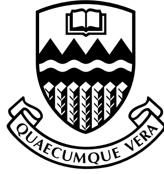

University of Alberta

Evaluation of JIAJIA Software DSM System on High Performance Computer Architectures

by

M. Rasit Eskicioglu and T. Anthony Marsland
and
Weiwu Hu and Weisong Shi

Technical Report TR 98-08
June 1998

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

Submitted to the 32nd Hawaii International Conference on System Sciences, *Software Technology, "Distributed Caching and Replication"* Minitrack, January 1999.

Evaluation of JIAJIA Software DSM System on High Performance Computer Architectures†

M. Rasit Eskicioglu and
T. Anthony Marsland
University of Alberta
Computing Science Department
rasit@cs.ualberta.ca

Weiwu Hu and Weisong Shi
Center for High Performance Computing
Institute of Computing Technology
Chinese Academy of Sciences
dsm@water.chpc.itc.ac.cn

June 1, 1998

Abstract

Distributed Shared Memory (DSM) combines the scalability of loosely coupled multicomputer systems with the ease of usability of tightly coupled multiprocessors. DSM has received much attention in the past decade and many consistency models, protocols, and systems were developed. In this paper, we describe a new software DSM system called JIAJIA, and evaluate it with a suite of widely different applications running on an IBM SP2 cluster, a high performance computer system. Our experiments show that applications can achieve moderate to good speedups with JIAJIA and have performance comparable to the commercial TreadMarks system.

1 Introduction

Shared memory multiprocessors provide an attractive programming model to develop parallel applications. However, as the number of processors increases, the memory access operations saturate the interconnection medium and degrade the performance, hence severely limiting the scalability of the entire system. Distributed memory multiprocessors, on the other hand, scale well but usually awkward to program because all the data movement between memories must be done explicitly in the application. Distributed shared memory is a useful abstraction for making non-uniform memory access (NUMA) multiprocessors more usable and for deploying networks of workstations (NOWs) as a parallel multicomputer. The idea behind DSM is to allow processes executing on different interconnected processors to share memory by hiding the physical location(s) of data, making the memory *location transparent* to the entire system. DSM handles "remote" memory accesses by transparently translating them into messages for the underlying communication media. An important benefit of this approach is that parallel programs developed for (real) shared memory multiprocessors can execute on distributed memory architectures without modification. Further, parallelization of a serial program is usually easier with (distributed) shared memory paradigm.

A DSM system can be realized by software, hardware, or a combination of two. A fundamental difference between software and hardware implementations is the granularity of the coherent shared data. Software DSM systems use a physical page as the unit of sharing (coarse-grain), whereas hardware implementations use smaller sizes (fine-grain), such as a word or a cache line. Generally, software implementation of DSM is more attractive because it involves no (special) hardware and

†This work is supported in part by NSERC grant IOGP7902 and by the National Climbing Program of China.

therefore less expensive. Due to its practical advantages, software DSM has been an active research area over the last decade and several approaches have been proposed and implemented [CBZ91, BZS93, KDCZ94, Kel96].

In this paper, we present the JIAJIA software DSM system [HST98b] and evaluate it with several parallel applications on a high performance computer system. The results of our experiments show that applications running on such environments as an IBM SP2 can take advantage of the DSM approach. Further, the performance of the applications are comparable to the commercial TreadMarks [KDCZ94] system. In the following sections, we give an overview of the JIAJIA software DSM system emphasizing on its distinguishing characteristics, tabulate the applications in our test suite, analyze the results of our experiments, and compare JIAJIA's performance with TreadMarks.

2 JIAJIA software DSM

As Figure 1 shows, JIAJIA is build onto UNIX as a user-level runtime library like several other software DSM systems and uses standard UNIX libraries to accomplish various tasks, such as remote program invocation, interprocess communication, and memory management. It is originally developed under Solaris 2.5 operating system to run on a network of Sun Sparc workstations. We recently ported it to an IBM SP2 cluster running AIX 4.1 to evaluate the performance implications of software DSMs on high performance computer systems.

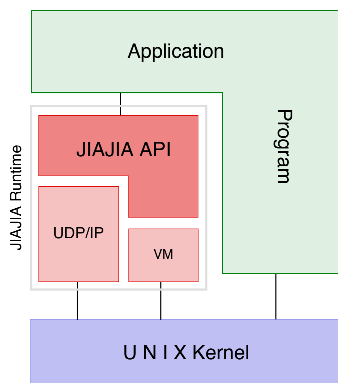

Figure 1: A JIAJIA Node

JIAJIA is a page-based software DSM system which supports scope consistency and uses write-invalidate approach to handle dirty data. Multiple writers are used to alleviate the false sharing problem, typical of software DSM systems. We choose write-invalidation because processors often modify different locations of the page (the unit of sharing) consecutively, thus making the alternative write-update protocol inefficient and more expensive. Such protocol invalidates all copies of the shared page and update only the original copy, which in our case, is the copy on the home (processor) of the page. JIAJIA is different from other software DSM systems in several ways. First, it provides data coherence with the *scope consistency* model [ISL96], which is implemented through a locked-based protocol [HST98b]. Second, it manages the shared memory using a "home"-based scheme. Third, it employs a unique (global) address mapping method and utilizes all the memory available on the processors by combining them to form one large global memory.

2.1 Scope Consistency

The first software DSM prototype IVY [Li88] successfully implemented *sequential consistency (SC)* [Lam79] over a network of workstations, but suffered greatly from high communication overhead as a consequence of "false sharing" (i.e., concurrent access to different variables on the same page, with at least one being a write operation) caused by coarse sharing granularity. More recent systems exploit relaxed models, such as *entry consistency (EC)* [BZ91] or *lazy release consistency (LRC)* [KCZ92] to address this problem. These memory models delay, and if possible restrict, the communication among processors until a synchronization operation occurs. Furthermore, relaxing the consistency requirements generally reduces the overhead of coherence operations. The relaxed models guarantee a "sequentially consistent" execution of programs with the help of two basic synchronization operations: *acquire* (request access to shared data) and *release* (grant access to shared data).

In the EC model, each shared data item is explicitly associated with a synchronization variable (e.g., a lock). At an acquire, only the data item associated with the current synchronization variable is guaranteed to be the most recent. The goal of EC is to minimize the communication costs that are caused by unnecessary data movements. Unfortunately, this approach imposes a programming burden on users by requiring explicit association of each shared data object with a synchronization variable. For simple programs, this may not be a problem. However, additional coding effort is usually needed for programs that use complex data structures, such as nested arrays and these programs may get unnecessarily large as well.

In contrast, LRC, a more relaxed form of *release consistency (RC)* [GLL+90], guarantees that the shared data are the "most recent" at all acquires. This model also allows multiple writers to modify data on the same shared page without any communication among the processors, assuming that the modifications are made to disjoint sections of the page. Furthermore, LRC delays propagating consistency information until the next acquire following a release operation. A (shared) page is duplicated before it is modified the first time. On the next acquire, the releaser compares the original pages (*twins*) with the modified copies and creates *diff*s (record of the changes made to a shared page) to capture the changes. These diffs are then propagated to the new owner of the pages. The drawback of this approach is that all the modifications made on the shared data are propagated to the acquiring processor, even if only a few of them are used later.

Scope consistency (ScC) [ISL96] is proposed to bridge between these two relaxed models, where synchronization variables are not associated with shared data, instead they are associated with sections of code (critical sections) implicitly. The ScC, also a relaxed model, is a refinement of both EC and RC, which establishes a binding between critical sections and the synchronization variables dynamically and transparently. Usually, the critical sections protected by locks imply the "scope" of the desired consistency. In this context, a "consistency scope" can be viewed as the collection of the critical sections protected by a lock. In addition to individual "local" scopes, there is also a "global" consistency scope which is typically marked by barriers and covers the entire program. A program enters a local scope and "opens" a new *session* when it acquires a lock. Any modification made during a session is guaranteed to be seen by other processors that later enter the session(s) of the same (local) scope. General rules for the ScC can be summarized as follows:

- An ordinary memory access (*read* or *write*) is allowed to perform with respect to any other processor only after **all** previous *acquire*s are performed.
- A *release* is allowed to perform with respect to any other processor only after **all** previous ordinary memory accesses to the region (protected by the same lock) are performed.
- Synchronization accesses are sequentially consistent.

Most of the applications tailored for LRC can run under ScC without any modification. However, some applications may require certain modifications to produce correct results. A typical modification is to expand the critical sections by moving the synchronization operations around in the applications. A more detailed description of the ScC model can be found in [ISL96].

We adopted the ScC model in JIAJIA because of its overall advantages and its simpler protocol.

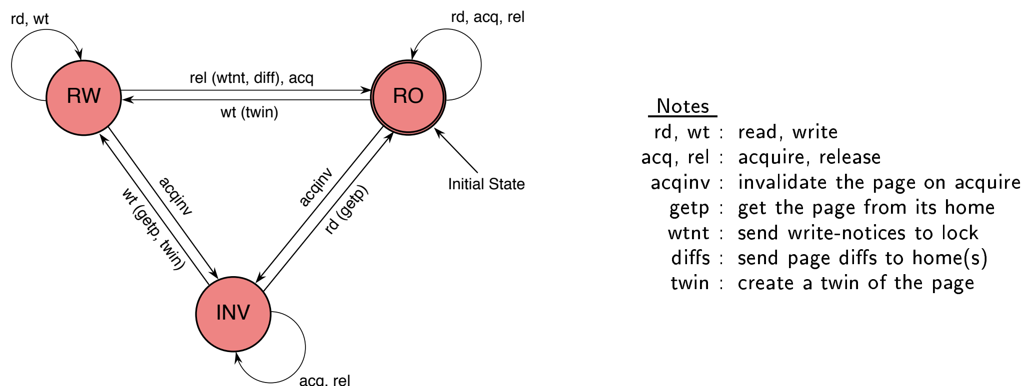

Figure 2: JIAJIA's Coherence Protocol

2.2 Lock-based Coherence Protocol

Based on the observation that the overhead caused by the complexity of a software DSM may easily offset the benefit of such system, the coherence protocol of JIAJIA is designed to be as simple as possible. Figure 2 summarizes the state transitions of the coherence protocol.

Each shared page in an application can either be "local" or "cached" on a given processor. In the former case, the processor is the *home* of that page. These pages can be in one of three states: `Invalid(INV)`, `Read-Only(RO)`, and `Read-Write(RW)`. Since multiple write accesses to shared data are allowed, a page may be cached by several processors in different states concurrently at a given time. Initially, all shared pages are in `RO` state at their home processors. Ordinary read and write accesses to a `RW` page, or read access to a `RO` page, or acquire and release on an `INV` or `RO` page do not cause any transition. Like the shared pages, each lock has a "home" processor which is assigned in a round-robin fashion during the system initialization.

On a release, the processor generates diffs for all modified pages and sends them to their respective homes eagerly. 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, acquiring processor sends a request to the lock's owner and waits until it receives a grant message for the lock. 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 applicable write-notices. After receiving the lock grant message, the acquiring processor invalidates the pages listed in the write-notices and continues with its normal operation. The correctness of the protocol is proved with an event ordering framework in a previous study [HST98a].

All processors exchange the write-notices for the entire shared memory and invalidate applicable pages, and pending diffs are applied to the shared pages at barriers. Thus, the processors start with a fresh up-to-date view of the shared memory after a barrier.

In summary, the protocol propagates a modified page to its home processor on a release and to the next processor on the following acquire, where, stale data are updated later than in EC but

sooner than in LRC. This approach keeps the diffs only for a short period of time, hence avoiding extensive local diff keeping overhead.

Unlike other DSM systems, JIAJIA does not keep any global directory structure separately, instead, locks keep the necessary information, such as ownership, for the relevant pages. This approach further reduces the space overhead of the system.

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

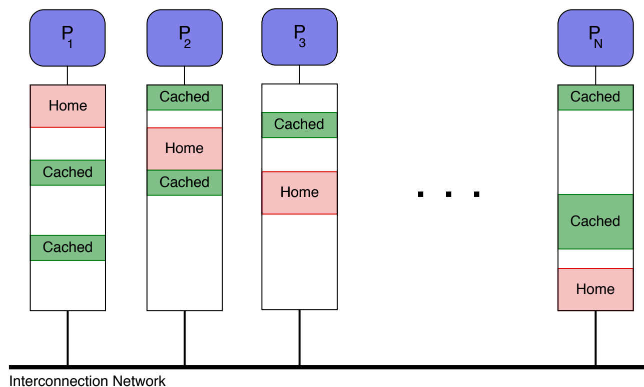

Figure 3: Memory Organization in JIAJIA

2.3 Memory Organization and Address Mapping

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 normally 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 and future 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. The page table 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. Since the shared pages are distributed across all processors, 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.

2.4 JIAJIA's Programming Interface

JIAJIA implements the single program multiple data (SPMD) programming model, in which each processor runs the same program on different parts of the shared data. Figure 4 summarizes JIAJIA's simple, yet powerful programming interface to support shared memory parallel programming.

Function	Purpose
`jia_init()`	Initialize JIAJIA
`jia_alloc()`	Allocate shared memory
`jia_lock()`	Acquire a global lock
`jia_unlock()`	Release a global lock
`jia_barrier()`	Perform a global barrier
`jia_wait()`	Sync without coherence
`jia_clock()`	Return elapsed time
`jia_error()`	Print out an error message
`jia_exit()`	Shut JIAJIA down

Figure 4: JIAJIA API

Additionally, JIAJIA provides two variables, `jiapid` and `jiahosts`, to the user. They specify the host identification number and the total number of hosts of a parallel program, respectively. The programming interface is defined in a C header file `jia.h`, which should be included by the application.

3 Experimental Platform and Applications

We tested JIAJIA software DSM system on an IBM SP2 cluster at the Center for High Performance Computing at the University of Utah. The SP2 cluster consists of 64 nodes, with slightly different characteristics. The results reported here were collected on 16 identical "thin nodes" of the SP2 cluster, each equipped with 120 MHz POWER2 Superchip processor and 128 MB physical memory. The nodes are interconnected with a high performance multi-stage Omega 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 an Ethernet and a FDDI links. Full version of AIX 4.1.5 operating system runs on each node. Our experiments were executed on the nodes in dedicated mode, i.e., with no other user process, thus utilizing the full capacity of each node.

Our test suite includes five applications, namely, Water, LU, EP, TSP, and Matmul, covering a broad range of problem domains with varying behaviors. Water and LU are from the SPLASH [SWG92] and SPLASH-2 [WOT+95], respectively. SPLASH is a collection of parallel applications developed for use in the design of shared-memory multiprocessors, as well as in the study of centralized and distributed share memory multiprocessors. SPLASH2 is the next generation of the SPLASH suite of applications. Consequently, these applications are tailored for hardware (sequential) cache-coherent systems with cache line granularity. EP is from the NAS Parallel Benchmarks [BBLS94]. NAS benchmarks are developed for evaluating the performance of highly parallel supercomputers. TSP is developed at the Rice University in conjunction with their commercial TreadMarks software DSM system [KDCZ94]. 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 amounts of data. Below, we briefly summarize the applications in our test suite. More detailed descriptions for most of them can be found elsewhere [SWG92, WOT+95, BBLS94].

Appl.	**Sync**	**Data Set** *Sm/Md/Lg*	**Sh Mem** *(4K pgs)*
Water	B, L	343 mols	27
		1000 mols	71
		1728 mols	121
LU	B	$1K \times 1K$	2,059
		$2K \times 2K$	8,205
		$3K \times 3K$	18,447
EP	B	2^{24}	1
		2^{26}	1
		2^{28}	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

Table 1: Application Characteristics (B=barriers, L=locks)

Water is an N-body molecular 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 simulates the state of the molecules in steps. 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 [LDCZ95] of Water in our experiments.

LU is a matrix decomposition kernel that factors a dense matrix into the product of a lower and an 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 on 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.

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,

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
	1000 mols	*370.41*	369.93	195.13	102.52	60.91	55.09
	1728 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
EP	2^{24}	*74.69*	74.72	37.58	19.27	9.37	4.37
	2^{26}	*300.41*	300.46	151.24	75.05	37.51	19.27
	2^{28}	*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: Results of JIAJIA (in seconds)

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.

4 Analysis of the Experimental Results

We used `gcc` with option `-O2` to compile both JIAJIA and TreadMarks versions of the applications. The statistics collection code has a negligible overhead (less than % 1) on 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 fact that the `malloc()` system call is cheaper than `mmap()` on most 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 a 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 side effect of this anomaly 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 data sets, small, medium, and large, to see the effect of problem size on application performance, Also, we ran the same set of applications

using TreadMarks version 0.9.4 to allow a fair comparison with JIAJIA. In the following sections, we first discuss the performance of each application under JIAJIA separately and then present a comparative analysis of JIAJIA and TreadMarks.

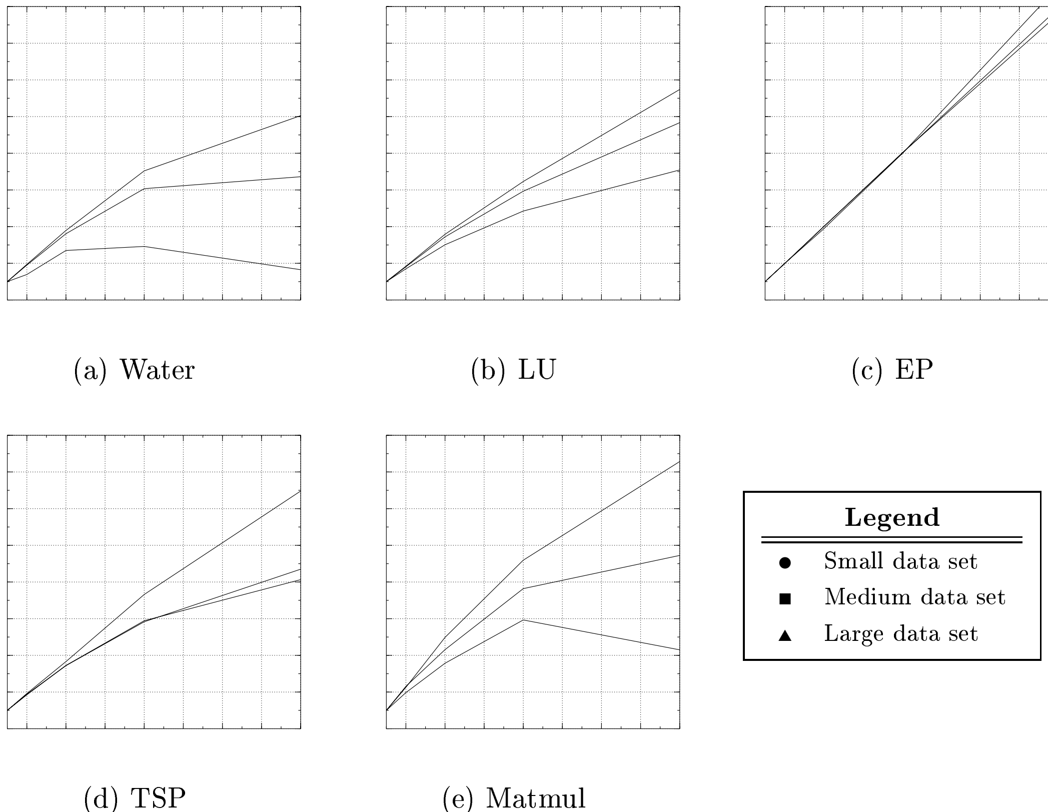

Figure 5: Speedups of JIAJIA

4.1 Performance of JIAJIA

The performance of the applications in our test suite with JIAJIA on a 16-node SP2 cluster 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 5 (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 degrading 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 [SWG92]. 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 5 (b) shows the speedups obtained ranging from 7.09 for a $1K \times 1K$ matrix to 11.48 for a $3K \times 3K$ matrix. Our results confirmed the findings of others that a better performance is achieved for larger problem sizes. We selected a block size of 64 bytes, because after performing

some additional tests, we observed that a block size of 64 (as opposed to 16 recommended by the developers of the application) yields the best performance. Based on this observation, we conclude that with page based software DSM systems, it is more important that the blocks to fit into the coherence unit of the software system (i.e., a physical page) rather than the hardware cache lines.

EP: This application achieved an excellent performance as expected and scaled well. As shown in Figure 5 (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 with 20 cities faster than with 19 cities due to the setup of the input data. The speedup for all three cities up to four processors is near linear. However, beyond four processors, as the number of processors increases, the larger data sets are penalized by our lock-based protocol. JIAJIA transfers mostly complete pages because the accumulation of lock releases unnecessarily invalidate more pages on acquire. Figure 5 (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 application Matmul also achieved good speedups, especially for larger data sets as shown in Figure 5 (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. Since JIAJIA allows the initial distribution of the shared data among processors, Matmul clearly benefits from our home-based coherence protocol.

Appl.	No. of Procs	Speedups	
		JIAJIA	TreadMarks
Water 1728 mols.	2	1.94	1.85
	16	10.05	9.40
LU $3K \times 3K$	2	1.84	1.43
	16	11.48	2.82
EP 2^{28}	2	1.98	2.00
	16	15.92	15.96
TSP 19 cities	2	1.92	1.94
	16	12.96	13.22
Matmul $2K \times 2K$	2	2.31	1.66
	16	9.45	2.33

Table 3: Comparison of JIAJIA and TreadMarks Speedups

4.2 Comparison of JIAJIA and TreadMarks Performance

Table 3 shows the speedups of applications on 2 and 16 processors for both JIAJIA and TreadMarks. Water, LU, and Matmul achieved better speedups with JIAJIA, whereas EP and TSP performed only slightly better with TreadMarks. Overall, JIAJIA versions of the applications outperformed TreadMarks, mainly due to the relatively low overhead of our simple coherence protocol. We also

collected the total number of messages and data exchanged by the processors with JIAJIA and TreadMarks. Table 4 shows these statistics.

Although the number of messages in TreadMarks version of Water is an order of magnitude more than that of JIAJIA version, the total data transferred is only twice as much. The reason for the higher data transfer rate is the fact that JIAJIA usually sends whole pages because of the write-invalidate protocol, but causes less diff accumulation. On the other hand ,TreadMarks sends many small diff messages.

The amount of transferred messages and data are quite similar in both JIAJIA and TreadMarks versions of LU on 2 processors. However, the data amount quadruples with TreadMarks when the we scale to 16 processors, whereas it only doubles with JIAJIA on the same number of processors. Also, the message count on 16 processors is more with TreadMarks, even though it is less on 2 processors. This indicates that JIAJIA's protocol scales better in applications like LU.

EP performs nearly identical with both JIAJIA and TreadMarks. JIAJIA sends more messages and data because of unnecessary invalidation of the shared page.

JIAJIA's lower speedup in TSP is due to the extensive amount of message (5 times more) and data (75 times more) transfers. However, speedups achieved by TreadMarks does not reflect this advantage, because it also suffers from the higher overhead of its diff management.

Matmul transfers slightly more data and messages with JIAJIA, but again, its simple protocol helps achieve better speedups.

The design of TreadMarks does not allow to share large number of pages. For this reason, TreadMarks cannot execute Matmul, for example, with $3K \times 3K$ matrices. On the other hand, JIAJIA's shared memory allocation scheme is only bound by the virtual memory management limitations of the underlying (UNIX) operating system. This feature of JIAJIA allows parallelization of applications that require large amounts of shared data.

Appl.	**No. of Procs**	**JIAJIA**		**TreadMarks**	
		No. of Messages	**Total Data**	**No. of Messages**	**Total Data**
Water 1728 mols.	2	12,100	34 MB	200,526	60 MB
	16	210,299	325 MB	1,546,692	719 MB
LU $3K \times 3K$	2	38,040	72 MB	37,513	73 MB
	16	152,288	144 MB	155,777	298 MB
EP 2^{28}	2	18	8 KB	9	4 KB
	16	270	128 KB	177	88 KB
TSP 19 cities	2	13,000	20 MB	2,763	268 KB
	16	27,357	43 MB	10,759	5 MB
Matmul $2K \times 2K$	2	16,396	32 MB	12,303	24 MB
	16	245,834	480 MB	236,314	468 MB

Table 4: Message count and data sizes with JIAJIA and TreadMarks

5 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 NUMA

architectures using the software DSM approach. The applications described above achieved a 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 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.

Currently, JIAJIA uses UDP/IP over the high performance switch for the inter-node communication. We are developing a new version to use MPI as the communication layer and also porting new applications to study the possible benefits of JIAJIA on a variety of other application domains. We're hoping to report more results for additional applications in the final version of the paper. Our future work with JIAJIA on high performance computer architectures includes experimenting with various techniques, such as multicast messages and write-update protocol, to improve the performance of the system.

6 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.

References

[BBLS94] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BZ91] B. N. Bershad and M. J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[BZS93] 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.

[CBZ91] 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.

[GLL+90] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[HST98a] W. Hu, W. Shi, and Z. Tang. A framework of memory consistency models. *Journal of Computer Science and Technology*, 13(2):110–124, March 1998.

[HST98b] 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.

[ISL96] 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.

[KCZ92] 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.

[KDCZ94] 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.

[Kel96] 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.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LDCZ95] 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.

[Li88] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[SWG92] 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.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22th Annual Symp. on Computer Architecture*, pages 24–36, June 1995.