Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

Title
Message passing vs. shared address space on a cluster of SMPs

Permalink
https://escholarship.org/uc/item/76p9b40g

Authors
Shan, Hongzhang
Singh, Jaswinder Pal
Oliker, Leonid
et al.

Publication Date
2001-01-22

eScholarship.org
Powered by the California Digital Library
University of California

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

Hongzhang Shan, Jaswinder Pal Singh
Dept. of Computer Science
Princeton University
Princeton, NJ 08544
{shz,jps}@cs.princeton.edu

Leonid Oliker
NERSC Center
Lawrence Berkeley Lab.
Berkeley, CA 94720
loliker@lbl.gov

Rupak Biswas
NAS Systems Division
NASA Ames Research Ctr.
Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Abstract

The emergence of scalable computer architectures using clusters of PCs (or PC-SMPs) with commodity networking has made them attractive platforms for high-end scientific computing. Currently, message passing (MP) and shared address space (SAS) are the two leading programming paradigms for these systems. MP has been standardized with MPI, and is the most common and mature parallel programming approach. However, MP 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 scalable performance 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.

1. Introduction

The emergence of scalable computer architectures using clusters of PCs (or PC-SMPs) with commodity networking has made them attractive platforms for high-end scientific computing. Currently, message passing (MP) and shared address space (SAS) are the two leading programming paradigms for these systems. MP has been standardized with MPI, and is the most common and mature parallel programming approach. It provides both functional and performance portability. However, MP code development can be extremely difficult, especially for irregularly structured computations. A coherent SAS has been shown to be very effective at moderate scales 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 MP, especially for complex, irregular applications that are becoming increasingly routine as multiprocessing matures. This ease of programming can often be translated directly into performance gains [18, 19]. Even as hardware-coherent machines replace traditional distributed-memory systems at the high end, clusters of commodity PCs and PC-SMPs have become popular for scalable computing. On these, the MP programming model is dominant while the SAS model is unproven since it is implemented in software. Thus, given the ease of SAS programming, it is important to understand its performance tradeoffs with MP on commodity cluster platforms.

Approaches to support SAS in software across clusters differ 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 [13, 14] for access control or commodity-oriented hardware support [12] with the protocol implemented in software. Page-grained software coherence takes advantage of the virtual memory management facilities to provide replication and coherence at page granularity [10]. To alleviate false sharing and fragmentation problems, a relaxed consistency model is used to buffer coherence actions. Lu et al. [11] compared the performance of 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 SP2. They found that TreadMarks generally performs slightly worse. Karlsson and Brorsson [9] compared the characteristics of communication patterns in MP and page-based software shared-memory programs, using MPI and TreadMarks running on an SP2. They found that the fraction of small messages in the TreadMarks executions lead to poor performance. However, the platforms both these groups used were of much lower performance, smaller scale, and not SMP based. In

addition, the protocols used for these experiments were not very efficient. Recently, both the communication network and protocols for shared virtual memory (SVM) have made great progress. Some GB/s networks have been put into use. A new SVM protocol, called GeNIMA, for page-grained SAS on clusters 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 comparable behavior [4, 7]. Thus, a study comparing the performance of using GeNIMA against the dominant way of programming for clusters today, namely MPI, becomes necessary and important.

In this paper, we compare the performance of MP and SAS programming models using the best implementations available to us (MPI/Pro from MPI Software Technology, Inc., for MPI, and the GeNIMA SVM protocol for SAS) on a cluster of eight, 4-way SMPs (for a total of 32 processors). Our application suite includes codes that scale well on tightly-coupled machines, as well as codes that are challenging to obtain scalable performance due to their high communication-to-computation ratios and complex communication patterns. Our results show that if very high performance is the goal, the difficulty of MP programming appears to be necessary for commodity SMP clusters of today. On the other hand, if ease of programming is important, then SVM provides it at roughly a factor-of-two deterioration in performance for many applications, and less for others. This may be considered encouraging for SVM, given the 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.

The remainder of this paper is organized as follows. Section 2 describes our PC cluster platform, and the implementation of the two programming models. The benchmark applications are briefly described in Section 3, as are the modifications that were made to improve cluster performance. Performance results are presented and critically analyzed in Section 4. Finally, Section 5 summarizes our key conclusions.

2. Platform and Programming Models

The platform used for this study is a cluster of eight 4-way 200 MHz Pentium Pro SMPs. Each of the 32 processors has separate 8 KB data and instruction L1 caches, and a unified 4-way set-associative 512 KB L2 cache. Each of the eight nodes, running Windows NT 4.0, has 512 MB main memory, and are connected together either by Myrinet [5] or Giganet [1]. The SAS and MP programming models are built in software 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 (SAS) for clustered architectures, both at page and at finer fixed granularities through code instrumentation. Among the most popular ways to support a coherent SAS in software on clusters is page-based shared virtual memory (SVM). SVM provides replication and coherence at the page granularity by taking advantage of virtual memory management facilities. To alleviate problems with false sharing and fragmentation, SVM uses a relaxed memory consistency model to buffer coherence actions such as invalidations or updates, and postpones 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, thereby reducing the impact of write-write false sharing and making the page consistent only when needed by applying `diffs` and `write` notices. Many distinct 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 shown good performance on moderate-scale systems for a fairly wide range of applications, achieving at least half the parallel efficiency of a high-end hardware-coherent system [4, 7]. It uses general-purpose network interface support to significantly improve protocol overheads. Thus, we select GeNIMA as our protocol for the SAS programming model. It is built on top of VMMC, a high-performance, user-level virtual memory mapped communication library [6]. 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. A single 16-way Myrinet crossbar switch is used to minimize contention in the interconnect. Each network interface has a 33 MHz programmable processor and connects the node to the network with two unidirectional links of 160 MB/s peak bandwidth each. The actual node-to-network bandwidth however is constrained by the 133 MB/s PCI bus. The parallelism constructs and calls needed by the SAS programs are identical to those used in our hardware-coherent platform (SGI Origin2000) implementation [16, 17, 18], making portability trivial between these systems.

2.2. MP Programming Model

The message-passing (MP) implementation used in this work is MPI/Pro from MPI Software Technology, Inc., and is developed directly on top of Giganet networks by the VIA [3] interface. By selecting MPI/Pro instead of build-

Msg. size	4	16	64	256	1024	4096	16384
VMMC	10.9	11.2	15.1	20.0	34.2	80.1	210
VIA	10.3	10.6	12.4	14.3	23.8	65.5	231

Table 1. Communication times (in microsecs) of different message sizes (in bytes) for the VMMC and VIA interfaces.

ing 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 potentially poor implementation of the communication layer. Fortunately, as shown in Table 1, VIA and VMMC have similar communication times for a range of message sizes on our cluster platform. Giganet performs somewhat better for short messages while Myrinet has a small advantage for larger messages. There should thus be little performance difference for similar MPI implementations across these two networks. Note that the Giganet network interfaces are also connected together by a single crossbar switch.

3. Benchmark 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 (FFT, OCEAN, and LU) as well as irregularly structured applications (RADIX sort, SAMPLE sort, and N-BODY). All six codes have either high communication-to-computation ratios or complex communication patterns, making scalable performance on cluster platforms a difficult task. FFT uses a non-localized but regular all-to-all personalized communication pattern to perform a matrix transposition; i.e., every process communicates with all others, sending different data across the network. OCEAN exhibits primarily nearest-neighbor patterns, but in a multigrid formation rather than on a single grid. LU uses one-to-many non-personalized communication. RADIX uses all-to-all personalized communication, but in an irregular and scattered fashion. In contrast, the all-to-all personalized communication in SAMPLE is much more regular. Finally, N-BODY requires all-to-all all-gather communication and unpredictable send/receive patterns. All these applications have shown good parallel performance under both MPI and SAS for reasonably large data sets on hardware-supported cache-coherent platforms [16].

Most of the MPI programs were ported directly onto our cluster platform without any changes. However, OCEAN and RADIX required some modifications for better performance. In OCEAN, the matrix is now partitioned by rows instead of by blocks. This allows each processor to communicate only with its two neighbors, thus reducing the number of messages while improving the spatial locality of the communicated data. For RADIX, in the key exchange stage, each processor now sends only one message to every other processor, containing all its chunks of keys that are destined for the destination processor. The receiving processor then reorganizes the data chunks to their correct positions. On a hardware-supported cache-coherent platform, a processor would send each contiguously-destined chunk of keys directly as a separate message, so that the data could be inserted into the correct position at the destination processor immediately. However, this requires multiple messages from one processor to every other processor. While this method succeeds on systems like the Origin2000, the modified approach is better suited for cluster platforms since reducing the number of messages at the cost of increased local computations is more beneficial. To study the two-level architectural effect (intra-node and inter-node), we tested our applications by reorganizing the communication sequence (intra-node first, inter-node first, or intra-node and inter-node mixed). Interestingly, our results showed that the performance of the MPI programs is insensitive to the communication sequence.

For the SAS codes, FFT, LU, and SAMPLE were ported without any modifications. For RADIX, we used the improved version described in [17] where keys destined for the same processor are buffered together instead of exchanging them in a scattered fashion. Several changes were made to the original version of OCEAN to improve its shared-memory performance [7] on clusters. The matrix was partitioned by rows across processors instead of by 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 during the shared-tree building phase. A new tree building method, called Barnes-spatial [15], has been developed to completely eliminate the expensive synchronization operations.

These applications have been previously used to evaluate the performance of different programming models on a hardware-supported cache-coherent platform [16]. In that study, it was shown that SAS programs provide substantial ease of programming compared to MP implementations, while performance, though application-dependent, was sometimes better for SAS. The ease of programming holds true also on cluster systems, although some SAS code restructuring was required to improve performance. Nonetheless, a SAS implementation is still easier than MPI as has been argued earlier in the hardware-coherent context [8].

A comparison between MPI and SAS programmability is presented in Table 2. Notice that SAS programs require

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

Table 2. Number of essential code lines for MPI and SAS implementations of our benchmark applications.

fewer lines of essential code (excluding the initialization and debugging code, and comments) compared with MPI. In fact, as application complexity (e.g., irregularity and dynamic nature) increases, we see a bigger reduction in programming effort using SAS.

4. Performance Analysis

In this section, we compare the performance of our applications under both the MP and SAS programming paradigms. For each application, parallel speedups and detailed time breakdowns are presented. To derive the speedup numbers, we use our best sequential runtimes as the baseline. The parallel runtimes are broken up into three components: LOCAL, RMEM, and SYNC. LOCAL includes CPU computation time and CPU waiting time for local cache misses, RMEM is the CPU time spent for remote communication, while SYNC represents the synchronization overhead. Two data set sizes are chosen for each application. The first is a *baseline* data set at which the SVM begins to perform "reasonably" well [7]. Next, we use a larger data set, since increasing the problem size generally tends to improve many inherent program characteristics, such as load balance, communication-to-computation ratio, and spatial locality.

4.1. FFT

FFT has very high communication-to-computation ratio, which diminishes only logarithmically with problem size. It requires a non-localized but regular all-to-all personalized communication pattern to perform the matrix transposition, and cannot overlap the transposition and computation stages. In general, it is much more difficult to achieve high performance on the one-dimensional FFT, studied here, compared with higher-dimensional FFTs. Speedups for the SAS and MPI versions are presented in Table 3 for 1M and 4M data sets.

Neither MPI nor SAS show high scalability for our test cases. Increasing the data set size improves performance, but only slightly. This is mainly due to the pure communication of the transpose stage whose communication-to-computation ratio is not affected by problem size. In the

	1M data set		4M data set	
	P=16	P=32	P=16	P=32
SAS	3.39	3.90	3.83	5.42
MPI	5.94	9.18	5.35	10.43

Table 3. FFT speedups.

sequential case, the transposition is responsible for approximately 16% of the overall runtime. However, this percentage increases to 50% when using all 32 processors. It is inherently difficult to scale pure all-to-all communications. As the number of active processors increases, so does the contention in the network interface. Additionally, since each remote request requires access to the memory bus, increasing the number of processors has a deleterious effect on the local memory access time. This is particularly true for our commodity 4-way SMP platform which suffers from high memory bus contention when all four processors simultaneously attempt to access memory. For example, the FFT LOCAL time (which includes the memory stall time) on two processors for the 4M data set is about 6 secs. However, LOCAL drops to only about 4.8 secs when all four processors are used, compared to an ideal of 3 secs.

Observe though that the MPI implementation significantly outperforms SAS. To better understand the performance difference, Figure 1 presents the time breakdown for the 4M data set running on 32 processors.

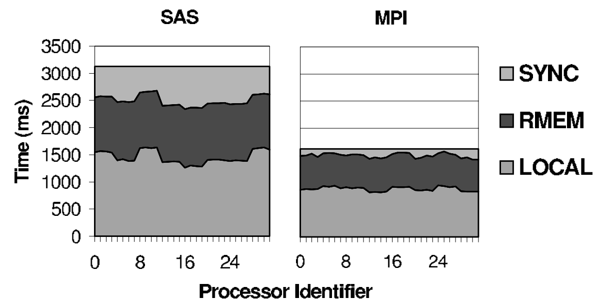

Figure 1. FFT time breakdown for SAS and MPI on 32 processors for 4M data set.

We find that all the three time components (LOCAL, RMEM, and SYNC) are much higher in SAS than in MPI. In order to maintain page coherence, a high protocol overhead is introduced in SAS programs, including: computing `diffs`, creating timestamps, generating `write` notices, and performing garbage collection. This protocol overhead dramatically increases compute time while degrading local cache performance, thus causing a higher LOCAL time. In addition, the `diffs` generated for maintaining coherence immediately cause pages to be propagated to their home processors, thereby increasing network traffic and possi-

bly causing more memory contention. Finally, at synchronization points, handling the protocol requirements causes a significant dilation of the synchronization interval, including the expensive invalidation of necessary pages. None of these protocol overheads exist in the MPI implementation. MPI does have the additional cost of packing and unpacking data for efficient communication; however, this overhead is incurred locally on the processors and is insignificant compared to the protocol costs associated with SAS. One possible way to improve SAS performance would be to restructure the code so that the data structures more closely resemble the MPI implementation. For example, instead of allocating the matrix as a shared data structure, each sub-matrix that is transposed onto a different processor could be allocated separately. However, this would dramatically increase the complexity of the SAS implementation, and thus sacrifice the programming ease of the shared-memory paradigm.

4.2. OCEAN

OCEAN exhibits a commonly used nearest-neighbor pattern, but in a multigrid rather than a single-grid formation. Parallel speedups are presented in Table 4. The scalability of the commodity SMP platform is relatively low, compared with previously obtained results on the hardware-supported cache-coherent architecture of the Origin2000 [16]. Although, the communication-to-computation ratio of OCEAN is high for smaller data sets, it quickly improves with larger problem sizes. This is especially true for the MPI version as shown in Table 4. Notice that SAS achieves superlinear speedup between 16 and 32 processors on the smaller data set. This occurs partly because as the number of processors increases, a larger fraction of the problem fits in cache.

	258×258 grid		514×514 grid	
	P=16	P=32	P=16	P=32
SAS	2.17	5.96	5.44	6.49
MPI	4.97	8.03	7.45	15.20

Table 4. OCEAN speedups.

The SAS implementation suffers from expensive synchronization overheads, as shown in Figure 2. After each nearest-neighbor communication, a barrier synchronization is required in SAS to maintain coherence. Further analysis of the synchronization costs show that about 50% of the synchronization overhead is spent waiting, while the remainder is for protocol processing [4]. Thus, the synchronization cost can be improved either by reducing protocol overhead or by increasing the data set size. Unfortunately, there is not enough computational work between the synchronization points for the 514×514 problem size, especially because this grid is further coarsened into smaller

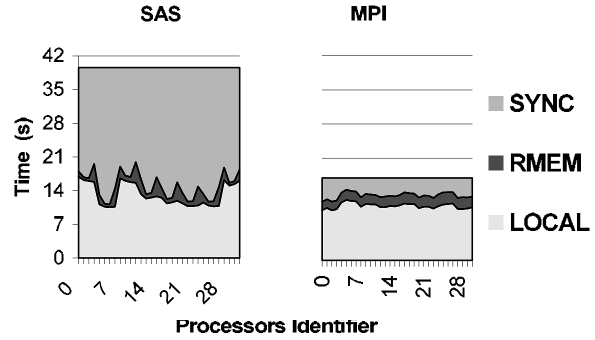

Figure 2. OCEAN time breakdown for SAS and MPI on 32 processors for 514x514 grid size.

subgrids during program execution. Moreover, OCEAN has a large memory requirement due to its use of more than 20 large data arrays, required for the multi-grid code. Thus, we are prevented from running larger data sets due to memory constraints. The synchronization within the MPI program is dramatically lower since it is implicitly implemented using send/receive pairs.

4.3. LU

The communication requirements of LU are relatively small compared to our other benchmark codes, and thus we expect better performance for this application. This is confirmed by the results shown in Table 5. LU uses one-to-many non-personalized communication where the pivot block and the pivot row blocks are each communicated to $\sqrt{P}$ processors. From the time breakdown in Figure 3, it is obvious that most of the overhead is in the LOCAL time. The LU performance could be further improved by reducing the synchronization cost caused by the load imbalance associated with the CPU wait time.

	4096×4096 matrix		6144×6144 matrix	
	P=16	P=32	P=16	P=32
SAS	12.48	22.98	11.79	21.78
MPI	13.15	23.04	12.31	22.43

Table 5. LU speedups.

Notice that for LU, the performance of the SAS and MPI implementations are very close in both speedup and time breakdown characteristics. The protocol overhead of running the SAS version constitutes only a small fraction of the overall runtime. Unlike our FFT example, the LU matrix is organized in a four-dimensional array such that blocks assigned to each processor are allocated locally and contiguously. Thus, each processor modifies only its own blocks,

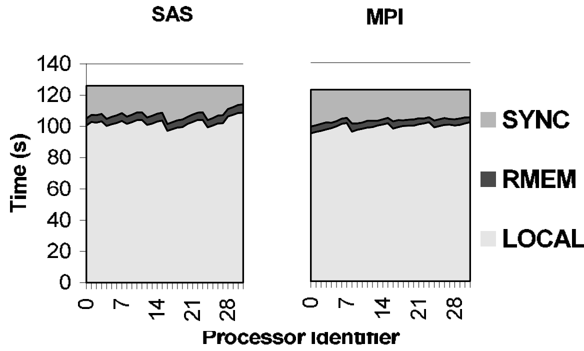

Figure 3. LU time breakdown for SAS and MPI on 32 processors for 6144x6144 matrix size.

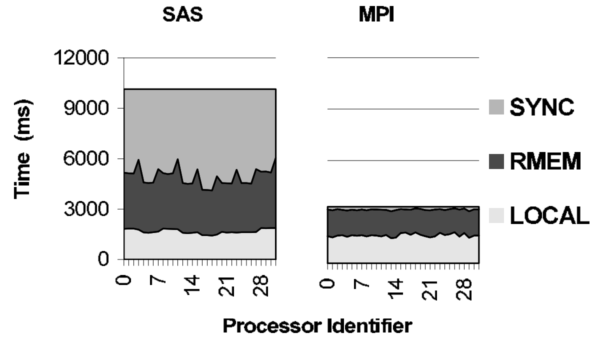

Figure 4. RADIX time breakdown for SAS and MPI on 32 processors for 32M integers.

and the modifications are immediately applied to local data pages. As a result, no `diffs` generation and propagation are required, greatly reducing the protocol overhead. These performance results show that for applications with relatively low communication requirements, it is possible to achieve high scalability on commodity clusters using both MPI and SAS programming approaches.

4.4. RADIX

Unlike the previous three regularly structured codes (FFT, OCEAN, and LU), we now investigate three applications with irregular characteristics: RADIX, SAMPLE, and N-BODY. The RADIX sort benchmark requires all-to-all personalized communication, but in an irregular and scattered fashion. It also has a high communication-to-computation ratio that is independent of problem size and the number of processors. This application has high memory bandwidth requirements which can exceed the capacity of current SMP platforms; thus, high contention is caused on the memory bus when all four processors of a node are in use. The "aggregate" LOCAL time across processors is much higher than in the uniprocessor case, which leads to the poor performance shown in Table 6. However, MPI significantly outperforms the SAS implementation, since the latter has higher RMEM and SYNC times as shown in Figure 4. These costs are due to the expensive protocol overheads of performing all-to-all communications, for similar reasons as already discussed for FFT.

	4M integers		32M integers	
	P=16	P=32	P=16	P=32
SAS	1.33	1.66	1.86	2.70
MPI	3.78	5.67	4.16	7.78

Table 6. RADIX speedups.

Note that choice of the proper implementation strategy for the MPI all-to-all communication is platform dependent. On the commodity cluster, each processor sends only one large message to all the other processors. The message contains all the data chunks required by the destination processor, which in turn reorganizes the separate blocks of data into their correct positions. This is similar to the algorithm used in the IS NAS Parallel Benchmark [2]. However, on the hardware-supported cache-coherent Origin2000 platform, each processor sends the contiguous chunks of data directly to their destination processors in separate messages. Thus, unlike the cluster, each processor sends multiple messages to all the other processors in the system. The difference in these two approaches stems from the relatively high latency and low bandwidth of the cluster, where it is more efficient to send fewer messages in exchange for increased computational requirements of assembling the scattered data chunks.

4.5. SAMPLE

SAMPLE sorting also requires an irregular personalized all-to-all communication pattern; however, it contains more regularity than the RADIX algorithm. Speedups for SAMPLE are presented in Table 7, and compare favorably to the RADIX performance. Note that the same sequential time is used as a baseline when computing the speedups for both RADIX and SAMPLE. In SAMPLE, each processor first performs a local sort on its partitioned data using the radix sorting algorithm. Next, an all-to-all communication is used to exchange keys, and a second local sort is performed on the newly-received data. However, in the sequential case, only a single local sort is required. Thus, it is reasonable to expect ideal SAMPLE performance to achieve only a 50% parallel efficiency.

Figure 5 presents the time breakdown of SAMPLE for the larger data set on 32 processors. Observe that the

	4M integers		32M integers	
	P=16	P=32	P=16	P=32
SAS	2.10	2.13	4.97	4.89
MPI	4.89	8.60	5.73	11.07

Table 7. SAMPLE speedups.

	32K particles		128K particles	
	P=16	P=32	P=16	P=32
SAS	6.05	9.31	10.64	14.30
MPI	8.15	14.10	14.05	26.94

Table 8. N-BODY speedups.

RMEM and SYNC times are significantly smaller than those of RADIX in Figure 4, for both MPI and SAS. As a result, the SAMPLE algorithm outperforms RADIX. Note that the LOCAL time in SAMPLE sort is only slightly higher than RADIX, even though much more computation is performed during SAMPLE. This indicates that contention on the memory bus for RADIX sorting is higher than that for SAMPLE due to the higher irregularity of its memory access patterns.

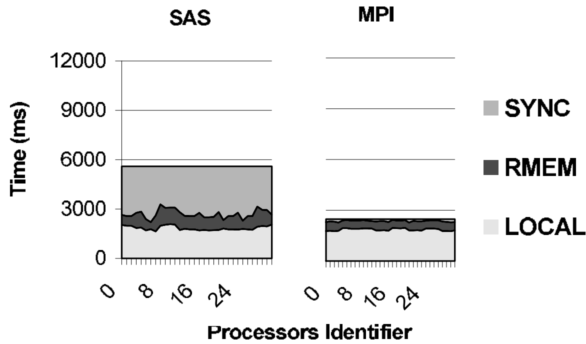

Figure 5. SAMPLE time breakdown for SAS and MPI on 32 processors for 32M integers.

4.6. N-BODY

Finally, we examine the performance of the N-BODY simulation. Table 8 shows that MPI again outperforms SAS, especially for the larger data set. For 128K particles on 32 processors, MPI achieves almost twice the performance of SAS. The time breakdown for this data set on 32 processors is shown in Figure 6. The SAS implementation has higher SYNC and RMEM times compared to MPI, but the synchronization overhead clearly dominates the overall runtime. This is because at each synchronization point, many `diffs` and `write` notices are processed by the coherence protocol. In addition, a large number of shared pages are invalidated. Further analysis shows that 82% of the barrier time is spent on protocol handling. This expensive synchronization overhead is incurred in all of our applications except LU, causing a degradation of SAS performance. Future research on the SVM coherence protocol should focus on reducing this synchronization cost. Possible approaches may include applying the `diffs` before the synchronization points, moving the shared-page invalidation operation out of synchronization points, and increasing the protocol hardware support.

Unlike our other five applications, the MPI version of N-BODY has a higher LOCAL time than the SAS counterpart. This is due to the use of different high-level algorithms for each programming model. In the SAS implementation, each processor builds one part of a globally shared tree; while in MPI, a locally essential tree is created on each processor. Building the locally essential tree across distributed memories is much more complex than using a shared address space to build a single globally addressable tree. Therefore, there is a higher computational tree-building cost in the MPI implementation [18]. However, with large data sets, the tree-building phase becomes computationally insignificant compared to the other phases of the N-body simulation, notably the force calculation.

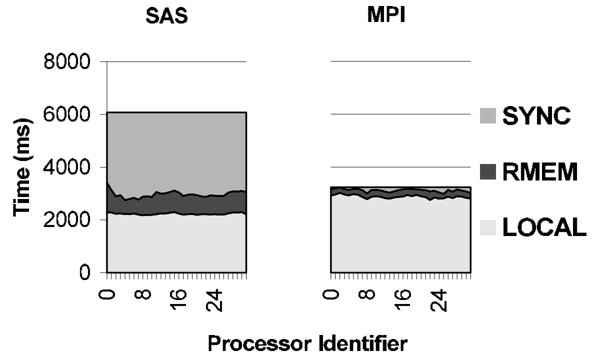

Figure 6. N-BODY time breakdown for SAS and MPI on 32 processors for 128K particles.

5. Conclusions

In this paper, we studied the performance of and programming effort required for six applications using the message passing (MP) and shared address space (SAS) programming models on a 32-CPU PC-SMP cluster. To create a fair comparison between the two programming methodologies, we used the best known implementations of the underlying communication libraries. The MP version used

MPI/Pro which is implemented directly on top of Giganet by the VIA interface. The SAS implementation uses the GeNIMA SVM protocol over the VMMC communication library, which runs on Myrinet. Experiments showed that VIA and VMMC have similar communication characteristics for a range of message sizes on our cluster platform. Our application suite consisted of codes that typically do not exhibit scalable performance under shared-memory programming due to their high communication-to-computation ratios and complex communication patterns.

Overall, SAS provides substantial ease of programming, especially for the more complex applications which are irregular or dynamic in nature. However, unlike in a previous study on hardware-coherent machines where the SAS implementations were also performance-competitive with MPI, and despite all the research in SVM protocols and communication libraries in the last several years: SAS achieved only about half the parallel efficiency of MPI for most of our applications. The LU benchmark was an exception, in which the SAS implementation on the PC cluster achieved very similar performance compared to the MPI version. The higher runtimes of the SAS versions were due to the excessive cost of the SVM 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, the difficulty of MP programming appears to be necessary for commodity SMP clusters of today. On the other hand, if ease of programming is important, then SVM provides it at roughly a factor-of-two deterioration in performance for many applications, and less for others. This may be considered encouraging for SVM, given the 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.

Acknowledgements

The work of the first two authors is supported by NSF under grant ESS-9806751. The second author is also supported by PECASE and a Sloan Research Fellowship. The work of the third author is supported by the U.S. Department of Energy under contract DE-AC03-76SF00098.

References

[1] www.giganet.com.

[2] www.nas.nasa.gov/Software/NPB.

[3] www.viarch.org.

[4] A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proc. 26th Intl. Symp. on Computer Architecture*, pages 282–293, 1999.

[5] N. J. Boden, D. Cohen, R. E. Flederman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Proc. 5th Hot Interconnects Symp.*, 1997.

[7] D. Jiang, B. O'Kelly, X. Yu, S. Kumar, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of SMPs. In *Proc. 13th Intl. Conf. on Supercomputing*, 1999.

[8] D. Jiang and J. P. Singh. Scaling application performance on cache-coherent multiprocessors. In *Proc. 26th Intl. Symp. on Computer Architecture*, pages 305–316, 1999.

[9] S. Karlsson and M. Brorsson. A comparative characterization of communication patterns in applications using mpi and shared memory on an IBM SP2. In *Proc. 2nd Intl. Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 189–201, 1998.

[10] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[11] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *J. of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[12] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. 21st Intl. Symp. on Computer Architecture*, pages 325–336, 1994.

[13] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, 1996.

[14] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[15] H. Shan and J. P. Singh. Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance. In *Proc. 12th Intl. Parallel Processing Symp.*, 1998.

[16] H. Shan and J. P. Singh. A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI Origin2000. In *Proc. 13th Intl. Conf. on Supercomputing*, pages 329–338, 1999.

[17] H. Shan and J. P. Singh. Parallel sorting on cache-coherent DSM multiprocessors. In *Proc. Supercomputing99*, 1999.

[18] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. A comparison of three programming models for adaptive applications on the Origin2000. In *Proc. Supercomputing2000*, 2000.

[19] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(6), June 1994.