University of Alberta

Shared Memory Computing on SP2: JIAJIA Approach

by

M. Rasit Eskicioglu and T. Anthony Marsland

Technical Report TR 98-10
July 1998

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Submitted to the 1998 IBM Centre for Advanced Studies Conference (CASCON'98), November 1998.

Shared Memory Computing on SP2: JIAJIA Approach[†]

M. Rasit Eskicioglu and T. Anthony Marsland
Department of Computing Science
University of Alberta
Edmonton, AB T6G 2H1
{rasit,tony}@cs.ualberta.ca

Abstract

Distributed shared memory (DSM) is a useful abstraction both for deploying networks of workstations as a parallel multicomputer and for increasing the usability of non-uniform memory access multicomputers. It provides an alternative programming model for distributed memory computers. In this paper, we present empirical evaluation of JIAJIA, a software DSM system, on an IBM SP2 cluster. We also discuss the performance of a suite of six widely different applications running under this software and compare them with CVM, another software DSM system. We show that these applications achieve moderate to good speedups and argue that shared memory computing is an attractive alternative to message passing on high performance computer systems, such as SP2 clusters.

1 Introduction

Shared memory is a simple programming model to develop parallel applications. Rapidly improving performance of uniprocessors and increasing availability of high-speed interconnection media make networks of workstations (NOWs), as well as distributed memory computers, an attractive alternative to shared memory multiprocessors. The abstraction of shared memory on a distributed computing system, referred to as "*Distributed Shared Memory*", provides an illusion of a large shared memory not available physically. This global memory spans the private memories of individual computers extending across machine boundaries. It allows processes executing on different computers interconnected by a network to share memory by hiding the physical location(s) of data and making the memory *location transparent* to the entire system. During the execution of an application, the runtime system detects all (shared) memory accesses and fetches the data from remote memories, if necessary. The DSM approach provides both ease of programming of shared memory multiprocessors and the scalability of distributed memory computers. We argue that software DSM systems can be efficiently used with high performance computer systems for a variety of applications with similar performance achievements to those of shared memory multiprocessors.

[†]This work is supported in part by NSERC grant OGP7902.

Software DSM can be build on stock hardware and generally only requires some common features (such as virtual memory support) found in today's modern processors. For this and other reasons, it has been an active research area over the last decade, and systems like Munin [3], Midway [2], TreadMarks [8], CVM [6], and JIAJIA [4] were designed and implemented.

This paper describes the port of the JIAJIA software DSM system to an IBM SP2 cluster. It also evaluates the impact of high performance, wide bandwidth interconnection medium, such as the SP2's switch, on the performance of a suite of six widely different applications. The results of our experiments show that programming with an "all-software" shared memory approach such as JIAJIA is an attractive alternative to message passing on high performance computer systems. In the following sections, we give an overview of the JIAJIA software DSM system, emphasizing its distinguishing characteristics, summarize the applications in our test suite, and analyze the results of our experiments. We also compare the performance of the same applications with CVM [6], another software DSM system.

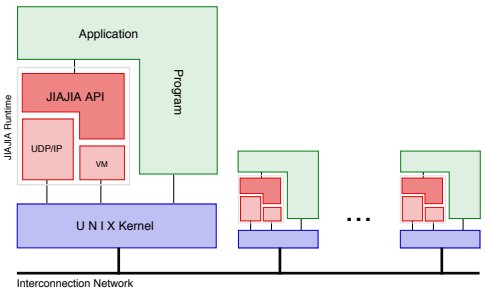


Figure 1: JIAJIA Runtime Architecture

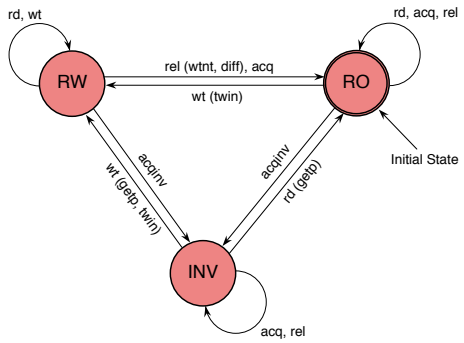
2 JIAJIA DSM System

JIAJIA is a software DSM system designed to run at user level on a network of stand-alone Sun Sparc workstations. As Figure 1 shows, it is built onto UNIX as a runtime library and uses standard libraries for remote program invocation, interprocess communication, and memory management. JIAJIA implements scope consistency [5] with a lock-based protocol and uses a write-invalidation scheme to handle dirty data. Also, its coherence protocol allows multiple writers to alleviate false sharing.

2.1 Coherence Protocol

Based on the observation that the overhead of a complex software DSM system may easily offset its benefits, the coherence protocol of JIAJIA, as summarized in Figure 2, is designed to be as simple as possible.

The shared pages in an application can either be "local" or "cached" on a given processor. In the former case, the processor is the *home* of the page. During the execution, these pages can be in one of three states: `Invalid(INV)`, `Read-Only(RO)`, and `Read-Write(RW)`. The initial state of the pages at their home processors is `RO`. Since multiple writers are allowed, a page may be in different states after several processors cache it.



Notes

rd, wt :	read, write
acq, rel :	acquire, release
acqinv :	invalidate the page on acquire
getp :	get the page from its home
wtnt :	send write-notices to lock
diffs :	send page diffs to home(s)
twin :	create a twin of the page

Figure 2: JIAJIA's Coherence Protocol

Ordinary read and write accesses to a `RW` page, or read access to a `RO` page, or acquire and release on an `INV` or a `RO` page do not cause any change in the page's state. Like the shared pages, each lock has a home processor which is assigned in a round-robin fashion during system initialization.

On a release, the processor generates "diffs" (run-length encoding of the changes made to a page) for all modified pages and eagerly sends them to their respective homes. Also, the processor sends a release request to the lock's home processor along with the write-notices (basically, a list of modified pages) for the associated critical section. Similarly, the acquiring processor sends a request to the lock's owner and waits until it receives a lock-grant message. Multiple acquire requests for a lock are queued at the lock's home processor. When the lock becomes (or is) available, a lock-grant message is sent to the first processor in the queue, piggy-backed with the current existing write-notices. After receiving the lock-grant message, the acquiring processor invalidates the pages listed in the write-notices and continues with its normal execution.

On reaching at a barrier, processors send write-notices along with diffs to the homes of the modified pages. Home processors, in turn, apply the diffs to the original copy of the pages. Thus, each processor resumes execution with an up-to-date view of the shared memory after a barrier.

In summary, the protocol propagates all the modifications (as diffs) to the home processor of a page on a release and to the next processor on the following acquire. This approach keeps the diffs only for a short period of time, hence reducing (local) diff keeping overhead.

Unlike other DSM systems, JIAJIA does not keep a separate global directory structure, instead, only a lock structure keeps the necessary information, such as ownership, for the relevant pages. This approach further reduces the overall space overhead of the system.

Currently, JIAJIA provides two synchronization operations (though, others can easily be added): *lock–unlock* and *barrier*. Either one of these operations can be used in an

application to control a critical section. A barrier can be viewed as a combination of a lock–unlock pair, but in reverse order: arriving at a barrier *exits* from the "previous" critical section and leaving a barrier *enters* the "next" (new) critical section. Since two barriers are needed to enclose a critical section, the start of an application is considered an implicit entry to the first critical section.

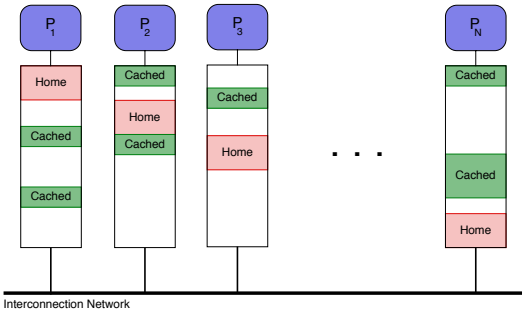


Figure 3: Shared Memory Organization

2.2 Shared Memory Organization

As Figure 3 shows, JIAJIA organizes the shared memory in a different and unconventional way. The global shared memory is distributed across the processors. Each processor acts as the home of a portion of the shared memory. Users can specify home size of each processor in a configuration file and hence control initial distribution of shared data. A page is accessed ordinarily when referenced by its home processor. A remote page, on the other hand, is first fetched from its home processor and cached locally for subsequent accesses. A page is always kept at the same user space address, in other words, the logical address of a page is identical on all processors, whether it is a home page or has been cached by the processor. This approach eliminates any address translation upon a remote access and provides a uniform view of the shared memory across the processors. Furthermore, each processor uses a local page table to keep information only about its "cached" pages. It contains the address, current state and a twin (if in `RW` state) for each cached page.

With the above memory organization, JIAJIA is able to support shared memory that is much larger than the physical memory of any single processor in the system. Hence, the total size of the shared memory is not limited by the physical memory of a single processor, but only by the virtual memory settings (e.g., maximum allowable user-mappable address range) of the underlying hardware and operating system.

2.3 Programming Interface

JIAJIA implements the SPMD programming model, in which each processor runs the same program on different parts of the shared data. It provides functions for system initialization, shared memory allocation, and synchronization with the following programming interface:

- `jia_init(argc,argv)`—initializes JIAJIA DSM runtime system. It must be called from every shared memory application.
- `jia_alloc(int size)`—allocates shared memory. The parameter `size` indicates the number of bytes allocated.
- `jia_lock(int lockid)`—acquires a global lock specified by `lockid`.
- `jia_unlock(int lockid)`—releases a global lock. `jia_lock()` and `jia_unlock()` should appear in pairs for obvious reasons.
- `jia_barrier()`—performs a global barrier by preventing any process from proceeding until all processes reach the barrier.
- `jia_wait()`—similar to `jia_barrier()` except that `jia_wait()` does not enforce any coherence operations across processors.
- `jia_clock()`—returns elapsed time since the start of application in seconds.
- `jia_error(char *str)`—prints out the error string `str` and terminates the application.
- `jia_exit()`—prints statistics (optional) and terminates the application.

Additionally, two variables, `jiapid` and `jiahosts`, specify the host identification number and the total number of hosts of a parallel program, respectively. This simple interface is defined in a header file, which must be included by the application.

The SP2 port of JIAJIA also fully supports the `M4` macros commonly used in many shared memory applications.

3 Performance

3.1 Experimental Environment

We tested JIAJIA software DSM system on an IBM SP2 cluster at the Center for High Performance Computing at the University of Utah. The cluster consists of 64 nodes, with slightly different characteristics. The results reported here were collected on 16 identical thin nodes, each equipped with 120 MHz POWER2 Superchip processor and 128 MB physical memory. The nodes are interconnected with a high performance multi-stage crossbar switch which provides a minimum of four simultaneous paths (with a bandwidth of 80 megabits each) between any pair of nodes. The nodes are also connected to the outside world by both Ethernet and FDDI links. A full version of the AIX 4.1.5 operating system runs on each node. Our experiments ran on the 1, 2, 4, 8, and 16 node configurations in dedicated mode. Since there were no other user processes, the applications used the full capacity of each node.

Our test suite includes six applications, namely, Water, LU, Barnes, EP, TSP, and Matmul, covering a broad range of problem domains with varying behaviors. Water, LU, and Barnes are from the SPLASH parallel benchmark suite [10]. The applications in this suite were developed for use in the design of shared-memory multiprocessors, as well as in the study of centralized and distributed share memory multiprocessors. Consequently, these applications are tailored for hardware (sequential) cache-coherent systems with cache line

granularity. EP is from the NAS Parallel Benchmarks [1]. NAS benchmarks are developed for evaluating the performance of highly parallel supercomputers. TSP is developed at Rice University in conjunction with their commercial TreadMarks software DSM system [8]. Our last application Matmul is a simple matrix multiplication program. Table 1 lists relevant characteristics of the applications in the test suite. Note that for simplicity JIAJIA allocates a new page for each `jia_alloc()` call, thus the page count in the last column of the table does not necessarily reflect the actual size of the shared data, except for LU and Matmul, which share large and contiguous amounts of data. Below, we briefly summarize

Appl.	**Sync.**	**Dataset** *Sm/Md/Lg*	**Sh Mem** *(4K pgs)*
Water	B, L	343 mols	27
		1,000 mols	71
		1,728 mols	121
LU	B	$1K \times 1K$	2,059
		$2K \times 2K$	8,205
		$3K \times 3K$	18,447
Barnes	B	8,192 bodies	498
		16,384 bodies	994
		32,768 bodies	1,986
EP	B	2^{24} numbers	1
		2^{26} numbers	1
		2^{28} numbers	1
TSP	L	18 cities	197
		20 cities	197
		19 cities	197
Matmul	B	$1K \times 1K$	3,075
		$2K \times 2K$	12,294
		$3K \times 3K$	27,657
B=barriers, L=locks			

Table 1: Characteristics of Applications

the applications used in this work.

3.2 Applications

Water is an N-body simulation program that evaluates forces and potentials in a system of water molecules in the liquid state using an $O(n^2)$ brute force method with a cutoff radius. Water does a step by step simulation of the molecular states. Both intra- and inter-molecular potentials are computed in each step. The most computation- and communication-intensive part of the program is the inter-molecular force computation phase, where each processor computes and updates the forces between each of its molecules and each of the $n/2$ following molecules in a wrap-around fashion. We used the slightly revised TreadMarks version [9] of Water in our experiments.

Appl.	Size	SEQ	1-proc	2-proc	4-proc	8-proc	16-proc
Water	343 mols	*42.96*	43.02	30.98	15.93	14.74	26.07
	1,000 mols	*370.41*	369.93	195.13	102.52	60.91	55.09
	1,728 mols	*1114.74*	1115.76	575.32	294.60	158.53	110.99
LU	$1K \times 1K$	*44.16*	44.19	25.93	14.66	9.11	6.23
	$2K \times 2K$	*353.64*	358.58	195.27	102.91	59.59	36.58
	$3K \times 3K$	*1193.90*	1192.55	647.45	333.51	184.66	103.92
Barnes	8,192 particles	*51.24*	51.60	28.36	19.65	19.62	29.33
	16,384 particles	*115.89*	118.86	65.04	43.53	40.20	59.78
	32,768 particles	*255.06*	270.56	148.28	98.15	86.15	120.61
EP	2^{24} numbers	*74.69*	74.72	37.58	19.27	9.37	4.37
	2^{26} numbers	*300.41*	300.46	151.24	75.05	37.51	19.27
	2^{28} numbers	*1203.67*	1203.83	607.34	301.15	150.66	75.62
TSP	18 cities	*42.82*	42.80	22.74	12.44	7.34	4.92
	20 cities	*277.20*	277.07	149.61	80.22	47.03	34.08
	19 cities	*435.17*	434.92	226.41	118.75	59.38	33.56
Matmul	$1K \times 1K$	*45.68*	48.58	24.63	13.61	8.19	11.31
	$2K \times 2K$	*367.13*	447.87	193.50	104.00	58.65	47.37
	$3K \times 3K$	*1251.18*	1748.95	773.78	351.35	190.35	120.02

Table 2: Performance of Applications with JIAJIA (in seconds)

LU is a matrix decomposition kernel that factors a dense matrix into the product of lower and upper triangular matrices. The dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality of sub-matrix elements. This version of the kernel (LU-Contiguous) factors the matrix as an array of blocks, allowing blocks to be allocated contiguously and entirely at the processors that own them, even though these blocks are not contiguous in the original array. The algorithm factors the matrix in several steps separated by barriers.

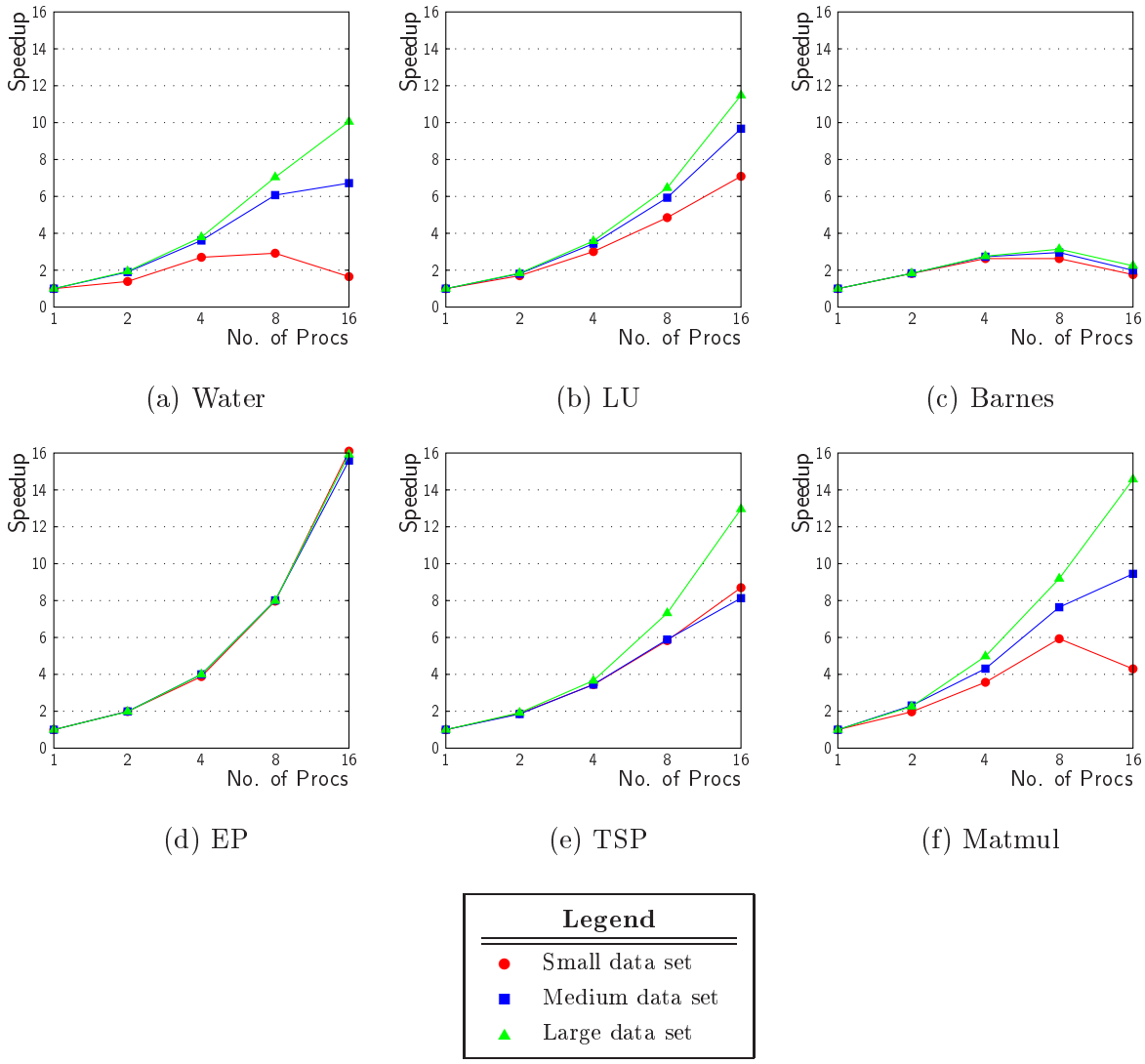


Figure 4: Application Speedups with JIAJIA

Barnes is the implementation of Barnes-Hut hierarchical N-body algorithm which simulates the interaction of a system of particles in 3-dimensions over a number of time steps. The particles are represented as the leaves of an octree. Each processor computes the forces for the particles it holds by partially traversing the octree, which is rebuilt at each time

step, based on the current particle positions. At the end of each time step, the properties of the particles are updated. This version was slightly modified by the Rice University TreadMarks group. The single lock used to protect global cell structures was removed and local structures were defined.

EP (embarrassingly parallel) kernel benchmark accumulates two-dimensional statistics from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well-suited for parallel computation. EP requires almost no communication, thus in some sense it provides an estimate of the upper achievable limit for floating-point performance on a particular system.

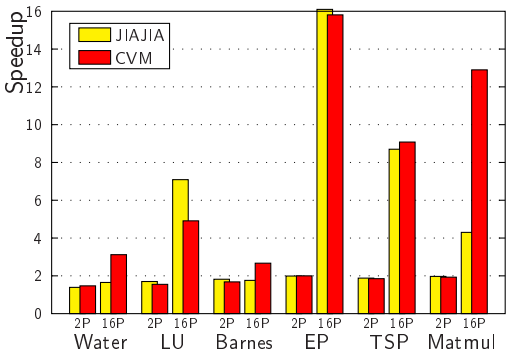
TSP solves the classical traveling salesman problem using a branch-and-bound algorithm to find the shortest path (tour). The cities are represented as the nodes of a directed graph in the program. Each processor performs the algorithm on a different branch and updates shared data. The program starts with an initial partial path and recursively permutes over the remaining nodes, updating the partial path if and when necessary, until it finds the shortest path between two cities.

Matmul is a simple implementation of the inner product algorithm used to multiply two $N \times N$ matrices. Both multiplicand matrices and the product matrix are shared by the processors. The work is divided among processors, where each processor computes the result for a certain number of rows. The partial results are then merged at a barrier after the computations.

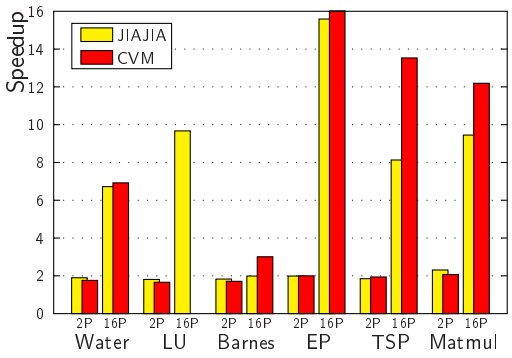
3.3 Analysis of Results

We used the GNU `gcc` compiler with `-O2` option to compile both JIAJIA and CVM versions of the test suite. The statistics collection code adds negligible overhead (less than 1%) to the execution times of applications. Table 2 summarizes the results of our experiments. Because some of the applications do not have sequential versions, we created pseudo-sequential executables by linking them with a special (`NULL`) library, in which all the API functions except `jia_alloc()`, return immediately. This dummy function calls `malloc()`, whereas the actual one uses `mmap()` to allocate (shared) memory, even on a single processor. The **SEQ** column shows the execution times of the pseudo-sequential runs. In fact, JIAJIA runtime reduces the system overhead to a bare minimum for most of the applications when the number of hosts is one. Thus, the values in the columns **SEQ** and **1-proc** are comparable, with the exception of Matmul. The slight variation between the results of sequential and 1-processor versions can be attributed to the `malloc()` system call, which is cheaper than `mmap()` on an SP2 node and most other architectures. The sequential version of Matmul performs increasingly faster with larger matrices. This anomaly is likely caused by caching, as well as paging, effects due to the large amounts of memory mapped data. Note that although one row of a $1K \times 1K$ matrix fits into a single 4096-byte page, each row of a $3K \times 3K$ matrix needs 3 pages. Thus, the overhead of this anomaly between sequential and 1-processor versions is only 6% for a $1K \times 1K$ matrix, whereas it increases to almost 40% for a $3K \times 3K$ matrix. Additionally, we ran each application with three different datasets—small, medium, and large—to see the effect of problem size on application performance. Also, we ran the same set of applications using CVM version 0.2 to allow a fair comparison with JIAJIA. In the following sections, we discuss the performance of each

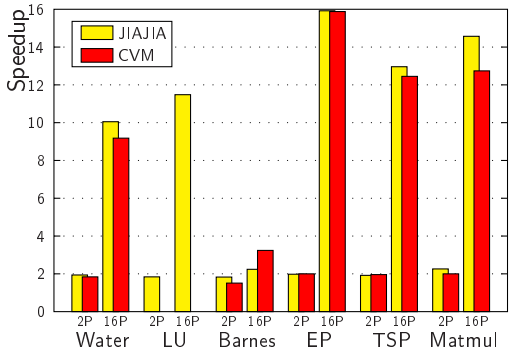
application under JIAJIA and compare these results with CVM.



(a) Small dataset



(b) Medium dataset



(c) Large dataset

Figure 5: Comparative Speedups on 2 and 16 processors

Table 2 summarizes the performance of our test suite. The detailed analysis of the applications is as follows:

Water: We simulated 343, 1000, and 1728 molecules, each for 25 steps. The amount of shared data in the revised Water code is smaller because the molecule data is split into shared and non-shared parts in this version. As shown in Figure 4 (a), with fewer molecules, the speedup is not good, in fact, the performance degrades after eight processors. The major cause of this problem, which is usually more detrimental with fewer number of molecules, is extensive fine-grain sharing, because the algorithm requires that each processor fetches modified data from half of the other processors. Moreover, the program to some degree suffers from false sharing [10]. On the other hand, with larger number of molecules, this overhead is compensated by higher computation rate, and therefore better speedups are achieved. In our test runs, we obtained speedups 1.65, 6.72, and 10.05 on 16 nodes for 343, 1000, and 1728 molecules, respectively.

LU: Figure 4 (b) shows the speedups obtained ranging from 7.09 for a $1K \times 1K$ matrix

to 11.48 for a $3K \times 3K$ matrix. The results confirmed our expectations that higher speedups and better performance can be achieved with larger problem sizes. We used a block size of 64 bytes, because after performing some additional tests, we observed that this size (*as opposed to 16 recommended by the developers of the application*) yields the best performance on our experimental platform. Based on this observation, we conclude that with page based software DSM systems, it is more important that the blocks fit into the coherence unit of the software (i.e., a physical page) rather than the cache lines of the underlying hardware.

Barnes: The performance of Barnes is not very good, despite the elimination of the critical and only lock from the code. This application suffers from not only excessive but also irregular fine-grain sharing, which causes invalidation and re-fetching of whole pages. Thus, Barnes only achieved speedups 1.76, 1.99, and 2.24 on 16 processors for 8192, 16384, and 32768 particles, respectively.

EP: This application achieves an excellent performance as expected and scales well. As shown in Figure 4 (c), the speedups are near linear (for example, 15.92 on 16 processors with 2^{28} random numbers), because the only communication among the processors, which is compensated by the high computation rate, occurs at the end of the number generation phase to accumulate the tabulated results.

TSP: This application uses only locks for synchronization while executing the branch-and-bound algorithm. There are also two barriers in the application, before and after the recursive evaluation of the tours. We tested TSP with 18, 19, and 20 cities with recursion levels (`-r` option) of 14, 14, and 15, respectively. Incidentally, the program finds the minimum tour length for 20 cities faster than for 19 cities due to the setup of the input data. The speedup for all three datasets up to four processors is near linear. However, as the number of processors increases beyond four, the larger datasets are penalized by our lock-based coherence protocol. JIAJIA transfers mostly whole pages because the accumulation of lock releases unnecessarily invalidate more pages on acquire. Figure 4 (d) shows the speedups achieved, despite the above deficiency. The reason for such good speedups is the high computation to communication ratio of this application.

Matmul: Our locally developed simple matrix multiplication program also achieved good speedups, especially for larger datasets as shown in Figure 4 (e). The speedup on 16 processors is low (4.30) for $1K \times 1K$ matrices, whereas it is near linear (14.57) for $3K \times 3K$ matrices. As a matter of fact, for 2-processor configurations, the speedups are super linear. Matmul clearly benefits from initial distribution of shared data among processors. Hence, each processor works only on the data assigned to it. As a result, extensive communication occurs only at the end of the computation when processors merge the partial results at the barrier.

Overall, the applications in the test suite achieved speedups from 1.39 (Water, small dataset) to 2.31 (Matmul, small dataset) on 2 processors and from 1.65 (Water, small dataset) to 15.92 (EP, large dataset) on 16 processors. Possibly, the major causes for the variance between speedups are irregular shared data access patterns, low computation to communication ratio and, to some extent, to our unoptimized software DSM system.

3.4 JIAJIA–CVM Comparison

We ran the same applications with CVM software DSM. CVM is based on lazy release consistency model [7]. Also, JIAJIA and CVM uses totally different allocation schemes for shared memory. JIAJIA distributes the shared pages among processors, whereas CVM replicates them on each and every processor. Although static data distribution through home processors improves performance of certain applications, the positive effect of this approach is not always possible.

Figure 5 summarizes the comparison of speedups achieved by three different datasets with JIAJIA and CVM. The above figure also shows that most applications achieve higher speedups with JIAJIA as the dataset size increases and that JIAJIA outperforms CVM in all applications with larger datasets, except Barnes.

Unfortunately, not all applications ran successfully under CVM. We were unable to run LU with a $2K \times 2K$ matrix on more than 4 processors and a $3K \times 3K$ matrix on more than a single processor. On the other hand, this application ran successfully on 1–16 processors with smaller (up to $1K \times 1K$) matrices. Similarly, for reasons unknown to us, Barnes did not run to its completion with 32768 particles on 2 processors. We linearly extrapolated the speedup of this case.

Finally, the applications ran faster with JIAJIA than CVM, mainly because of the simpler coherence protocol of our software DSM system.

4 Conclusions

We described the implementation of a new software DSM system called JIAJIA and its performance on high performance computing environments, such as an IBM SP2 cluster. We also demonstrated that many applications can take advantage of the shared memory programming model on distributed memory computers using the software DSM approach. Generally, the difference between the efforts of programming simple applications, such as EP or Matmul with either models is negligible. However, the code size increases with the complexity of the algorithm, hence writing programs using the message passing model becomes non-trivial. Further, algorithms that have irregular data access patterns and use indirect methods (such as pointers) to access complex data structures make programming with explicit message passing an even more difficult and error-prone task.

The applications described above achieved moderate to good speedups with JIAJIA. The main reason for this is the simplicity of the coherence protocol and the memory organization scheme. We measured the performance of applications with JIAJIA on up to 16 nodes of the SP2 cluster. The implications of larger number of nodes is yet to be investigated. We believe that as the speed of the interconnection media increases, the overhead of extensive message exchange will be less important and coherence protocols with less space and computation overhead will be the winner. On the other hand, faster processors would reduce slightly the speedup from the high bandwidth interconnects.

Currently, JIAJIA uses the UDP protocol with BSD sockets over the high performance switch for the inter-node communication. Our simple communication protocol provides guaranteed message delivery, since UDP is unreliable. We are conducting more detailed timing measurements which hopefully will allow us identify the major bottlenecks in the

system. We are also porting new applications to study possible benefits of JIAJIA on a variety of other application domains.

Acknowledgments

We gratefully acknowledge the Center for High Performance Computing (CHPC) at the University of Utah for the allocation of computer time to run our experiments. CHPC's IBM SP system is funded in part by NSF Grant #CDA9601580 and IBM's SUR grant to the University of Utah.

About the Authors

M. Rasit Eskicioglu received his B.Sc. in Chemical Engineering from Istanbul Technical University (Turkey) and M.Sc. in Computer Engineering from Middle East Technical University (Turkey). He is currently finishing his long overdue PhD thesis. He has been in academia, teaching and doing research and development mainly in systems area for nearly 20 years. His research interests include operating systems, parallel and distributed systems, computer architecture, and computer networks. His recent work involves investigating ways to make software DSM systems more efficient and scalable. He is a member of ACM, IEEE Computer Society, and USENIX.

T. Anthony (Tony) Marsland received his B.Sc. in Honours Mathematics from the University of Nottingham (UK) and M.S.E(E) and Ph.D. degrees in Electrical Engineering from the University of Washington, Seattle (USA). After working one year as an assistant professor he went to AT&T Bell Laboratories in New Jersey for two years as a research scientist, before joining the Computing Science Department at the University of Alberta, where he is currently a professor. He was an ACM National Lecturer during 1979-81 and a McCalla Research Professor in 1985-86. His current teaching and research interests are in the area of distributed computing systems design.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[4] W. Hu, W. Shi, and Z. Tang. A lock-based cache coherence protocol for scope consistency. *Journal of Computer Science and Technology*, 13(2):97–109, March 1998.

[5] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[6] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, May 1996.

[7] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[8] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[9] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message-passing vs. distributed shared memory on networks of workstations. In *Proc. of Supercomputing'95*, December 1995.

[10] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.