ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Future Generation Computer Systems 22 (2006) 57–66

FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Simulation tools to study a distributed shared memory for clusters of symmetric multiprocessors

Darshan D. Thaker [a,*], Vipin Chaudhary [b]

[a] *University of California, Davis, 2569 Wallace Street, Oakland, CA 94606, USA*
[b] *Wayne State University, Detroit, MI, USA*

Available online 27 June 2005

Abstract

Distributed shared memory (DSM) systems have become popular as a means of utilizing clusters of computers for solving large applications. We have developed a high-performance DSM, *Strings*. In addition, to improve the performance of our DSM, a memory hierarchy simulator has been developed that allows us to compare various techniques very quickly and with much less effort. This paper describes our simulator, DSM*Sim*. We show that the simulator's performance closely matches the real system and demonstrate potential performance gains of up to 60% after adding optimization features to the simulator. The simulator also accepts the same code as the software distributed shared memory.
© 2004 Elsevier B.V. All rights reserved.

1. Introduction

In recent years, single processor based computers have evolved rapidly. Processor speeds are now in excess of 2 GHz. Yet, they are not able to solve increasingly large and complex scientific and engineering problems. Another trend is the decline in the number of specialized parallel machines being built to solve such problems. Instead, many vendors of traditional workstations have adopted a design strategy wherein multiple state-of-the-art microprocessors are used to build high performance shared-memory parallel workstations. From a computer architecture point of view, such machines could use tightly coupled processors that access the same shared memory, known as symmetrical multiprocessors (SMPs), or one could cluster multiple computing nodes with a high-performance interconnect. The latter approach has been extended by combining multiple SMPs to provide a scalable computing environment. Examples of this important class of machines include the IBM SP2, SUN Enterprise and SUNFire Series and machines from HP.

These symmetrical multiprocessors (SMPs) are then connected through high-speed networks or switches to form a scalable computing cluster. A suitable runtime system should allow parallel programs to exploit fast shared memory when exchanging data within nodes and using the slower network only when

* Corresponding author.
E-mail address: thaker@cs.ucdavis.edu (D.D. Thaker).

0167-739X/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2003.12.024

necessary. Existing sequential application programs can be automatically converted to run on a single SMP node through the use of parallelizing compilers such as SUIF [1], etc. However, using multiple nodes requires the programmer to either write explicit message passing programs, using libraries like MPI [5] or PVM [6], or to rewrite the code using a new language with parallel constructs ex. OpenMP [3], HPF and Fortran 90. Message passing programs are cumbersome to write and have to be tuned for each individual architecture to get the best possible performance. Parallel languages work well with code that has regular data access patterns. In both cases, the programmer has to be intimately familiar with the application program as well as the target architecture. The shared memory model is easier to program since the programmer does not have to worry about the data layout and does not have to explicitly send data from one process to another. However, hardware shared memory machines do not scale that well and/or are very expensive to build. Hence, an alternate approach to using these computing clusters is to provide an illusion of logically shared memory over physically distributed memory, known as a distributed shared memory (DSM) or shared virtual memory (SVM). Recent research projects with DSMs have shown good performance, for example IVY [9], TreadMarks [4], Quarks [8] and CVM [10].

At Wayne State University, we developed a DSM called Strings. The Strings system consists of a library that is linked with a shared memory parallel program. Strings is a page based DSM, that uses an update protocol for Release Consistency. The performance and capabilities of Strings are described in detail in previous work [1]. Currently, our effort is to enhance the performance of Strings. This effort includes studying in detail how Strings manages memory and designing different methods to alleviate the areas that act as a drag on efficiency. Towards this end we have developed an execution driven memory hierarchy simulator that closely mimics the behavior of Strings. The simulator allows us to test different approaches and ideas quickly and at a cost far lesser than what we would have incurred if we had modified Strings itself. In the rest of this paper, we review some features of Strings in Section 2. The following section describes the implementation and features of our simulator. Thereafter is a section on observed performance. We conclude in Section 5.

2. Strings

The Strings distributed shared memory was derived from the publicly available Quarks [8]. It consists of a library that is linked with a shared memory parallel program. The system allows the creation of multiple application threads on a single node, thus increasing the concurrency level on an SMP cluster.

A program starts up and registers itself with the server. It then forks processes on remote machines. Each forked process in turn registers itself with the server and then creates a DSM server thread, which listens for requests from other nodes. The master process creates shared memory regions coordinated-ordinated by the server in the program initialization phase. The server maintains a list of region identifiers and global virtual addresses. Each process translates these global addresses to local addresses using a page table. Application threads are created by sending requests to the appropriate remote DSM servers. Shared region identifiers and global synchronization primitives are sent as part of the thread create call. The newly created threads obtain the global addresses from the server and map them to local memory. The virtual memory sub-system is used to enforce proper access to the globally shared regions.

Shared memory in the system is implemented by using the UNIX mmap call to map a file to the bottom of the stack segment. The mprotect call is used to control access to the shared memory region. When a thread faults while accessing a page, a page handler is invoked to fetch the contents from the owning node. Strings currently supports sequential consistency using an invalidate protocol, as well as release consistency using an update protocol [5], [14].

Allowing multiple application threads on a node leads to a peculiar problem with the DSM implementation. Once a page has been fetched from a remote node, its contents must be written to the corresponding memory region, so the protection has to be changed to writable. At this time no other thread should be able to access this page. User level threads can be scheduled to allow atomic updates to the region. However, suspending all kernel level threads can potentially lead to a deadlock and would also reduce concurrency. The solution used in Strings is to map every shared region to two different addresses. It is then possible to write to the secondary region, without

changing the protection of the primary memory region.

Strings utilize the concepts of *twins* and *diffs* to allow multiple application threads on the same node. All shared regions are mapped to two different addresses. This makes it possible to write to the secondary region, without changing the protection of the primary region. The copy of the page that is mapped to the secondary region is called a *twin*. The thread then modifies the *twin*. When the pages have to be synchronized, the difference between the original page and its twin is computed. These are called the *diffs*, which are then sent to the remote nodes that share the page. For further details, we invite you to refer to a previous paper [1] that discusses Strings in greater detail.

3. Simulator for Strings

Previous work has demonstrated the performance capabilities of Strings. Nonetheless, there exist various areas where improvements and performance enhancements can be achieved. Some of these improvements can be in the area of networking, memory management, etc. With regard to memory management there are various approaches that can potentially change and improve the manner in which Strings behaves.

3.1. Motivation

To experiment with certain memory management functions in Strings would be time consuming. And consider that even after the change was accomplished there is no guarantee that this would positively effect the outcome in terms of performance enhancement. It is with this in mind that we decided to build a simulator for Strings. It would be easy to make a modification in the simulator and see the effects it would produce. Provided the performance enhancement was satisfactory, the changes could then be incorporated into Strings. Thus, it would function as a test bed for nascent ideas and design philosophies.

The simulator could also serve as a teaching tool, in that it would allow a student unfamiliar with the complexities of the distributed shared memory system, to understand the underlying principles and reasons behind certain design decisions.

3.2. Simulation system details

The simulator is based on Augmint [7]. Nonetheless, it is portable and can also be used on SUN machines with UltraSparc Architectures (using ABSS [11]). Some of the features of the simulator are:

- accurately mimics the behavior of Strings;
- it works with real applications that interact with the environment;
- applications that work on the simulator will work on Strings;
- provides 'ease of use' with respect to understanding and modifying the simulation architecture;
- runs on both Intel and UltraSparc architectures;
- manages the scheduling of events;
- recognizes memory references.

The subsystems of the simulator are:

- Augmint Doctor—the x86 code augmenter;
- simulation infrastructure (Augmint) - including event management, task scheduling and thread switching;
- the distributed shared memory model under study;
- applications—written using ANL m4 macros.

3.3. Augmint

Augmint is a software package on top of which multiprocessor memory hierarchy simulators can be constructed. Augmint consists of a front–end memory event generator and a simulation infrastructure which damages the scheduling of events. The Augmint Doctor takes an x86 assembly source file as input and analyzes it for memory references. Each instruction that performs a memory reference causes additional code to be added before or after that instruction. This code handles details like saving state and setting up the simulation event structure.

3.4. Simulator build and execution model

DSM*Sim* is an execution based simulator. The simulator build process (Fig. 1) creates an executable. The build process is as follows—the source code of the application under study is passed through the macro m4 preprocessor, where all the ANL like macros are expanded. (This is another advantage of the simulator

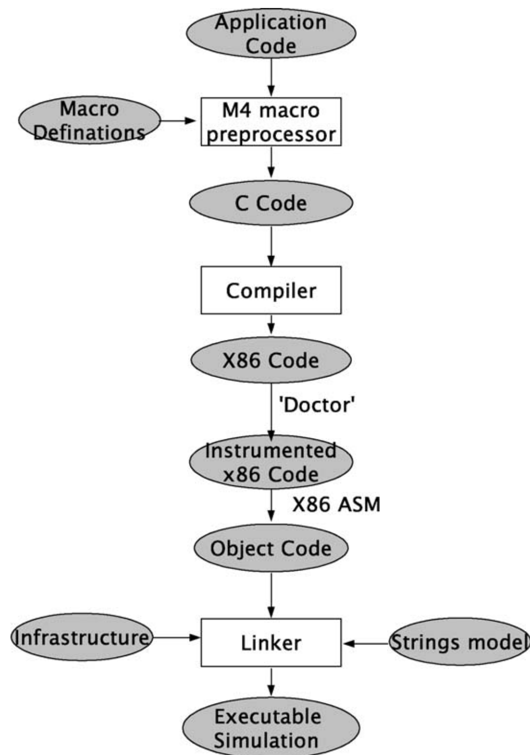

Fig. 1. DSM*Sim* build process for the Intel architecture.

that all the applications that run on the simulator, run on Strings. The same application source code can be used for both the simulator and Strings. The difference is at the m4 preprocessor stage.) Following this, the C compiler is run that converts the C code into assembler code. The Augmint Doctor is then used to augment the assembler code. This is then lined with the simulation libraries. The final result is an executable that contains the application code, the distributed shared memory system under study and the simulation infrastructure. When running, this executable consists of several user level threads that act as simulated processors and a simulation thread that acts as the scheduler.

When the executable is run, it first initializes the environment. The command line arguments are parsed by the architecture simulator and the application. *sim_init* is then called. The address space and thread structures are then initialized. The main application thread is initialized and set to point to the function *main_envelope*, which calls the application. The time wheel is initialized. An empty task is scheduled for the main application thread on the time wheel. The purpose of this task is to invoke a thread switch to the main application thread to start running *main_envelope*.

To ensure that all operations are simulated in chronological order, Augmint provides support for creating and scheduling arbitrary operations; each independently schedulable operation is called a task. Each task has associated with it a time value, a priority and a function pointer. Each task is executed at its specified time in order of priority. Tasks are executed by calling the function through the function pointer. The return value from this function controls thread execution. Every event generated by a thread blocks that thread and creates a new task. The function pointer for the task points to the simulator function corresponding to the type of event. The task also contains a pointer to the event structure and the process id.

3.5. Implementation details

When developing DSM*Sim*, the first goal was to replicate the behavior of Strings as accurately as possible. When running a simulation executable, the programmer passes the simulator arguments and the application arguments. At the start of execution, all the shared memory is allocated. The simulator uses hash tables to keep track of the pages that reside on each node. Thus every node in the simulation has a hash table, the entries of which are the pages that reside on that node. The base address of the page is used when inserting into the tables. Every page is owned by one node. Page ownership is migratory at first fault.

We use the concept of ownership for a page. A page is owned by a node that is participating in running the application. When the application requests for shared pages, the simulator assigns an owner to each page. Ownership of the page is migratory on first fault. The first node that faults on the page becomes the new owner. Thereafter, the ownership does not change for that page. Initially the simulator assigns node zero as the owner for all pages. The page tables are initialized and the copy-set for each page is also initialized. For every page, the owner has both read and write permissions, whereas other nodes have neither.

If the ownership of a page changes, i.e., on a first fault, then the copy-sets and page permissions are updated. Since the simulator is not distributed across physically distinct machines, we can have a global table

that all the nodes in the simulation have access to. This reduces complexity.

3.6. Memory references

Every memory reference has to be analyzed. If a reference is accessing a shared memory location, then appropriate action has to be taken. During the simulation, all memory references are intercepted by the simulator front–end. An access may either be a read or a write and may either be to a shared memory location or to one that is not shared. If the memory access is to a non-shared location, then the execution of the application continues after the statistics are updated. If the access occurred on a shared region, then the simulator needs to look up the page related information for that memory location. In order to do this, the task pointer that is passed by the simulator to the *sim-read* function is utilized. (The code is outlined in Fig. 2.) The task pointer provides many details, like event associated with the task, task owner, next task in the queue, etc. We are mainly interested in the event that was associated with the task (in this case a memory access). The event structure gives details like event type, time at which it occurred, size of data access, access address, etc. (For detailed description of each field in the task and event structures, please refer to [7].) From the task pointer, we obtain the memory address that was being accessed. Thereafter, this address is converted to the base address of the page, to which that memory location belongs. This has to be done because, the hash table lookup needs the base address of the page, otherwise it will not be able to return a valid match. It has to be remembered that each node has its own separate page table and we have to access the correct one.

Once the hash table lookup returns with a page table entry, the permissions for the page determine whether we have a page hit or a page fault. If the permissions are PAGE_READ or PAGE_WRITE, then a memory read can proceed. Returning a value of T_ADVANCE means that the executing thread can proceed. When a page hit occurs, the statistics are updated and we return with T_ADVANCE, thus allowing the simulation to continue. If the access was a page fault, then the read or the write fault handler is called depending on the type of access.

When a page fault handler is called, it means that faulting node does not have a current copy of the page. The first thing that the read fault handler does, is send a request to the owner of the page for a copy of the page. When the page is received, the requesting node changes its copy-set of the page to mirror that of the copy-set sending node. Once the copy-set is changed, the receiving node checks to see if it is the only one with a valid copy of the page, it if is, then it becomes the owner of the page. (This happens when the page fault is the first fault that has occurred on the page.) Details on how the copy-set is used are further ahead. Once the permission of the page is changed to PAGE_READ, the read fault handler returns. The accompanying flowchart 3 describes a memory read operation.

The write access to shared memory is handled in a very similar manner. There are a few obvious differences. The permission on the page being accessed has to be PAGE_WRITE, else we have a write fault. The write fault handler sends a request for the page only if the page permissions are PAGE_NONE. In that case, it will behave just like the read fault handler. Upon receipt of the page, the permission will be set to PAGE_READ.

```
pg *get_page_info(task_ptr ptask)
{
int node_id = NODE(ptask->pid);
unsigned int addr = ptask->pevent->paddr;
pg *pte;

PGROUND(addr);

pte = (struct pg *)hash_lookup(p_table[node_id],addr);
return pte;
}
```

Fig. 2. Get_page_info() details.

Following this, the write fault handler checks the copy-set of the page to see if the current node is the only one writing to the page. If it is not, then a twin of the page is created, further writes are done on the twin.

It should be understood that upon a page fault, when a node requests the required page from the owner, frequently the request may not be immediately fulfilled. For example, let's assume that there are four nodes N0, N1, N2 and N3 in a simulation. Initially the owner of page P1 is N0. Therefore, all four nodes have N0 as the owner of P1 in their page tables. N2 faults on P1. It sends a request to N0 for P1. The request is serviced and since this is the *first fault*, there is a change of ownership. The new owner of P1 is N2. This is reflected in the page tables of N0 and N2 only. When N1 faults on P1, it believes that N0 is still the owner and hence sends a page request to N0. This page request is forwarded by N0 to N2. N2 services the request. When N1 receives the page, it checks the copy-set and updates its page table accordingly. Thus, it is said that page requests are sent to the *probable* owner.

4. Performance analysis

Applications that are written for Strings using the ANL m4 macros, can be run on the simulator fairly easily. The macro file *c.m4.dsmsim* should be used for the simulator. This file is used to convert *.C and *.H files to *.c and *.h, respectively. The usage can be understood from the sample applications that are available with the simulator. It is derived from the Augmint macro file.

We used applications from the Splash-2 [2] Benchmark Suite. FFT performs a transform of n complex data points and requires three all-to-all interprocessor communication phases for a matrix transpose. The data access is regular. LU-c and LU-n perform factorization of a dense matrix. The non-contiguous version has a single producer and multiple consumers. It suffers from considerable fragmentation and false sharing. The contiguous version uses an array of blocks to improve spatial locality. Radix performs an integer radix sort and suffers from a high-degree of false sharing at page granularity during a permutation phase. The matrix multiplication program uses a block-wise distribution of the resultant matrix. Each application thread computes a block of contiguous values; hence, there is no false sharing.

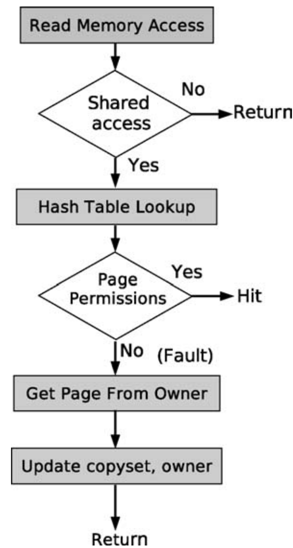

Fig. 3. Flowchart for a memory read operation.

As stated earlier, the simulator can run on the Intel and the SPARC platforms. In most cases, a single CPU (600 MHz Pentium) with 192 MB RAM was used for the runs. Strings on the other hand was run on a cluster of SUN Enterprise 3500s each with four UltraSparc II processors at 330 MHz and with 512 MB of RAM. The following programs were utilized for our experiments—LU-c, FFT, Radix, LU-n and matrix multiplication. We judged our results by how closely the number of page faults in DSM*Sim* matched that of the real system. Since every memory access was trapped by DSM*Sim*, this gave us the ability to collect great amounts of data that gave details of the behavior of the different applications. We could collect details on the number of hits, faults, types of accesses, etc., for each page, each thread and each node. In addition, we could calculate the number of *diffs* that would be generated. It was also possible to see when, as a result of the update protocol, a thread releasing a lock would send *diffs* to another thread that was no longer interested in them. The biggest advantage of gathering such vast amounts of data is that it gives the application developer an idea of how to partition the data, to boost performance (Fig. 3).

Figs. 4 and 5 show the results comparing DSM*Sim* and Strings. It can be observed that most of the runs resulted in a negligible difference between Strings and DSM*Sim*. The runs shown in the figure are 128×128

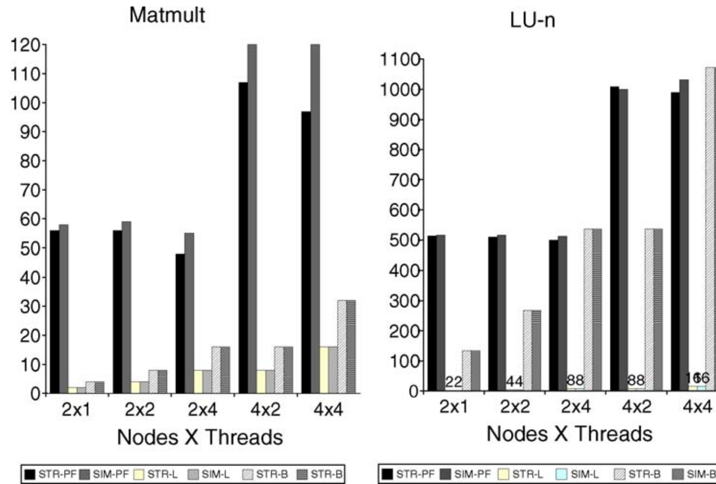

Fig. 4. Comparison between Strings and the simulator.

matrix multiplication; FFT with 65536 complex doubles, LU decomposition on a 512×512 matrix with a 16×16 block size. We show the number of page faults in the simulator (SIM-PF) and in Strings (STR-PF), the locks and barriers for the simulator (SIM-L and SIM-B) and for Strings (STR-L and STR-B). In the legend, STR is for strings and SIM is for DSM*Sim.* The average difference in number of page faults is 1.2%, with the worst case being a difference of 12%. The difference in the page fault reading stems from the fact that DSM*Sim* runs as a single process, whereas Strings spawns multiple processes that in turn spawn threads on multiple machines. As a result, in Strings, if two threads on the same node access the same page (causing a fault) at the same time, this is regarded as just one fault. However, in DSM*Sim*, this would be considered two faults. From our results we can deduce that DSM*Sim* accurately mimics Strings.

From the accompanying Figs. 4 and 5, it can be observed that most of the runs resulted in a negligible difference between the Strings and the simulator. The runs shown in the figure are 128×128 matrix for

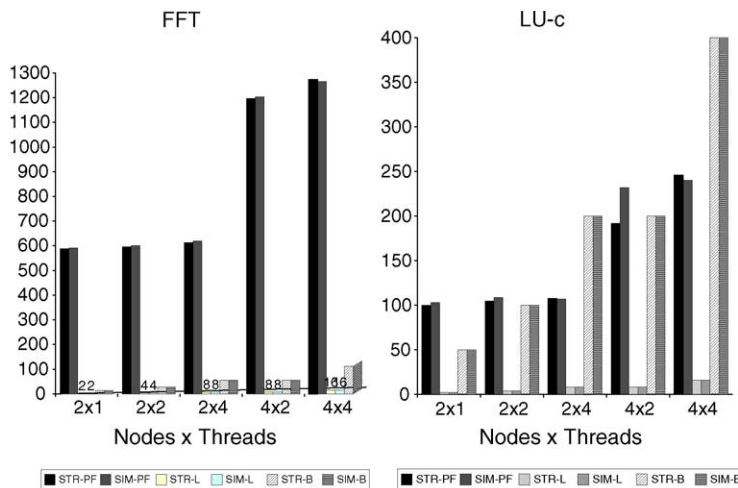

Fig. 5. Comparison between Strings and the simulator.

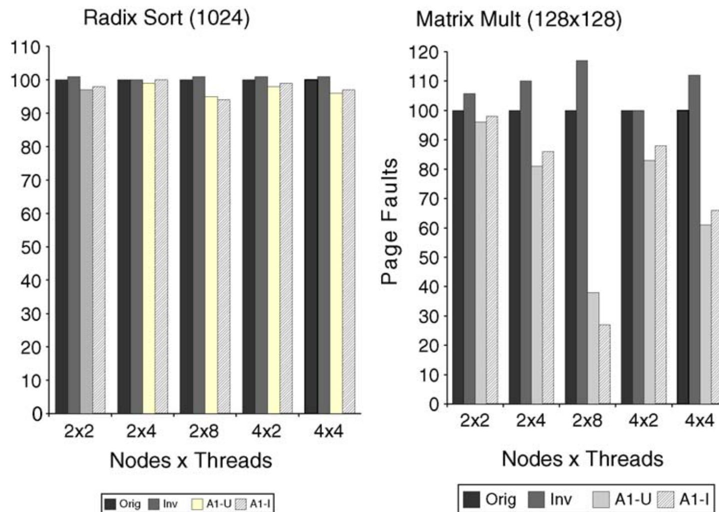

Fig. 6. Performance enhancements on the simulator.

matrix multiplication. For FFT 65536 complex doubles, for Lu 512×512 matrix and a 16×16 block size. The messages measured are protocol messages. Messages that are sent as a result of UDP based networking in Strings are not considered in the simulator. Thus, we could deduce that the simulator was behaving very much like Strings would. The next challenge was to add features to the simulator that would yield an increase in performance.

The following features were added to the simulator. We changed the relationship between the threads and the nodes. Since the ownership of the pages is based on the nodes, this changed the manner in which the threads were accessing the pages. We included an invalidate protocol. Another change was the manner in which owners are assigned to pages. In Strings, pages are initially assigned to node zero; thereafter, on first fault the ownership changes hands. We kept this feature and also assigned pages in a round robin fashion without changing ownership at first fault. The user can select either of the new options or the original system behavior by selecting the appropriate command line arguments at runtime. Figs. 6 and 7 show the effect of the different features that were added. On the '*y*-axis' we plot the number of page faults (normalized). The original version uses an update protocol and release consistency. The second column is an invalidate protocol and the last two columns are both the update (A1-U) and invalidate (A1-I) protocols with a different scheme of assigning data. It can be observed that the most effective feature added was the first one. For matrix multiplication, there is a significant improvement when we have more threads per node. The same trend is seen in radix, although the improvements are not so significant. It can be seen that the invalidate protocol when compared to the update does not always give an improvement in the number of faults.

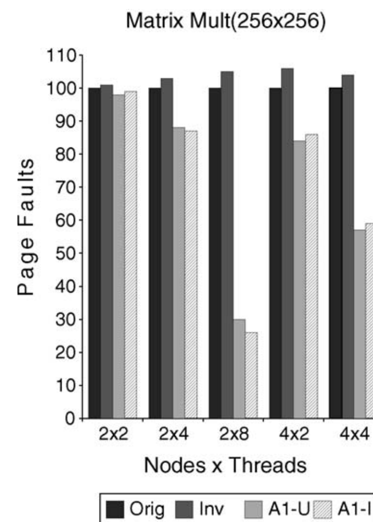

Fig. 7. Performance enhancements on the simulator.

4.1. Optimizations

With the DSM*Sim* accurately mimicking Strings, we added features to DSM*Sim* that would potentially improve the performance of Strings. The following features were added to the simulator:

- We changed the relationship between the threads and the nodes. Since the ownership of the pages is based on nodes, this changed the manner in which the threads were accessing the pages. The threads could be assigned in a block fashion or in a round robin fashion.
- The default behavior of DSM*Sim* follows Release Consistency with an update protocol. We added an invalidate protocol that guarantees sequential consistency. Thus, when a thread releases a lock, instead of sending *diffs* to other nodes that share the page, it invalidates those pages. Also after a barrier has been acquired, only the owner shall have a valid page.
- Another change was the manner in which owners are assigned to pages. In Strings, pages are initially assigned to node zero; thereafter, the first node that faults on a page becomes the new owner of that page. This is the default behavior of the simulator. In addition, the owners can be initially assigned in a round robin fashion and the user can choose to change the ownership at first-fault or leave the owners unchanged.

The user can choose to retain the original system behavior or use any of the options previously mentioned. This can be done by selecting the appropriate command line arguments at runtime. In this manner, it is possible to combine different options and observe the results. Some features that we are currently working on include allowing the page_size to be a variable, by using the concept of a page_block, which may be one or more pages. Its value could be different for different applications. Fig. 7 shows the effect of the different features that were added. On the '*y*-axis' we plot the number of page faults (normalized). The original version uses an update protocol and release consistency. The second column is an invalidate protocol and the last two columns are both the update (A1-U) and invalidate (A1-I) protocols with a different scheme of assigning data. It can be observed from figure that the most effective feature added was the first one. For matrix multiplication there is a significant improvement when we have more threads per node. The same trend is seen in radix, although the improvements are not so significant. It can be seen that the invalidate protocol when compared to the update does not always give an improvement in the number of faults.

5. Conclusions and future work

We developed a distributed shared memory simulator, DSM*Sim* that effectively matches the behavior of Strings. It also provides a portable platform for conducting research in memory consistency models and protocols. In addition, the simulator can be used as a teaching tool to understand the complexities of a DSM. DSM*Sim* is portable across both SPARC and Intel platforms. Various applications can easily be ported for the simulator. It is also very modular and allows a researcher to easily add features and change its behavior. Its ability to run on a single CPU machine is invaluable, as one does not need expensive SMP machines to study DSM characteristics.

Our future work includes incorporating changes that we have observed as advantageous, into Strings. We plan on adding features related to memory consistency models and coherency protocols and comparing their characteristics. In addition, features like data visualization will be added in the future to effectively study and understand the data that is generated by the simulator.

References

[1] S. Roy, V. Chaudhary, *Strings:* a high-performance distributed shared memory for symmetrical multiprocessor clusters, in: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, July, 1998, pp. 90–97.

[2] S.C. Woo, M. Ohara, E. Torri, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the International Symposium on Computer Architecture, June 1995, pp. 24–36.

[3] The OpenMP Forum. OpenMP: Simple, Portable, Scalable SMP Programming. http://www.openmp.org/.

[4] P. Keleher, A. Cox, S. Dwarkadas, W. Zwaenepoel, TreadMarks: distributed shared memory on standard workstations and operating systems, in: Proceedings of the Winter 1994 USENIX Conference, 1994.

[5] Message Passing Interface (MPI) Forum, MPI: a message-passing interface standard, Int. J. Super-Comput. Appl. 8 (3/4) (1994).

[6] V.S. Sunderam, PVM: a framework for parallel distributed computing, Concurrency: Pract. Experience 2 (4 (December)) (1990) 315–339.

[7] A.-T. Nguyen, M. Michael, A. Sharma, J. Torrellas, The Augmint multiprocessor simulation toolkit for Intel x86 architectures, in: Proceedings of the 1996 IEEE International Conference on Computer Design (ICCD), October 1996.

[8] D. Khandekar. Quarks: Distributed shared memory as a basic building block for complex parallel and distributed systems. Technical Report, University of Utah, March 1996.

[9] K. Li, IVY: a shared virtual memory system for parallel computing, in: Proceedings of the 1988 International Conference on Parallel Processing, August 1988, pp. 94–101.

[10] H. Han, C.-W. Tseng, P. Keleher, Eliminating barrier synchronization for compiler-parallelized codes on software DSMs, Int. J. Parallel Programming (October) (1998).

[11] D. Sunada, D. Glasco, M. Flynn, ABSS v2.0: a SPARC simulator in, in: Proceedings of the eighth Workshop on Synthesis And System Integration of Mixed Technologies, 1998.