to be performed by each process, that is, every send must have a matching receive. The regular message passing communication achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver. The Remote Memory Access (RMA) programming model is also based on the concept of processes with their own local memory, but it separates the communication and synchronization step. A process is allowed to directly read from or write to data areas located on other processes, without the exchange of messages. Data transfer between two processes is performed by only one side and does not require a matching operation by the other process. The correct ordering of memory accesses has to be imposed by the user through explicit synchronization.

Both programming models are applicable on distributed as well as shared memory computer architectures. Message passing on a shared memory machine may be implemented as memory-to-memory, however, libraries supporting this paradigm, such as the MPI 1.1 standard [3], often impose a high latency. The RMA functionality allows implementations to directly take advantage of fast communication mechanisms provided by the hardware platform, such as coherent shared memory, hardware supported put and get operations or communication co-processors.

In this study we evaluate their effect on performance for programming shared memory architectures. We first discuss different RMA programming paradigms in Section 2, present benchmark implementations with RMA in Section 3, compare the performance results in Section 4, and conclude in the last section.

2. Library Support for Different Parallel Programming Paradigms

To study the impact on performance of the message passing vs. RMA parallel programming paradigm, we chose two libraries supporting these programming models.

2.1. MPI and MPI-2

MPI (Message Passing Interface) [3] is a widely accepted standard for writing message passing programs. It is a standard programming interface for the construction of a portable, parallel application in Fortran or in C, especially when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree). MPI provides the user with a programming model where processes communicate by calling library routines to send and receive messages to other processes. Pairs of processes can perform point-to-point communication to exchange messages. A group of processes can call collective communication routines to implement global operations such as broadcasting values or calculating global sums. Global synchronization can be implemented by calls to barrier routines. Asynchronous communication is supported by providing calls for probing and waiting for certain messages. For all communication operations the sending as well as the receiving side have to issue calls to the message-passing library.

MPI-2 [4] is an extension to the MPI standard. MPI-2 provides one-sided communication routines to support the RMA programming model. These routines extend the communication mechanism of MPI by allowing one process to specify all communications parameters. RMA is initiated with a collective library call where each process specifies an area of memory that is made accessible to remote processes. This shared memory buffer is used for the exchange of data. A call to a one-sided communication routine needs to be issued only by one process and does not require a matching call by sender or receiver respectively. MPI-2 provides point-to-point and barrier synchronization operations and it is the user's responsibility to ensure memory coherence. The MPI-2 extensions that we used in our study are:

- `MPI_Win_create`: A collective routine for setting up a shared memory buffer.
- `MPI_Get`, `MPI_Put`: Routines for transferring data to and from a shared memory buffer.
- `MPI_Win_fence`: A routine for performing collective synchronization.

MPI-2 extensions also include routines for point-to-point synchronization, however, they were not available on the hardware platforms that we used for our study.

The SGI Origin offers the SHMEM library which provides similar functionality as the MPI-2 extensions for one-sided communication. Since this library is only available on SGI systems we chose the MPI-2 extensions for our study to have more potential for portability to other systems.

2.2. MLP and SMPlib

MLP is a methodology of programming developed by Taft [8] at NASA Ames Research Center for achieving high levels of parallel efficiency on shared memory machines. It exploits two-level parallelism in applications: coarse-gained (domain decomposition) with forked processes and fine-grained (loop level) with OpenMP threads. Communication between MLP processors is done by directly accessing data in a shared memory buffer, and as a result MLP has very high bandwidth and low latency. Coupled with the second level parallelism MLP has demonstrated scalability on more than 500 processors for real CFD problems [8].

The shared-memory parallel programming model in MLP is summarized in Figure 1. A program starts with a single process, the *master* process, to perform initialization, such as reading input data from a file, and set up necessary shared memory buffers (or arena) for communication. Additional processes are then created via the *fork* call. The forked processes have a private copy of the virtual memory of the master process except for the shared memory arena. Thus, broadcasting any input data is not necessary in this model as it would have been required in a message passing program. The master and its forked processes then work on the designated code segments in parallel and synchronize as needed.

The MLP library (MLPlib) consists of only three routines: `MLP_getmem` to get a piece of shared memory, `MLP_forkit` to spawn processes and `MLP_barrier` to synchronize processes. The simplicity of MLPlib makes programming with MLP relatively easier, even though a user still needs to perform the tedious task of domain decomposition. The main limitation of MLPlib is its lack of point-to-point synchronization primitives, which are usually required for more general class of applications.

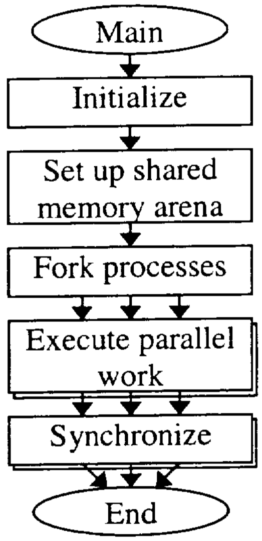

Figure 1: The shared-memory parallel programming model with forked processes.

We have extended the MLP concept to overcome some of its limitations and developed the SMP library (SMPlib). SMPlib includes the `SMP_Signal` and `SMP_Wait` primitives for point-to-point synchronization between processors. A processor may update a shared buffer and use `SMP_Signal` to inform another processor the availability of the data; the other processor can use `SMP_Wait` for the notification of the signal to copy data from the shared buffer. The Signal/Wait approach is very flexible and in general has less communication overhead than a global barrier. In the meantime SMPlib still maintains a simple programming interface like MLP and can easily be applied to more general applications.

In the current study, we focus on the effectiveness of the first level parallelism with SMPlib, that is, the fine-grained loop-level parallelism with OpenMP is not considered.

SMPlib supports RMA programming but employs a somewhat different programming paradigm from MPI-2. The properties of the different programming models are summarized in Figure 2

	MPI	MPI-2	SMPlib
Data Model	private	private	mix of private and shared
Domain Decomposition	required	required	required
Communication	explicit call to communication routines by sender AND receiver	explicit call to communication routine by sender OR receiver	direct access of shared memory buffer without library calls
Communication Latency (SGI Origin)	~5 microseconds	~0.5 microseconds	~0.5 microseconds
Synchronization	implicit in message exchange	explicit library calls	explicit library calls
Portability	shared and distributed architectures	shared and distributed architectures	shared memory architectures

Figure 2: Properties of the different programming paradigms.

3. Benchmark Implementations

We used the NAS Parallel Benchmarks (NPBs) [1] for our RMA study. The NPB suite consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. The five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. We chose a subset of the NPB consisting of the three application benchmarks (BT, SP and LU) for our study.

3.1. Porting Message Passing to RMA

As a basis for our evaluation we started with the MPI implementation NPB2.3 [2] of the benchmarks, which we ported to the RMA programming model. We adopted the domain decomposition strategy of these implementations which we will explain in more detail below. Porting from message passing to RMA consists of three major steps. In the RMA initialization phase a shared memory buffer has to be allocated. This buffer will be used to hold data that needs to be accessed by remote processes. The second step consists of replacing the calls to the message passing routines by read and write operations from and to the shared memory buffer. At last necessary calls to synchronization routines have to be inserted.

There are two approaches to synchronization. A collective call to a barrier routine will cause all processes to wait until the last process has reached the barrier. Another approach is point-to-point synchronization where a process waits for one particular named process until it receives a signal.

In all of our implementations each process logically owns a specific part of the shared memory buffer. We distinguish between two methods to update the values in the shared memory buffer. A process can place values to be communicated in its own segment of the buffer. The remote process requiring the data will read it from there. We refer to this approach as the GET method. Alternatively a process can write data directly into a remote processes segment of the shared memory buffer. We refer to this approach as the PUT method. Figure 2 illustrates the two methods.

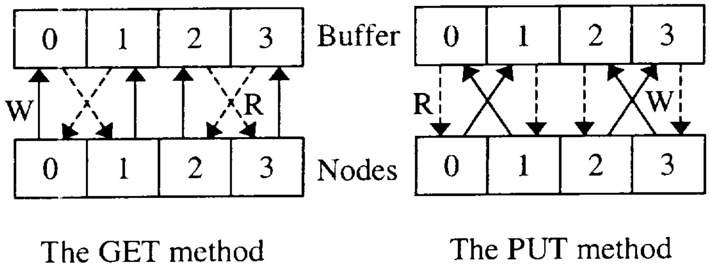

Figure 3: Two ways of updating the shared memory buffer. Arrows with solid line indicate writing, while the dashed lines indicate reading.

The code fragments in Figure 4 show the nature of the coding differences when employing the various communication libraries. The code implements the communication of one word in variable A from process P1 to process P2. In the MPI message passing version process P1 issues a call to mpi_send while process P2 makes the corresponding call to mpi_receive. When using the MPI-2 extension for one-sided communication, process P1 writes A to the shared memory buffer. Then the processes synchronize via a call to mpi_win_fence before process P2 issues a call to mpi_get to read A. For the SMP based implementation we show the use of point-to-point synchronization. Process P1 write A to its segment of the shared memory buffer. For simplicity we assume that the size of the segment is 1 and use the process ID of P1 to index the buffer. Then process P1 sends a signal to P2. Process P2 waits until it receives a signal from P1 and then reads the updated value from the buffer.

MPI	MPI-2	SMP Signal/Wait
if (iam .eq. P1) then	if (iam .eq. P1) then	if (iam .eq.P1) then
call mpi_send(...A, P2,...)	buffer (1) = A	buffer (P1) = A
endif	endif	call smp_signal (P2)
if (iam eq. P2) then	call mpi_win_fence(...)	endif
call mpi_receive(...B, P1,...)	if (iam .eq. P2) then	if (iam .eq. P2) then
endif	call mpi_get (..buffer, P1,...)	call smp_wait (P1)
	B = buffer (1)	B = buffer (P1)
	endif	endif

Figure 4: Code examples for communication operations

3.2. BT and SP Benchmarks

BT and SP benchmarks have a similar structure: each solves three systems of equations resulting from an approximate factorization that decouples the x, y and z dimensions of 3-dimensional Navier-Stokes equations. These systems are block tridiagonal of 5×5 blocks in the BT code and scalar pentadiagonal in the SP code. Each direction is alternatively swept.

The MPI implementations of BT and SP employ a multi-partition scheme [2] in 3-D to achieve good load balance and coarse-grained communication. In this scheme, processors are mapped onto sub-blocks of points of the grid in a special way such that the sub-blocks are evenly distributed along any direction of solution, as illustrated in Figure 3 for a 2-D case. Throughout the sweep in one direction, each processor starts working on its sub-block and sends partial solutions to the next processor before going into the next stage. Communications occur at the *sync points* as indicated by gray lines in Figure 5.

In the RMA implementations of the benchmarks, communications are handled by data exchange through the shared memory buffers and proper synchronization primitives. As mentioned in Section 3.1, we have used two methods to handle the communication at the *sync points* in the solvers: barrier synchronization (BAR) and signal/wait (SW). With the BAR method, all processors copy local data to their designated shared memory buffers and place a global barrier before copying the shared data to the local area. With the SW method, each sending processor copies local data to its designated shared memory buffer and signals its neighbor the shared data is ready; each receiving processor waits for a signal from its neighbor and, then, copies the shared data to its local area. In essence the SW approach is very similar to SEND/RECV in the message passing except that data are exchanged directly through the shared memory buffer rather than messages. To avoid that data in the shared buffer is overwritten before it has been read in the previous stage, we have subdivided each shared buffer area into separate sections for each stage.

Besides in the main solvers, communications also occur in copy_faces where all processors exchange solutions for the ghost points in all three directions. It is straightforward to use global barrier synchronization for this case.

We also produced versions of BT using the PUT and GET methods for updating the shared memory buffer as described in Section 3.1. The performance of different versions will be compared in Section 4.

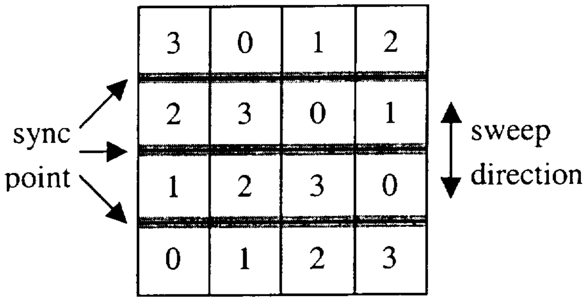

Figure 5: The multi-partition scheme in 2-D. Four processors are evenly mapped onto 4x4 sub-blocks in a 2-D grid. The solving (or sweep) direction is in vertical.

3.3. LU Benchmark

LU benchmark employs the symmetric successive over-relaxation (SSOR) scheme to solve 3-D Navier-Stokes equations. The inherited data dependences in the scheme require the solutions at $(i+e,j,k)$, $(i,j+e,k)$ and $(i,j,k+e)$, where e=−1 or +1, be available before the calculation at (i,j,k) is performed. The MPI implementation of LU utilizes a 2-D partitioning of the grid onto processors and a 2-D coarse-grained pipeline model [9] for parallelization. To illustrate the pipeline method Figure 6 shows a case of a 1-D pipeline in which data are distributed in the J direction among four processors. Processor 0 starts from the low-left corner and works on one slice of data for the first K value. Other processors are waiting for data to be available. Once processor 0 finishes its job, processor 1 can start working on its slice for the same K and, in the meantime, processor 0 moves onto the next K. This process continues until all the processors become active. Then they all work concurrently to the opposite end, as indicated by the large arrow in the figure. The cost of pipelining results mainly from the wait in startup and finishing. A 2-D pipelining can reduce the wait cost and was adopted in the MPI version of LU.

Implementing the SMP version of LU is relatively simple because of the use of the Signal/Wait functions for point-to-point synchronizations in the 2-D pipeline. Shared memory buffers were allocated large enough to hold boundary points in one K slice assigned to each processor. Special care has been taken to guide the update of the shared memory buffers during the K sweep so that these buffers are properly copied to the local areas before their values are overwritten at the next K slice. We did not use global barrier synchronization to synchronize communications in the pipeline for the two reasons: use of a global barrier would be very expensive, especially when the barrier is inside a loop (K) nest, and bookkeeping the global synchronization points would increase the porting effort. For the same reason we did not implement an MPI-2 version of LU.

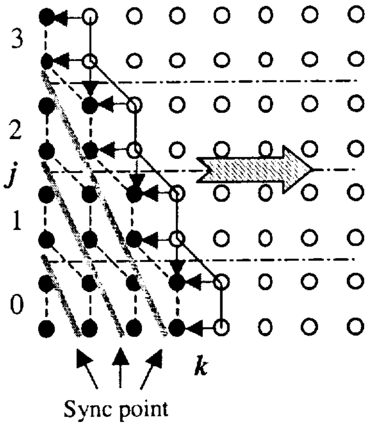

Figure 6: 1-D pipeline used in the SSOR solver of LU. Data are distributed in the *J* dimension.

4. Timing Results

We tested our RMA implementations of the benchmarks on two platforms: an SGI Origin 2000 and a SUN Enterprise 10000. The Origin 2000 consists of 512 MIPS R12K 400MHz processors, each with 8MB L2 cache, running IRIX 6.5. The SGI MIPSpro 7.3.1.2m compiler was used for compilation and the Message Passing Toolkit (MPT) 1.4.0.3 for MPI codes. A highly tuned, efficient implementation of MPI is part of the MPT. Within a single system, MPI messages are moved memory-to-memory. Between nodes of an Silicon Graphics Array system, MPI messages are passed over a HIPPI network. Latency and bandwidth are intermediate between memory-to-memory data exchange and socket-based network communication.

The SUN E10K consists of 16 Ultra SPARC 333MHz processors, running Solaris 7. The Sun Workshop 6 compiler was used in the compilation and SunHPC 3.1 for MPI codes.

There are different classes of the benchmarks depending on their problem size. For our study we considered class A (64x64x64 grid) and class B (102x102x102 grid).

4.1. Comparison of Different RMA Implementation Strategies

We chose the BT benchmark of class A to compare different implementation strategies based on the RMA programming model. We obtained the timings on the SGI Origin. In a first experiment we compared the PUT versus the GET method as described in Section 3.1. For both the SMP and the MPI-2 library, the GET method showed a better performance than the PUT method. The maximum performance advantage of GET versus PUT was about 15% for 256 processes. In the left panel of Figure 7 we show the comparison of SW versus BAR implementation, based on the SMP library. The numbers of MFLOP per second as plotted are those reported by the benchmarks and reflect the scalability. The SW version shows a strong performance advantage over the BAR version, which is due to less time spent in process synchronization. The comparison of SMP versus MPI-2 is shown in the right panel of Figure 7. Since MPI-2 extensions for point-to-point synchronization are not available on the SGI Origin we only compared the BAR versions of the benchmarks. The results were very similar with a slight performance advantage for the MPI-2 based code. We expect MPI-2 to behave close to the SMP SW version once the signal and wait extensions of MPI-2 become available on the SGI Origin.

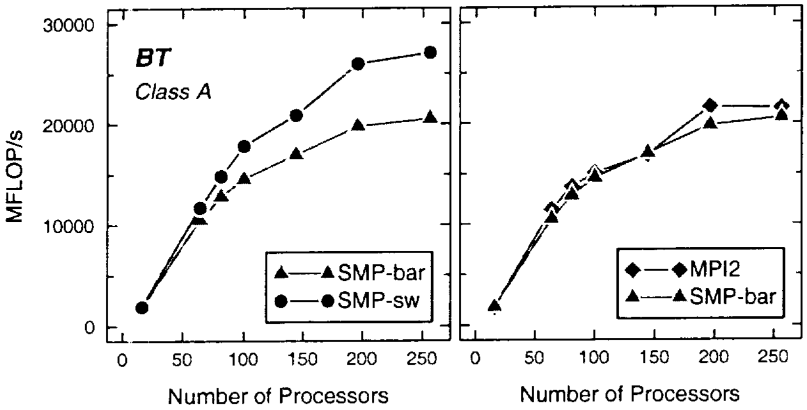

Figure 7: Performance comparison of different implementation strategies based on RMA.

4.2. Comparison of RMA versus Message Passing

In this section we compare the SMP based BAR and SW Get versions on the code against the MPI message passing version for different benchmarks, problem classes, and computer architectures. The reasons why we chose SMP instead of MPI-2 are:

- MPI-2 extensions are not available on our SUN evaluation platform while the SMP library could be easily ported to the SUN.
- MPI-2 extensions for signal and wait were not available on either platform.

We expect similar behavior for MPI-2 once the full functionality becomes available on all platforms.

The MFLOP/s results obtained on the SGI Origin 2000 are summarized in Figure 8 for all three benchmarks and two problem sizes (class A and class B). A straight line in the figure is a reference of a linear speedup based on the timing from the single process run. In all cases, the SMP-SW versions show the best performance, especially on a large number of processors. The MPI versions of BT and SP performed slightly better than the SMP-BAR versions for the class A problem, however, the MPI scaling suffered a performance drop on more than 200 processors for the class B problem. In fact the SMP-BAR versions even outperformed MPI.

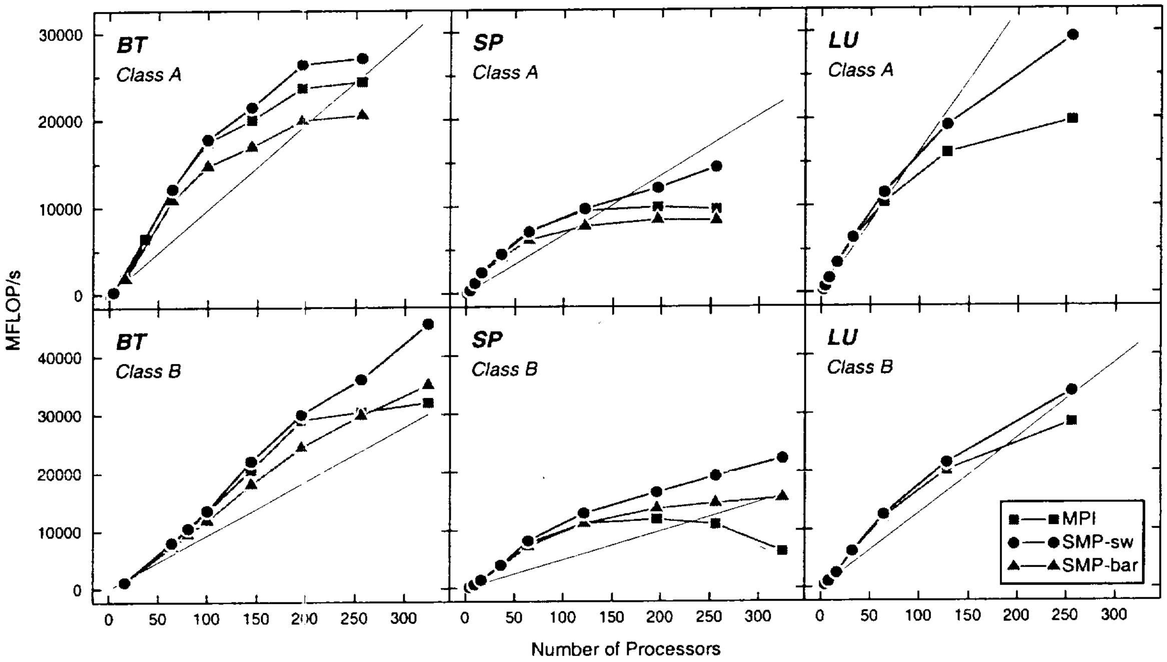

Figure 8: Comparison of MPI, SMP-SW and SMP-BAR implementations of the three benchmarks on the SGI Origin 2000.

OpenMP implementations of the same benchmarks suffer from the fact that parallelism is only exploited at the outermost loop level. The scalability is therefore restricted by the number of grid points in one dimension, which is 64 for class A and 102 for class B.

The MFLOP/s results obtained on the SUN E10K are summarized in Figure 9 for all three benchmarks, class A problem size. Because of the limited number of processors in the machine, the MPI, SMP-SW and SMP-BAR implementations of the benchmarks show very similar performance. However, the SMP version of LU does show better performance than the MPI version on 16 processors, which may indicate the lower overhead of the SMP Signal/Wait functions over the MPI send/receive.

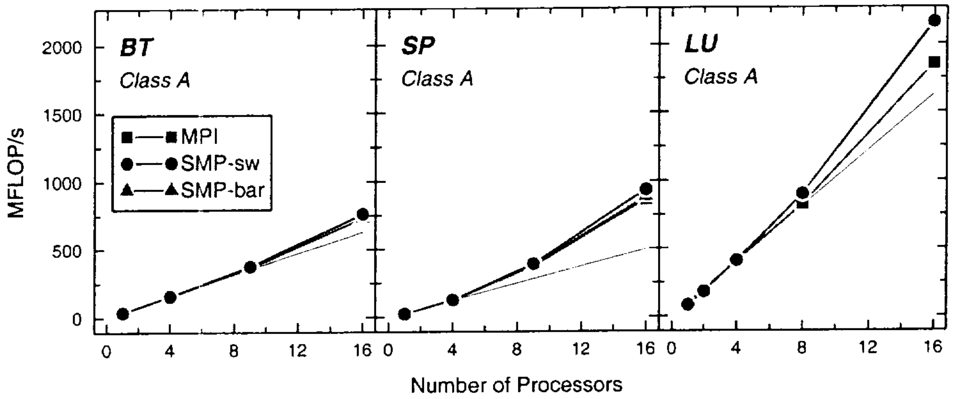

Figure 9: Comparison of MPI, SMP-SW and SMP-BAR implementations of the three benchmarks on the Sun E10K.

5. Related Work

In [8] Taft discusses the performance of a large CFD application. He compares the scalability of message passing versus hybrid parallelization based on RMA and OpenMP. The RMA programming employed in this paper has extended synchronization functionality from the one in [8], but we only consider outer level parallelization.

There are number of papers reporting on comparisons of different programming paradigms. A comparison of message passing and RMA is given in [6] and [7]. The study uses the SGI SHMEM library for RMA programming. The programming paradigm supported and the functionality provided by the SHMEM library is similar to MPI-2. With SMPlib we are employing a somewhat different programming model and compare it to both, message passing as well as one-sided communication.

6. Conclusion

We have ported several benchmarks from the NPB2.3 suite to the RMA programming model. Porting the code was straightforward, since we could adopt the same domain decomposition approach in the message passing implementation. We compared different implementation strategies of RMA for shared memory computer architectures. Point-to-point synchronization and the GET memory access showed the best performance. In comparing RMA versus message passing we found that RMA yielded better scalability.

The MPI-2 extensions for one-sided communication provide support for RMA programming, but the full functionality is currently not available on many hardware platforms. As an alternative programming paradigm to the one provided by the MPI-2 extensions we have implemented the SMPlib library for RMA support. SMPlib provides functionality for process creation, allocation of shared memory as well as barrier and point-to-point synchronization. The library could be easily ported to different hardware platforms and the performance was comparable to MPI-2 based code where available.

We are currently working on porting full-scale applications to the RMA programming model. We also plan develop hybrid versions of these applications with RMA on the outer and OpenMP on the inner level of parallelism.

Acknowledgements

This work was partially supported by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995. http://www.nas.nasa.gov/Software/NPB.

[3] MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich.

[4] MPI–2: Extensions to the MPI Interface, http://www-unix.mcs.anl.gov/mpi/mpich.

[5] OpenMP Fortran Application Program Interface, http://www.openmp.org/.

[6] H. Shan, J. Pal Singh, "A comparison of MPI, SHMEM, and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessor", *International Journal of Parallel Programming*, Vol. 29, No. 3, 2001.

[7] H. Shan, J. Pal Singh, "Comparison of Three Programming Models for Adaptive Applications on the Origin 2000", *Journal of Parallel and Distributed Computing 62, 241-266, 2002.*

[8] J. Taft, "Achieving 60 GFLOP/s on the production CFD code OVERFLOW-MLP," *Parallel Computing*, 27 (2001) 521.

[9] M. Yarrow and R. Van der Wijngaart, "Communication Improvement for the LU NAS Parallel Benchmark," *NAS Technical Report NAS-97-032*, NASA Ames Research Center, Moffett Field, CA, 1997.