Message Passing Vs. Shared Address Space on a Cluster of SMPs

HONGZHANG SHAN, JASWINDER PAL SINGH

Department of Computer Science
35 Olden Street, Princeton University, Princeton, NJ 08544
{shz,jps}@cs.princeton.edu

LEONID OLIKER
National Energy Research Scientific Computing Center
Mail Stop 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
loliker@lbl.gov

RUPAK BISWAS
Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035
rbiswas@nas.nasa.gov

September 24, 2000

Abstract

The convergence of scalable computer architectures using clusters of PCs (or PC-SMPs) with commodity networking has become an attractive platform for high end scientific computing. Currently, message-passing and shared address space (SAS) are the two leading programming paradigms for these systems. Message-passing has been standardized with MPI, and is the most common and mature programming approach. However, message-passing code development can be extremely difficult, especially for irregularly structured computations. SAS offers substantial ease of programming, but may suffer from performance limitations due to poor spatial locality and high protocol overhead. In this paper, we compare the performance of and programming effort required for six applications under both programming models on a 32 CPU PC-SMP cluster. Our application suite consists of codes that typically do not exhibit high efficiency under shared memory programming, due to their high communication to computation ratios and complex communication patterns. Results indicate that SAS can achieve about half the parallel efficiency of MPI for most of our applications; however, on certain classes of problems SAS performance is competitive with MPI. We also present new algorithms for improving the PC cluster performance of MPI collective operations.

1 Introduction

The convergence of scalable computer architectures using clusters of PCs with commodity networking has become an attractive platform for high-end scientific computing. Currently, message-passing and shared address space (SAS) are the two leading programming paradigms for these systems. Message-passing has been standardized with MPI [6], and is the most common and mature programming approach. It provides both functional and performance portability. However, message-passing code development can be extremely difficult, especially for irregularly structured computations. A coherent shared address space has been shown to be very effective at moderate-scale for a wide range of applications when supported efficiently in hardware. The automatic management of naming and coherent replication in this programming model also substantially eases the programming task compared to explicit message passing, especially for complex irregular applications that are naturally becoming increasingly popular as multiprocessesing matures. This ease of programming can often be translated directly into performance gains [19, 20]. Even as hardware-coherent machines replace traditional message passing systems at the high end, clusters of commodity PCs and PC-SMPs have become increasingly popular for scalable computing. On these, the message passing programming model is dominant and the shared address space model unproven since it is implemented in software. Thus, especially given the ease of programming, it is important to understand the performance tradeoffs of message passing with the shared address space programming model on clusters.

Approaches to support a shared address space in software across clusters differs not only in the specialization and efficiencies of networks but also in the granularities at which they provide coherence. Fine-grained software

coherence uses either code instrumentation [15, 5] for access control or commodity-oriented hardware support [22] with protocol implemented in software. Page-grained software coherence takes advantage of the virtual memory management facilities to provide replication and coherence at page granularity [14]. To alleviate false sharing and fragmentation problems, a relaxed consistency model is used to buffer coherence actions. Multiple writer protocols allow more than one processor to modify copies of a page locally and incoherently between the synchronization points, reducing the impact of write-write false sharing, and making the page consistent only when needed. Lu [7] compare the performance of the PVM and the TreadMarks page-based software shared memory library on an 8-processor network of ATM-connected workstations and on an 8-processor IBM SP-2. They find that TreadMarks generally performs a little worse. Karlsson and Brorsson [13] compared the characteristics of communication patterns in message passing and page-based software shared memory programs, using MPI and TreadMarks running on an IBM SP2. They found that the fraction of small messages in the TreadMarks executions lead to poor performance. However, the platforms they use are much lower-performance and smaller scale and not SMP-based. The protocols are not high efficient. Recently, both the communication network and protocols for shared virtual memory have made great progress. Some gigabits-networks (per second) have been put into use. A new SVM [11] protocol called GeNIMA has also been developed for the page-grained shared address space on clusters. GeNIMA uses general-purpose network interface support to significantly reduce protocol overheads. It has been shown to perform quite well for moderate-scale systems on a fairly wide range of applications: achieving at least half of the parallel efficiency of a high-end hardware-coherent system and often exhibiting better behavior [10, 1]. Thus, a study of comparing the performance of GeNIMA against message passing implementations of the same applications, which is the dominant way of programming applications for clusters today, becomes necessary and important.

In this paper, we compare the performance of these two programming models using the best implementation available to us (MPI/Pro from MPI Softtech INC. and the GeNIMA SVM protocol for SAS) on a clusters of eight, 4-way SMPs (for a total of 32 processors). The applications used are those that have been selected to compare the performance and programming ease on hardware-coherent platforms, including regular as well as dynamic irregular applications. Our application suite includes codes that scale well on tightly-coupled machines as well as codes that are challenging to obtain scalable performance on due to their high communication to computation ratios and complex communication patterns. With a couple of exceptions, they are generally applications where developing efficient message passing implementations is not extremely difficult (even though it still takes a lot more work than the shared address space implementations). Porting these applications to the cluster did not require code modifications; however, some optimizations were performed to improve performance on the cluster platform [11, 18]. We find that while some classes of applications can achieve similar performance for MPI and SVM on the cluster, in most cases MPI performs significantly better than the shared address space model. The performance of SVM suffers greatly from the protocol overhead, especially at the synchronization points, which often become the performance bottleneck. Further research into reducing the protocol overhead for SVM is required to achieve high performance. Some of the applications we have selected, such as 1-D FFT and radix sorting, are difficult to implement efficiently in either programming model on a cluster due to the limited bandwidth of the memory bus on the SMP nodes. This increases the challenge for scalable performance since the memory bus is also involved in communication. Some other, irregular and unpredictable applications are challenging to to implement efficiently in message passing but have low communication requirements, so for these the performance of the shared address space programming model is expected to be much better. For example, while we don't have a message passing implementation of this application, the speedups for the volrend volume rendering application is approximately 27 on 32 processors using the same platform with the GeNIMA protocol [10].

Currently, if very high performance is the goal, then the difficulty of MPI programming appears to be justified for commodity clusters of SMPs today. On the other hand, if ease of programming is important then SVM provides it at roughly a factor-of-two cost in performance for many applications (and less for others). This may be considered encouraging for SVM, given the ease of programming advantages for complex applications as well as the difficult nature of our application suite and the relative maturity of the MPI library. Application-driven research into coherence protocols and extended hardware support should reduce SVM and SAS overheads on future systems.

We also present new algorithms for implementing MPI collective functions on our PC cluster platform. Results show that these techniques achieve a significant improvement compared the default MPI/Pro implementation.

The rest of the paper is organized as follows. Section 2 describes the platform we used and the implementation of different programming models on it. Applications are discussed in Section 3. In Section 4, we analyze the performance differences between the two programming models for each application. Section 5 explores new algorithms used to efficiently implement collective functions for MPI. Finally, we present our conclusions in Section 6.

[11] The words *SAS* and *SVM* are used synonymously throughout this paper.

2 Platform and Programming Models

The platform we used for our study is a cluster of 4-way Pentium Pro SMPs. Each node has 4 CPUs running at 200MHz. Each processor has separate 8KB data and instruction L1 caches and a unified 4-way set-associative 512KB L2 cache. Each node has 512MB main memory, running WINDOWS NT 4.0. The nodes are connected together either by Myrinet [2] or Giganet [8]. The SAS and MPI programming models are built on top of these two networks respectively.

2.1 SAS Programming Model

Much research has been done in the design and implementation of shared address space for clustered architectures, both at page granularity and at finer fixed granularities through code instrumentation. Among the most popular way to support a coherent shared address space in software on clusters, is page-based shared virtual memory (SVM). SVM takes advantage of the virtual memory management facilities to provide the replication and coherence at page granularity. To alleviate false sharing and fragmentation problems, SVM uses the relaxed memory consistency model to buffer coherence actions such as invalidations or updates, and postpone them until a synchronization point. Multiple writer protocols are used to allow more than one processor to modify copies of a page locally and incoherently between synchronization points, reducing the impact of write-write false sharing and making the page consistent only when needed by applying `diffs` and `write` notices. Many different protocols have been developed which use different timing strategies to propagate write notices and apply the invalidations to pages. Recently, a new protocol for SVM called `GeNIMA` has been developed and has shown good performance at moderate-scale systems for a fairly wide range of applications: achieving at least half of the parallel efficiency of a high-end hardware-coherent system and often exhibiting much better behavior [10, 1]. It uses general-purpose network interface support to significantly improve protocol overheads. Thus, in this study we select the `GeNIMA` as our protocol for the SAS programming model. `GeNIMA` is built on top of the VMMC, a high-performance, user-level virtual memory mapped communication library [4]. VMMC itself, runs on top of the Myrinet network.

Each SMP node in our cluster is connected to a Myrinet system area network via a PCI bus. The Myrinet network interfaces are connected together through a single 16-way Myrinet crossbar switch, thus minimizing contention in the interconnect. Each network interface has a 33MHz programmable processor and connects nodes to the network with two unidirectional links of 160 MB/s peak bandwidth each. The actual node-to-network bandwidth is constrained by the 133MB/s PCI bus.

The parallelism constructs and calls needed by the SAS programs are exactly the same as those used in our hardware-coherent platform implementation (SGI Origin 2000) [17, 18, 19]. This make portability trivial between these platforms.

2.2 Message Passing Programming Model

The message passing implementation used in this study is from MPI Software Technology Inc., developed directly on top of Giganet networks by the VIA [9] interface. By selecting MPI/Pro, instead of building our own MPI library from VMMC, we can compare the best known versions of both programming models. Thus our final conclusions are not affected by a poor implementation of the communication layer. Fortunately, the VIA interface and the VMMC have similar latency (Figure 1) and bandwidth (Figure 2) characteristics on our cluster platform. Giganet performs slightly better for short messages and while Myrinet has a small advantage for for larger messages. Thus, for message passing programs, there should be little performance difference for similar implementations across these two networks. Similarly to Myrinet, the Giganet network interface is connected together by a single Giganet crossbar switch.

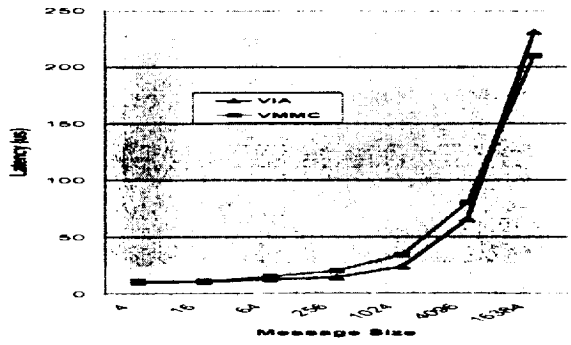

Figure 1: The latency of different message sizes for VMMC and VIA communication interface

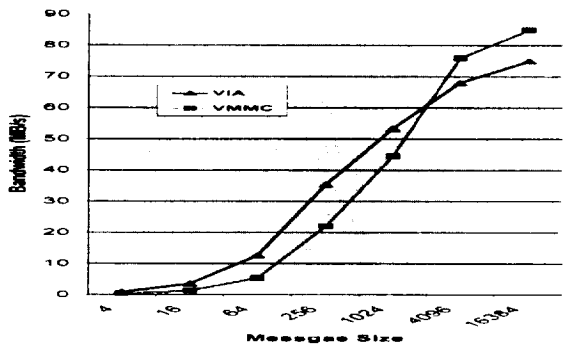

Figure 2: The bandwidth of different message sizes for VMMC and VIA communication interface

3 Applications

Our application suite consists of codes used in previous studies to examine the performance and implementation complexity of various programming models on hardware-supported cache-coherent platforms. These codes include regular applications (FT, OCEAN and LU) and irregularly structured applications (RADIX Sorting, SAMPLE Sorting and N-body). All six codes have either high communication to computation ratio or complex communication patterns, making scalable performance a difficult task on cluster platforms. FFT uses a non-localized but regular all-to-all personalized communication pattern to perform a matrix transposition; i.e. every process communicates with every other, sending different data across the network. OCEAN exhibits primarily nearest neighbor patterns, but in a multi-grid, rather than a single-grid formation. RADIX sorting uses all-to-all personalized communication but in an irregular and scattered fashion. SAMPLE sorting also uses all-to-all personalized communication, however, the communication patterns are more regular than in RADIX sorting. LU uses one-to-many non-personalized communication. Finally, N-BODY requires all-to-all, all-gather communication and unpredictable send/recv communications patterns. These applications have shown high performance under both MPI and SAS, for reasonably large data sets on hardware-supported coherent platforms.

Most of the MPI programs were been ported directly onto the cluster platform without any changes. However, OCEAN and RADIX sorting required some changes for high-performance. In OCEAN, the matrix is partitioned by the rows instead of the blocks (see Figure 3). This allows each processor to communicate with only its upper and lower neighbors, thus reducing the number of messages sent across the network, while improving the spatial locality of the communicated data. For RADIX, in the key exchange stage, each processor sends only one message to every other processor, containing all its chunks of keys that are destined for the destination processor. The destination processor then reorganize the data chunks to their correct positions.

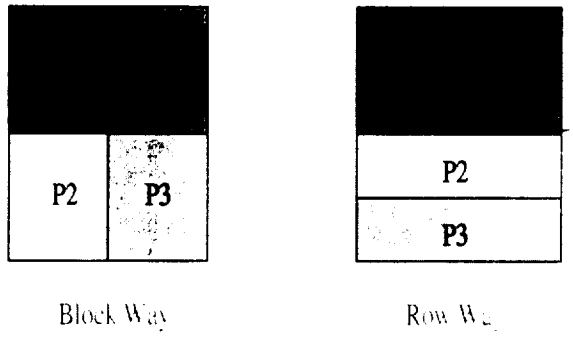

Figure 3: The partition method for ocean for four processor case: block way vs. row way

On hardware-supported coherent platform, each processor sends contiguously-destined chunk of keys directly as a separate message. This is done so that the data can immediately be placed into the correct position at the destination processor. As a result, multiple message are sent from one processor to every processor. While this method succeeds on the hardware-supported coherent systems, the modified approach is better suited for cluster platforms. On clusters systems, there is a performance gain when reducing the number of sent messages even though local computation

is increased. In order to study two-level architecture effect (intra node and inter node), we test our applications by reorganizing the communication sequence (intra-node first, inter-node first, intra-node and inter-node mixed). Interestingly, our results show that the MPI programs are insensitive to the communication sequence and this method of exploiting a two-level communication hierarchy, and the various communication sequences have almost no effect on the application performance.

For the SAS codes, FFT, LU, and SAMPLE sorting, were ported without any modifications. In RADIX sorting, we used the improved version [18]. Here, instead of exchanging the keys in a scattered way, keys that are destined for the same processor are buffered together and then written to their destination. Several modification were applied to original version of OCEAN to improve performance [10] on the clusters. The matrix was partitioned by rows across processors instead of blocks, and significant changes were made to the data structures. The N-BODY code also required substantial modifications since the original version suffered from the high overhead of synchronizations used during the shared-tree building phase. A new tree building method called barnes-spatial has been developed to completely eliminate these expensive synchronization operations [16].

These applications have been previously used to evaluate the performance of different programming models on a hardware-supported cache-coherent platform. In that study, it was shown that SAS programs provide substantial ease of programming compared to message-passing implementations, and performance varies depending on the application but is often better for SAS as well. The ease of programming holds true on cluster systems, although some SAS code restructuring was required to improve performance. For example, in N-body the tree-building methodology has been changed from the original synchronization intensive scheme to spatial method. Nonetheless, the implementation is still easier than the message-passing approach, as has been argued earlier in the hardware-coherent context [12], where it was shown that this implementation is valuable for high-end scalability even on hardware-coherent machines.

A comparison between SAS and MPI programmability is presented in Table 1. Notice that SAS programs require fewer lines of essential code lines (excluding the initialization and debugging code and comments) compared with message passing. As application complexity (e.g. irregularity and dynamic nature) increases we see a more significant reduction of programming effort using SAS.

	FFT	LU	OCEAN	RADIX	SAMPLE	N-BODY
SAS	210	309	2878	201	450	950
MPI	222	470	4320	384	479	1371

Table 1: The number of the essential code lines needed by SAS and MPI for different applications.

4 Performance Analysis

In this section, we compare the performance of our applications using both programming paradigms. We first examine the speedup numbers, and then analyze the performance in more detail using time breakdowns. The speedups for different programming models are based on the best sequential time, without any parallel programming model overhead. The total running time is divided into three components: LOCAL, RMEM, and SYNC. The LOCAL time includes the CPU computation time and the CPU waiting time for local cache misses. The RMEM time is the CPU time spent for remote communication and SYNC is the time spent for synchronization. We select two data sets for each application. First we examine a "basic" data set, at which the Shared Virtual Memory begins to perform reasonably "well" [10]. Next, we use a larger data set, since in general, increasing the problem size tends to improve many inherent program characteristics, such as load balance, communication to computation ratios, and spatial locality.

4.1 FFT

FFT has very high communication to computation ratio, which diminishes only logarithmically with problem size. It needs a non-localized but regular all-to-all personalized communication pattern to perform the matrix transposition, and there is no overlap between the transposition and computation. It is much more difficult to achieve performance on the 1-dimensional FFT used here than on higher-dimensional FFTs. The speedups for MPI and SAS are presented in Figure 4.

Both MPI and SAS did not achieve good scalability. Increasing the data set size helped but not significantly. This is mainly due to the pure communication transpose stage whose communication to computation ratio does not change with problem size. It occupies only 16% of the total execution time in sequential run. However, the percentage increases to 50% for 32-processor run. Dealing with the scaling of the pure all-to-all communication is very difficult. As the number of nodes that fetch data from each node increases, the contention in the network interface of the servicing node increases. At the same time, since these remote requests need to access the memory bus, the additional contention for the memory bus affects local memory access time as well. Also, the program suffers from the low

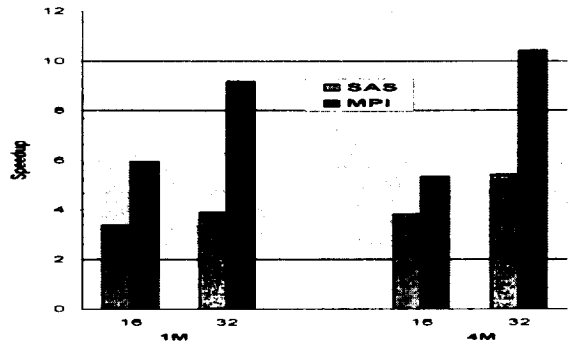

Figure 4: Speedups of FFT for SAS and MPI on 16 and 32 processors for 1M and 4M data set size

bandwidth of the memory bus on our commodity 4-way SMP nodes, which is a significant problem with commodity SMPs. High contention is caused when 4 processors work simultaneously within a node. For example, the LOCAL time (which includes local memory stall time) for 4M data set when using 2 processors is about 6s. This drops to only 4.8s (compared to ideal of 3s), when 4 processors are used.

Even though both programming models do not scale well for FFT, MPI significantly outperforms SAS. To better understand the performance difference, Figure 5 presents the breakdown for 4M data set size running on 32 processors.

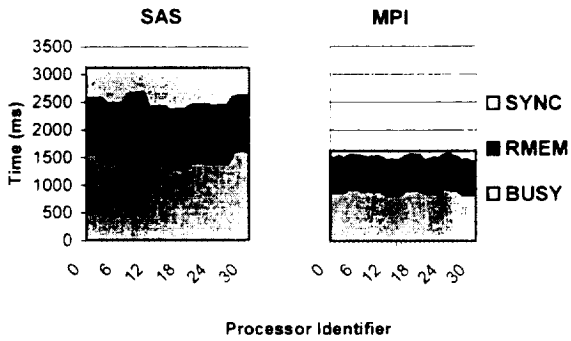

Figure 5: The FFT time breakdown for SAS and MPI on 32 processors 4M data set size

Surprisingly we find that all the three components (LOCAL, RMEM, and SYNC) times are much higher in SAS than in MPI. In order to maintain the page coherence, high protocol overhead is introduced in SAS programs, including computing the diffs, creating the timestamps, creating the write notices, and garbage collection. This dramatically increases the compute time, and also causes an increase in local cache misses for the application data, leading to higher LOCAL time. The diffs generated for the coherence will be immediately propagated to the home of the pages, thus increasing the network traffic and possibly causing more contention. At synchronization points, handling the protocol requirements highly dilates the synchronization interval, including the expensive invalidation of necessary pages. In the MPI program, all these protocol overheads do not exist. MPI does need to pack the data at the source and unpack them at the destination to make communication more efficient. However, this overhead is much smaller and completely local compared with the protocol overhead in SAS program. If the SAS program were structured such that each the sub-matrix transposed to a different processor is allocated separately (similarly to MPI program), instead of all being allocated together in a shared data structure of a row set or a full matrix (which is most natural due to the row-based partitioning of the computation), the performance of the transpose could be improved. But this causes a lot of the programming ease of SAS to be given up, and in fact can make the row-wise local FFTs complex to implement.

4.2 OCEAN

OCEAN exhibits a commonly used nearest neighbor pattern, but in a multi-grid rather than a single-grid formation. The communication to computation ratio is large for smaller problem sizes but diminishes rapidly with increasing problem sizes. The speedups are shown in Figure 6. The speedups are still relatively lower compared with those that we have achieved on the hardware-supported cache-coherent platform. Larger data sets improve performance, especially for the MPI programs.

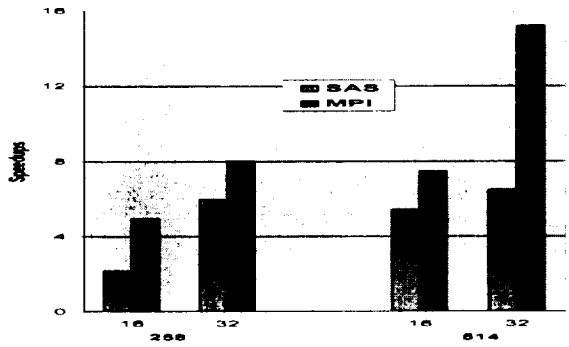

Figure 6: Speedups of OCEAN for SAS and MPI on 16 and 32 processors for 258x258 and 514x514 grid sizes

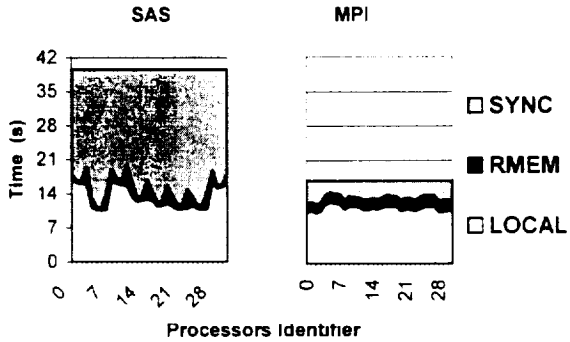

Figure 7: The OCEAN time breakdown for SAS and MPI on 32 processors 514x514 grid size

The SAS program suffers from the expensive overhead of synchronization. This is clearly shown in Figure 7. After each nearest-neighbor communication, a synchronization operation is required in SAS to maintain coherence. Thus, there are many of thousands of barrier operations throughout OCEAN. Further analysis of the synchronization time shows that about 50% of the synchronization time is spent waiting, and 50% of the time is for protocol processing [1]. Thus, the synchronization cost can be improved either by reducing protocol overhead or by increasing the data set size. There is not enough computational work between the synchronization points for the 514 × 514 grid size, especially when this grid size is further coarsened into smaller grid sizes during program execution. However, OCEAN has very high memory requirement due to use of more than twenty large data arrays. This prevents us from running larger data sets. In the MPI program, the synchronization is much cheaper since it is implicitly implemented in the send/receive pairs.

4.3 LU

LU uses one-to-many non-personalized communication: the pivot block and the pivot row blocks are communicated to $\sqrt{p}$ processors each. The communication needs are relatively small compared with our other applications. Thus, we expect better performance for this application, as seen in Figure 8. From the time breakdown in Figure 9, we also see that most of the overhead is in LOCAL time. Here, the communication cost is very small. Further improvement can be achieved by reducing the synchronization cost, though this is mainly wait time caused by load imbalance.

Notice that for LU, the performance of SAS is very close to MPI since both of them have similar time breakdowns. The protocol overhead running SAS program becomes less important for this application. This is because, unlike in FFT, the matrix is already organized in a 4-dimensional array to ensure that the blocks assigned to a processor are allocated locally and contiguously. Thus, each processor will only need to modify its own blocks which are allocated locally, and the modifications are immediately applied to the data pages. No diffs are generated and propagated to other nodes. This will greatly reduce the overhead of protocol processing. These performance results indicate that the two programming models do not show much performance difference, due to the relatively low communication requirements of LU.

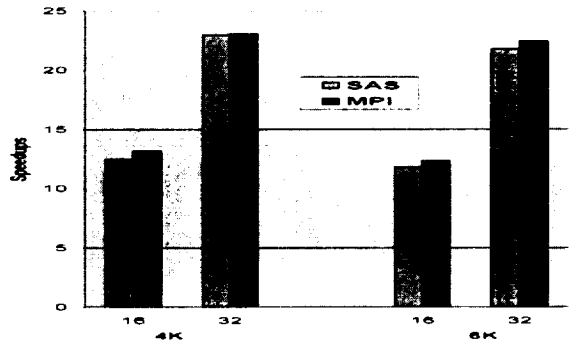

Figure 8: Speedups of LU for SAS and MPI on 16 and 32 processors for 4096x4096 and 6144x6144 matrix sizes

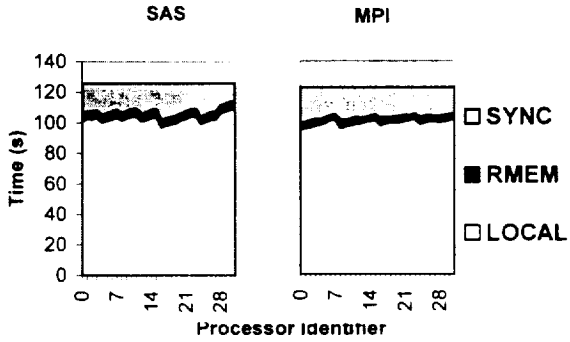

Figure 9: The LU time breakdown for SAS and MPI on 32 processors for 6144*6144 matrix size

4.4 RADIX

The above three applications (FFT, OCEAN, and LU), are all show regular characteristics. The following three codes we investigate, RADIX, SAMPLE, and N-BODY, are irregularly structured. RADIX sort uses all-to-all personalized communication but in an irregular and scattered fashion. It also has a very high communication to computation ratio that is independent of problem size and number of processors. This application has high bandwidth requirements for the memory bus, which is often not satisfied on current SMP platforms. Thus, high contention is caused on the memory bus when 4 processors are used on a node. The "aggregate LOCAL" time across processors is much higher than the uniprocessor case. This leads to the poor performance shown in Figure 10. MPI still performs better than SAS. From the time breakdown in Figure 11, we can find that the RMEM time and SYNC time are much higher for SAS. This is due to similar reasons as discussed in the FFT subsection, as the communication is all-to-all in chunks here as well (albeit irregular).

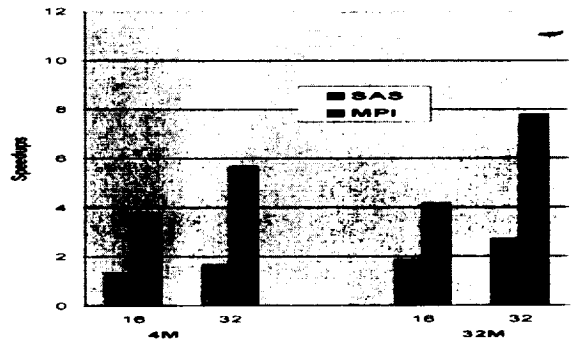

Figure 10: Speedups of RADIX for SAS and MPI on 16 and 32 processors for 4M and 32M integers

The implementation of the all-to-all communication in the MPI Radix program on the cluster is different from that

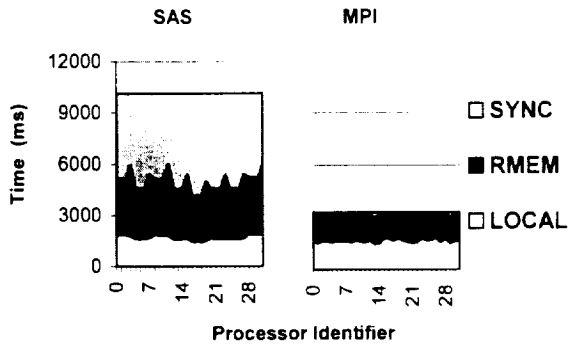

Figure 11: The RADIX time breakdown for SAS and MPI on 32 processors for 32M integers

used in the MPI version on a hardware-supported cache-coherent platform. In the cluster MPI program, each processor sends only one message to every other processor, containing all the chunks that are destined for the destination processor. The destination processor will then reorganize the data chunks to their correct positions. This is similar to the algorithm used in the NAS parallel applications IS [3]. But on the hardware-supported cache-coherent platform, each processor will send each contiguously destined chunk directly as a separate message so that the destination can put the data into the correct position, thus leading to multiple messages from each processor destined for every other processor. This is a tradeoff between computation and communication and depends on the cost of communication messages on the machines. On high-overhead and low-bandwidth clusters, using less messages is more important than the computation involved in gathering and scattering chunks.

4.5 SAMPLE

SAMPLE sorting also uses an irregular personalized all-to-all communication, but compared with RADIX sorting, it is more regular and the communication is much better structured. SAMPLE speedups are presented in Figure 12. Compared with RADIX, the performance is much better. Notice that we use the same sequential time to compute the speedups for both RADIX and SAMPLE sorting. In SAMPLE sort, each processor does a local sort on its partitioned data first using the radix sort, then performs the all-to-all communication to exchange keys, followed by another local sort on the newly-received data. In the sequential case, only one local sort is enough to sort all the keys. Thus, we can reasonably expect SAMPLE sort to achieve only 50% parallel efficiency.

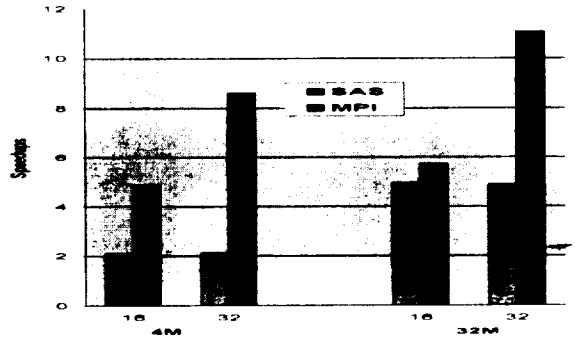

Figure 12: Speedups of SAMPLE for SAS and MPI on 16 and 32 processors for 4M and 32M integers

If we look at the SAMPLE time breakdown in Figure 13, and compare with RADIX time breakdowns in Figure 11, for both MPI and SAS, the RMEM time and SYNC time are greatly reduced. On the SGI Origin2000, we found that in most cases RADIX performs better than SAMPLE sort. However, on the cluster platform the opposite is true. SAMPLE sorting outperforms RADIX sorting. This result verifies that reducing messages is much more important on the cluster than increasing the computations (despite the large increase in local computation here).

Note that the LOCAL time in SAMPLE sort is only slightly higher than in RADIX, even though much more computation is performed in SAMPLE. This means that the contention on the memory bus for RADIX sorting is much higher than for the SAMPLE sort, due to the more sequential memory access patterns.

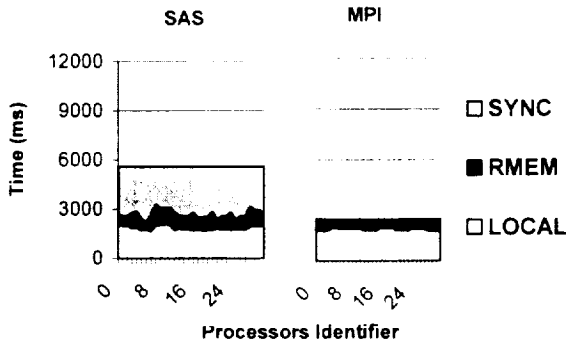

Figure 13: The SAMPLE time breakdown for SAS and MPI on 32 processors for 32M integers

4.6 N-BODY

Finally, we examine the performance of the N-BODY application. From Figure 14, we find that MPI still performs better than SAS, especially with the increase of data sets. For 128K bodies and 32 processors, MPI achieves a speedup of 27 compared to speedup of 15 using SAS. The time breakdown for 128K data set on 32 processors is shown in Figure15. SAS has higher SYNC time and RMEM time, but the SYNC time is much higher and dominates. This is because at each synchronization point, many diffs and write notices need to be processed. Also, many shared pages have to be invalidated. So the synchronization wait time is further dilated due to serialization. Further analysis shows that 82% of the barrier time is spent on protocol handling. This expensive synchronization problem occurs in all of our applications except LU, and the performance of SVM programs suffer heavily from it. Further research on the SVM coherence protocol should focus on reducing the synchronization cost. Possible approaches include applying the diffs before the synchronization points, moving invalidating shared pages out of synchronization points, and increasing hardware supports.

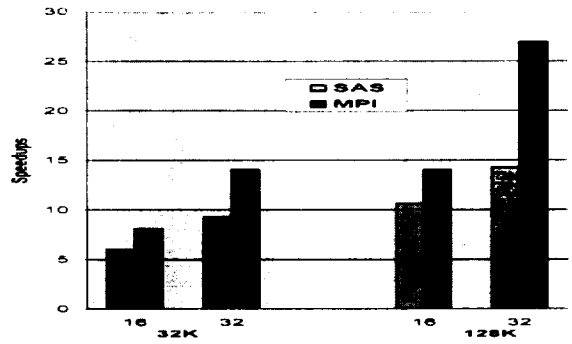

Figure 14: Speedups of N-BODY for SAS and MPI on 16 and 32 processors for 32K and 128K bodies

Unlike our other five applications, the MPI version of N-BODY has a higher time LOCAL than the SAS code. This is due to the use of different high level algorithms for each programming model. In the SAS program, a shared tree is built and each processor builds one part of it; while in MPI program, a locally essential tree is used. Building the locally essential tree is quite expensive and involves a lot of more computation. A processor has to first build a local tree using those bodies partitioned to it, then it computes the nodes needed by every other processors from its local tree and sends those nodes to their destination. After receiving all the necessary nodes for the ensuing force calculation, it has to add them into its local tree and generate the locally essential tree. Thus, a lot of computation overhead has been introduced into the MPI version for building locally essential trees. With the increase of larger data sets, these differences in building the tree become less important and the force calculation phase begins to dominate.

5 Implementation of Collective Functions for MPI

An interesting question for clusters, and particularly hybrid clusters of SMPs, is how to structure collective communication. In the MPI library, the communication functions can be divided into three categories: the basic send/receive

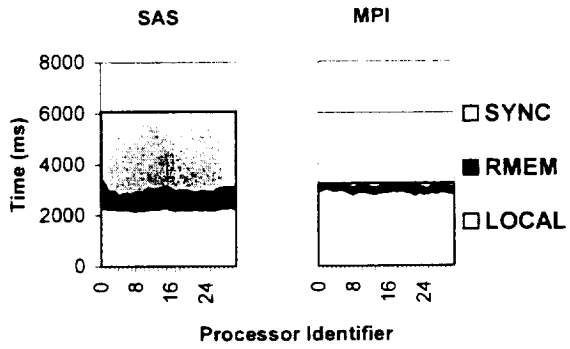

Figure 15: The N-BODY time breakdown for SAS and MPI on 32 processors for 128K bodies

functions, collective functions, and other operations. The performance of the basic send/receive are mainly dependent on the underlying communication hardware and low-level software. However, the performance of the collective functions are affected by their implementation algorithms. Research in this area has been performed for several other platforms [21]. In this section, we discuss the algorithms suitable for our platform: a clusters of 4-way SMPs. Specifically, we explore the algorithms for two collective functions, MPI_Allreduce and MPI_Allgather that are used in our applications. Here, we call the MPI/Pro implementation the "original" (the exact algorithms used are not well documented) and use it as our baseline.

5.1 MPI_Allreduce

The most commonly used algorithm is the binary tree (B-Tree) which is shown in Figure 16. The structure of our 4-way SMP nodes leads us to change the lowest level of the tree to a four-way structure (called B-Tree-4). And within a node, the communication can either be implemented by shared memory or basic MPI send/receive functions. We observe no difference in performance between these two in-node approaches for the collective communication. The result for reducing a double variable is shown in Table 2. The binary tree algorithm performs similarly to the MPI/Pro implementation. The B-Tree-4 algorithm performs somewhat better.

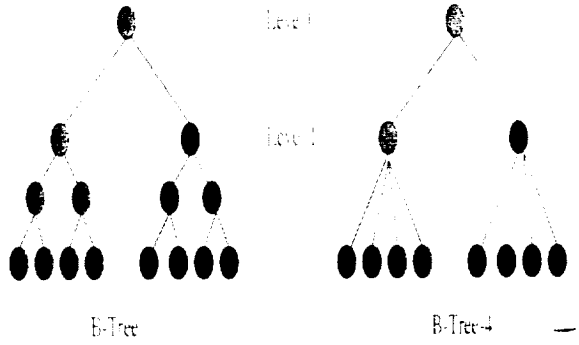

Figure 16: The algorithms used to implement the MPI_Allreduce using two nodes as an example

Algorithm	Original	B-Tree	B-Tree-4
Time (us)	1117	1035	981

Table 2: The time needed for MPI_Allreduce on 32 processors (8 nodes) for different algorithms

5.2 MPI_Allgather

We explored several algorithms for this case. The first method is the binary tree method, and the second is the B-Tree-4 approach. To understand their behavior, we examine level 0 and level 1 (see Figure 16). In the B-Tree-4 algorithm,

after the level-0 processor collects all the data, it broadcasts all the collected data to level-1 processors, then from level-1 to level-2 and so on. At this time, each processor in level 1 already owns those data collected from the subtree rooted at it. So it is not necessary to broadcast all the data back to them. Instead, those processors in level 1 can immediately exchange data themselves. There are two processors left in level 1. These two can just exchange data themselves. We call this algorithm B-Tree-4-New. We can further extend this idea to level-2. In the B-Tree-4-New algorithm, the four processors in level-2 exchange data amongst themselves. Each processor needs two send/receive function calls. This method is called B-Tree-4-New-2. In Table 3, we present the time needed for the different algorithms to perform the allgather function for 1K-integer and 1-integer messages. We found that using the new algorithm, the performance on our platform has been greatly improved, indicating that the original MPI-Pro algorithm doesn't use the optimization of sending down only the necessary data and performing a data exchange. However, since most of the remote communication time in the applications was spent on the send/receive functions, the overall performance was only slightly improved.

Algorithm	Original	B-Tree	B-Tree-4	B-Tree-4-New	B-Tree-4-New-2
Time (1K)	1538	1540	1479	1395	1124
Time (1)	1633	1052	993	994	975

Table 3: The time needed for MPI_Allgather on 32 processors (8 nodes) for different algorithms using 1k and 1 integers

6 Conclusion

In this paper, we studied the performance of and programming effort for six applications using message-passing and SAS programming on a 32 CPU PC cluster. The system consisted of eight 4-way Pentium Pro SMPs running WINDOWS NT 4.0. To create a fair comparison between the two programming methodologies, we used the best known implementations of the communication libraries. The message-passing version of MPI/Pro is implemented directly on top of the Giganet network by the VIA interface, which the Giganet network interface implements in hardware. The SAS implementation is a shared virtual memory (SVM) implementation and uses the GeNIMA protocol over the VMMC low-level communication library, which is implemented in firmware and software on the Myrinet network. Experiments showed that VIA and VMMC have similar latency and bandwidth characteristics on the cluster platform. The GeNIMA protocol has been developed for page-grained shared address space on clusters, and uses general purpose network interface support to significantly reduce protocol overhead. Our application suite consists of several codes that are challenging for scalable performance due to their high communication to computation ratios and complex communication patterns.

Three regular applications (FFT, OCEAN, and LU) and three irregularly structured codes (RADIX, SAMPLE, and N-BODY) were presented. Porting these applications did not require code modifications; however, some optimizations were performed to improve performance on the cluster platform. Changes included reducing the number of messages in the message-passing versions, and removing fine-grained synchronizations from SAS codes. FFT, OCEAN, RADIX, and SAMPLE did not scale well under both programming models due to their high communication to computation ratios and/or the limited bandwidth of the memory bus on the 4-way SMP nodes (and the resulting contention between communication and local computation). LU and N-BODY showed better performance characteristics due to lower communication-to-computation ratios.

Overall, SVM provides a substantial ease of programming, especially for the more complex applications which are irregular or dynamic in nature. However, unlike in a previous study for hardware-coherent machines where the shared address space implementations were also performance-competitive with MPI, despite all the research in SVM protocols and communication libaries in the last several years SVM achieved only about half the parallel efficiency of MPI for most of our applications. LU was an exception, in which the SVM implementation achieved very similar performance to the MPI version. The higher runtimes of the SVM versions were due to high cost of the protocol overhead associated with maintaining page coherence and implementing synchronizations. These costs include: computing diffs, creating timestamps, generating write notices, and performing garbage collection. Thus, if very high performance is the goal, then the difficulty of MPI programming appears to be justified for commodity clusters of SMPs today. On the other hand, if ease of programming is important then SVM provides it at roughly a factor-of-two cost in performance for many applications (and less for others). This may be considered encouraging for SVM, given the ease of programming advantages for complex applications as well as the difficult nature of our application suite and the relative maturity of the MPI library. Application-driven research into coherence protocols and extended hardware support should reduce SVM and SAS overheads on future systems.

Finally, we presented new algorithms for implementing MPI collective functions on our PC cluster platform. Results show that some of these techniques achieve a significant improvement compared the default MPI/Pro implementation.

References

[1] A.Bilas, C.Liao, and JP.Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *ISCA'99*, May 1999.

[2] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] NASA Ames Research Center. The NAS parallel benchmarks 2.0. http://science.nas.nasa.gov/Software/NPB, November 1995.

[4] C. Dubnicki, A. Bilas, Y. Chen, S. Damian, and K. Li. Vmmc-2: Efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug 1997.

[5] Ioannis Schoinas et al. Fine-grain access control for distributed shared memory. In *the sixth ASPLOS*, October 1994.

[6] Message Passing Interface Forum. Document for a standard message-passing interface. http://www-c.mcs.anl.gov/mpi, June 1993.

[7] H.Lu, S.Dwarkadas, A.L.Cox, and W.Zwaenepoel. Quantifying the performance differences between pvm and threadmarks. In *Journal of Parallel and Distributed Computing*, June 1997.

[8] http://www.giganet.com. Giganet.

[9] http://www.viarch.org. Vi architecture.

[10] D. Jiang, B. O'Kelly, X. Yu, S. Kumar, A.Bilas, and JP. Singh. Application scaling under shared virtual memory on a cluster of SMPs. In *ICS99*, June 1999.

[11] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[12] Dongming Jiang and Jaswinder Pal Singh. Does application performance scale on modern cache-coherent multiprocessors: a case study of a 128-processsor sgi origin2000. In *The 26th International Symposium on Computer Architecture*, May 1999.

[13] S. Karlsson and M. Brorsson. A comparative charaterization of communication patterns in applications using mpi and shared memory on the ibm sp2. In *Network-based Parallel Computing, CANPC98*, pages 189–201, 1998.

[14] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[15] Daniel Scales, Kourosh Gharachorloo, and Chandramohan Thekkath. Shasta: A low-overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[16] Hongzhang Shan and Jawinder Pal Singh. Tree-building on a range of shared address space multiprocessors: algorithm and application performance. In *International Parallel Processing Symposium*, April 1998.

[17] Hongzhang Shan and Jawinder Pal Singh. Comparison of message passing, SHMEM and cache-coherent shared address space programming models on the SGI Origin 2000. In *International Conference on Supercomputing*, June 1999.

[18] Hongzhang Shan and Jawinder Pal Singh. Parallel sorting on cache-coherent DSM multiprocessors. In *Supercomputing*, November 1999.

[19] Hongzhang Shan, Jawinder Pal Singh, Leonid Oliker, and Rupak Biswas. A comparison of three programming models for adaptive applications on the origin 2000. In *Supercomputing*, November 2000.

[20] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: performance and architectural implications. *IEEE Computer*, 27(6), june 1994.

[21] S. Sistare, R. Vaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Supercomputing*, November 1999.

[22] James R. Larus Steven K. Reinhardt and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.