Running OpenMP applications efficiently on an everything-shared SDSM *

J.J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
{jcosta,toni,xavim,eduard,jesus}@ac.upc.es

Abstract

Traditional software distributed shared memory (SDSM) systems modify the semantics of a real hardware shared memory system by relaxing the coherence semantic and by limiting the memory regions that are actually shared. These semantic modifications are done to improve performance of the applications using it. In this paper, we will show that a SDSM system that behaves like a real shared memory system (without the afore mentioned relaxations) can also be used to execute OpenMP applications and achieve similar speedups as the ones obtained by traditional SDSM systems. This performance can be achieved by encouraging the cooperation between the SDSM and the OpenMP runtime instead of relaxing the semantics of the shared memory. In addition, techniques like boundaries alignment and page presend are demonstrated as very useful to overcome the limitations of the current SDSM systems.

1. Introduction

Clusters, or networks of workstations, are becoming one of the most widely used platforms to run parallel applications. The reasons behind this trend are mainly two: the cost of these systems is quite affordable and the technology to build them has reached a maturity point to make them stable and easy enough to be exploited.

In these architectures, when the shared-memory programming model is used, applications are run on top of a software distributed shared memory runtime (SDSM), but unfortunately it is not a real shared memory one. In order to improve performance, all implementations change the semantics of the shared memory to something easier to implement [1, 15, 6, 12, 16]. The two main changes are the relaxation of the coherence semantic and the limitation of the shared areas. The idea of the first modification to the semantic is that a change done in a node will not be visible on the other nodes till something special occurs (synchronization, new thread creations, ...). The other modification usually done to the "shared-memory" semantics is that not the whole memory is shared and thus the specific areas to be shared have to be explicitly defined. These two changes mean that in order to run a real shared-memory application, it has to be modified both in the memory really shared and in the way it is accessed. In addition, if we want to achieve performance, many tricks have to be done by the programmer, taking into account the side effects of the relaxations and their implementation and that are not portable from one implementation to another. It is clear that this is not an approach with future.

Our proposal is to first implement an everything-shared distributed memory system (NanosDSM) and then prove that it can be used to run OpenMP applications efficiently. To achieve this efficiency, the problems that appear in the SDSM performance will have to be solved by a tight cooperation between the application runtime and the SDSM system. To prove our point, we will implement an OpenMP runtime that cooperates with the SDSM system and achieves speedups comparable with the ones achieved over a relaxed-semantic SDSM, but without paying the price of relaxation.

*This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC2001-0995-C02-01 contract and by the FET program of the EU under contract IST-2001-33071

2. Running OpenMP on SDSM

In this section, we will first present the main difficulties found when trying to run efficiently OpenMP applications on a distributed memory system such as a cluster. Then, we will present the solutions traditionally proposed by OpenMP implementators and SDSM builders.

2.1 Main problems and solutions

The main problem in a SDSM system is that moving or copying a page from one node to another is a quite time consuming task. This overhead can be assumed as long as pages are not moved from one node to another very frequently. Due to the locality found in most programs, only a few pages are needed by a node at any given time.

One problem appears when a page is needed in exclusivity by two different nodes at the same time. If the two nodes modify different data located in the same page, we have a ping-pong of the page from one node to another when it is not really shared. Should the data be placed in different pages, the "sharing" would be avoided. This problem is known as false sharing.

A popular solution to this problem is to relax the idea that any modification has to be seen immediately by all nodes. This will allow several nodes to hold the same page and modify it as long as they do not modify the same data. Once the application reaches a synchronization point, the modified data is forwarded to the rest of the nodes. This solution solves the problem but lies a tougher one to the application: programmers have to change their way of thinking as they cannot assume a modification is done till the next synchronization point. In addition, this is not even always true as threads in the same node will see this modification while "threads" in a different node will not.

Even when there is no false sharing, we still have the problem of copying or moving pages from one node to another. A traditional solution proposed to solve this problem is to overlap the data movement with the execution of the application. Prefetching is the mechanism used, but this mechanism does not cooperate with the application and thus it is very difficult to prefetch the pages on the right moment, just when it will not cause any extra problems such as a ping pong or unnecessary network/CPU consumption. Although prefetching may work in some cases, there are many where this prediction and finding the appropriate time to do it are not feasible.

Finally, current SDSM systems do not share the whole memory, but only the regions explicitly declared as shared. This reduces the potential overhead as the number of shared pages is minimized. The drawback is that, like in the previous solution mentioned, it also places the burden on the programmer that has to have, a priori, a clear idea of the regions that are shared and the ones that are not.

2.2 Related Work

As we have mentioned, several OpenMP runtime systems have already been implemented on top of SDSM systems. The most significant ones are the OpenMP translator developed by Hu et al. [7], OpenMP on the SCASH system [12, 5], and ParADE [9].

Hu et al. [7] develop an OpenMP translator to run OpenMP applications on top of TreadMarks on a cluster of IBM SP2 nodes. As TreadMarks stacks are private, the translator deals with variables declared locally in procedures, allocating them in a shared heap in order to allow them to be accessed from parallel regions.

OpenMP has also been implemented on top of the SCASH system [12, 5]. This approach uses the Omni compiler [13] to transform OpenMP code to parallel code with calls to a runtime library. The SCASH system is based on a release consistency memory model which allows multiple writers. The consistency is maintained at explicit synchronization points, such as barriers.

ParADE [9] is implemented on a lazy release consistency (HLRC[16]) system with migratory home. The communications subsystem relies on MPI. The translator for ParADE is also based on the Omni Compiler. The synchronization directives are translated to collective communications on MPI. The work-sharing constructs are mapped by the compiler to explicit message passing.

Although they may have some differences among them, the main differences compared to our proposal are: i) that they all use a relaxed semantic SDSM while

our proposal is to use a sequential-semantic memory system and encourage the cooperation between the shared memory and the OpenMP runtime; and ii) that they generate specific code for the SDSM, which is different from the code generated for SMP machines. Our proposal does not require any modification to the OpenMP compiler, as all the message management is done at the level of the OpenMP runtime.

3. Our environment

3.1 NanosDSM: An everything-shared SDSM

Main philosophical ideas

As we have already mentioned, our environment (NanosDSM) offers an everything-shared SDSM. We understand as an everything-shared SDSM, the one that complies with the following two conditions:

- **The whole address space is shared**. This issue is very important because it reduces the stress placed on the file system and the administration. If the code and libraries are shared, we only need to have them in the initial node (where we start the application) and the rest of nodes will fault the pages and get them. It is also important to have shared stacks because they will be needed if we want to have several levels of parallelism or some kind of parallelism within functions when a local variable becomes shared among all threads.
- **The system offers a sequential semantic**. This semantic guarantees that any modification done in any node is visible to any other node immediately (as would happen in a hardware shared memory system).

When these two conditions are full filled, a SDSM behaves in the same way than a hardware shared memory and thus the applications that run on a real shared-memory machine will run with no modification in our system.

Managing sequential consistency

In order to offer a sequential semantic, we have implemented a single-writer multiple-readers coherence protocol in NanosDSM. Any node willing to modify a page has to ask permission to the master of the page, which will take care that only one node has write permission for a given page. It will also invalidate the copies of the page to be modified that are located in the other nodes.

In order to avoid overloading any node with master responsibilities, we can migrate masters to any node. The current policy is that the first node to modify page is the master of that page. If nobody modifies it (a read only page), the node where the application was started will behave as the master for that page.

Support to allow cooperation with higher levels

The most important support consists on offering up-calls. This mechanism allows the application (the OpenMP runtime in our case) to register a memory region, which means that NanosDSM will notify the higher level whenever a page fault occurs within this memory region. The mechanism to notify this page faults consists of executing the function that was passed as a parameter when registering the region. As this function is part of the application, it allows the higher level to know what is happening at the NanosDSM level, which is normally transparent. Later in this paper, we will present mechanisms that use these up-calls to build the cooperation between the OpenMP runtime and NanosDSM.

NanosDSM also needs to offer the possibility to move pages from one node to another (presend) and to invalidate pages when requested by the application or runtime. A more detailed description of these mechanisms will also be presented later in this paper.

It is important to keep in mind that these mechanisms are not thought to be used by regular programmers, but by runtime implementators, compiler developers, etc. These mechanisms should be transparent to regular applications.

Communication

Another important component of NanosDSM is the communications subsystem. It is mainly used to move pages from one node to another. It also provides an interface to the application (the OpenMP runtime in our case) to implement specific message-based communications with the goal to avoid the much more costly page faults. Later in the paper, we will describe how

the OpenMP runtime uses this communication subsystem for thread creation and synchronization.

Usability

Although in this paper we focus on running OpenMP applications on top of NanosDSM, we have also tested other shared-memory programming models such as Pthreads. We have been able to execute unmodified pthread applications on top of our system. This portability is achieved because our system is a real shared-memory system and not an SDSM with relaxed consistency. In the later case, we would have needed to modify the pthread applications to fit the relaxed model.

3.2 Nanos OpenMP runtime

In our environment, OpenMP applications are parallelized using the NanosCompiler [3, 4]. This compiler understands OpenMP directives embedded in traditional Fortran codes, such as the NAS benchmarks 2.3 [8] and generates parallel code. In the parallel code, the directives have triggered a series of transformations: parallel regions and parallel loop bodies have been encapsulated in functions for an easy creation of the parallelism. Extra code has been generated to spawn parallelism and for each thread to decide the amount of work to do from a parallel loop. Additional calls have been added to implement barriers, critical sections, etc. And variables have been privatized as indicated in the directives.

Nthlib [11, 10] is our runtime library supporting this kind of parallel code. NthLib has been ported to several platforms, including Linux/Pentium, Linux/IA64, IRIX/MIPS, AIX/POWER and SPARC/Solaris. We are currently working with the Linux/Pentium version in order to support a distributed memory environment.

Following our philosophy, the first try was to run NthLib as a shared library in the NanosDSM environment. After that was achieved (with terrible performance), we started the adaptation of the services in NthLib to take advantage of the message layer in NanosDSM.

As a result, only three aspects of NthLib were finally changed: The way the parallelism is spawned, the implementation of barriers and spin locks.

NthLib spawns parallelism using an abstraction called work descriptor. A thread sets up a work descriptor an it provides the other threads with it. A work descriptor contains a pointer to the function to be executed in parallel and its arguments. Usually, the work descriptor is set up in a shared memory area. In the NanosDSM implementation, the work descriptor is set up in a local memory area and then it is send through the message queues to reach the other threads. This solution avoids any page fault while spawning parallelism.

NthLib joins the parallelism using barriers. In its simplest form, a barrier contains the number of threads participating and the number of threads that have reached it. The threads arriving spin-wait until both number are the same. After that, they continue the execution. In NanosDSM, the same functionality is implemented through messages. All threads send a message to the master thread, and a message is sent back to the threads when the master detects that all the threads have reached the barrier. This way, there are no page faults when the application reaches a barrier.

The last tool modified are the spin locks. Spin locks are used both internally by NthLib to protect shared structures and by the parallel application when using critical regions. They are also implemented on top of the message system in NanosDSM.

In addition to that, all the shared data structures in NthLib are the same that in shared-memory implementations, except that they have been padded to page boundaries in order to avoid false sharing among them.

4. Our approach: Cooperation between runtimes

As we have already mentioned, our approach does not consist on modifying the behavior of the application nor the semantics of the SDSM software, but to encourage the cooperation between the OpenMP runtime and the SDSM software. In this section, we present the three kinds of cooperations we have already implemented and tested. it is important to notice that these are not the only possibilities, but the ones we have found necessary for our goal.

4.1 Boundaries alignment

The problem: Parallelizing a loop is a widely used technique. The idea is to divide the number of iteration

among the processors that will execute them in parallel. On a SDSM system, if this distribution does not take into account the page boundaries, we may have several processors writing on the same page causing false sharing and thus degrading the performance.

The solution: As most parallel loops are executed more than once, our proposal consists of scheduling the iteration in two steps. In the first execution of a parallel loop, the runtime starts with an static scheduling of the iterations (where all iterations are evenly distributed among all processors) and then learns which iterations access to which pages. Once this is known, the runtime reschedules the iterations avoiding the sharing of a page among two processors. This mechanism has some overhead the first time the loop is executed, but the benefits are then seen in all further executions of the loop.

How is the solution implemented: To compute the new schedule of iterations we follow these steps:

1. Register the memory regions where writes are done (using the up-call mechanism). We only care about write areas because they are the important ones for page alignment. Read pages can be replicated in all nodes that need them.

2. When a page fault occurs, the SDSM sends an up-call, and the OpenMP runtime checks if the address is the first one in the page. In this case, it marks that the current iteration corresponds to the beginning of a page. Otherwise, it does nothing.

3. Once each node has its list of iterations that correspond to the beginning of a page, they send them to the master, who will do the redistribution taking into account the list of iterations and the time spent by each thread. We have to note that these times include the page faults and thus may not correspond to the reality. For this reason we have to do the task several times till it becomes stable (repeat steps 1 to 3).

This algorithm generates a new schedule that is then reused every time the loop is executed. It could also be used by the parallel loops with the same characteristics.

The modules in the Nanos OpenMP runtime that take part into the alignment mechanism are presented in Figure 1, connected by a solid line.

This mechanism does the best possible load balance taking into account the page granularity and it adds little overhead.

4.2 Presend

The problem: In order to overlap data movement with computation, we need to know which pages will be needed by which nodes and when they will be needed. Prefetching, the traditional solution, can easily detect the pages, but not the exact time when the data movement will be best done without interfering with the application.

The solution: Our solution is to allow a cooperation between the runtime and the SDSM, who will actually do the presend. The idea is to detect the end of a loop and send the pages that each node has to the nodes that will need them in the next loop. As the work is normally a little bit unbalanced (specially if we align boundaries), we can start sending pages from one node while others are still computing. The only remaining question is to know if there is enough time to send the pages between loops. We will show that in all our experiments, there is enough time to do it.

How is the solution implemented: To compute the list of pages that have to be copied when presending pages, we follow these steps:

1. Learn the sequence of loops in the application to be able to know which loop comes after the current one.

2. Register the memory regions that are accessed by the parallel loop (note that in this case regions that are read are also important, not like in page alignment where write regions where the only ones to check).

3. Each thread keeps a list of the page faults it has generated for each loop (using the up-call mechanism) and sends it to the master.

4. The master makes a new list with the pages that each node has that should be sent, once the loop is over, to which nodes. For performance reasons, if more than one node have a page that another one will need, all nodes holding the page will have this page in their list of pages to send. In the execution, only the first one to request the copy will

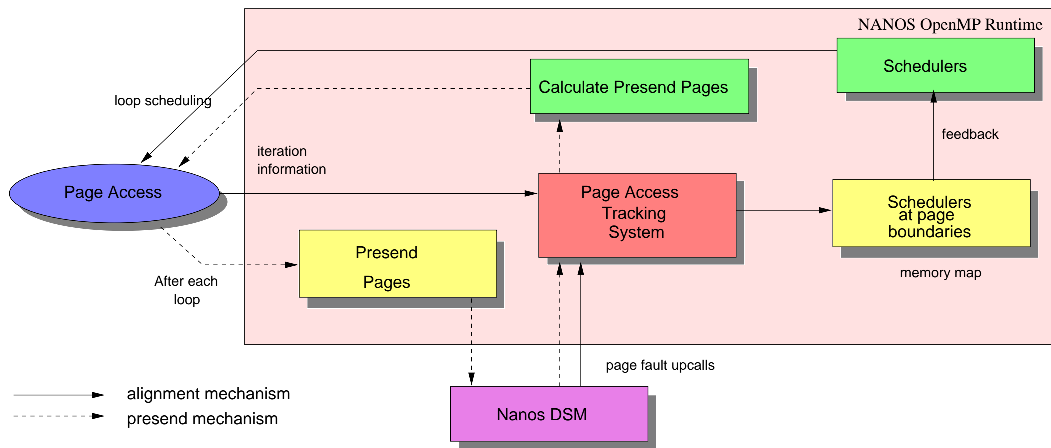

Figure 1. Components that take part in the alignment and presend mechanism

actually do it. With this mechanism we guarantee that pages are copied as soon as possible.

5. Once the thread has this list back, whenever it finishes a loop, it send the pages specified in the list using the presend mechanism implemented in the NanosDSM.

The modules that take part into the presend mechanism are presented in Figure 1, connected by a dashed line.

4.3 Preinvalidation

The problem: A very similar problem consists on invalidating the copies of a page once a node wants to modify them (remember our SDSM implements a sequential semantic). This task is also time consuming and it would be desirable to be able to overlap it with the computation as we do with presends.

The solution: Our approach is very similar to the one presented for presends. When we detect which nodes will need a page, we also detect if it will need it for writing. If this is the case and a node that holds the page will not need the page, then we invalidate our copy and inform the page master that we do not have a copy anymore. Hopefully, when the node wants to write the page, it will be the only one holding it as all other nodes will have preinvalidated it and thus it will be able to write it with no extra overhead.

How is the solution implemented: The mechanism used is exactly the same as in the presend but taking into account the page writes to invalidate the pages a node has that will be written by other nodes in the next loop.

5. Methodology

5.1 Benchmarks

In order to test our proposal we have executed three standard OpenMP applications. Two of them are NAS benchmarks [8] (EP and CG) and the third one is the Ocean kernel from the Splash2 benchmark suite [14, 17].

The EP benchmark kernel

This kernel generates pairs of Gaussian random deviates according to a specific scheme. This is a really parallel benchmark, all the data in the loop is private and it finally does a reduction.

Ocean

The Ocean application studies the large-scale ocean movements based on eddy and boundary current. It takes a simplified model of the ocean based on a discrete set of points equally spaced and simplified again as a set of 2D point planes. In this situation, it solves a differential equation via a finite difference method using a Gauss-Seidel update, computing a weighted average for each point based on its 4 neighbors. And it repeats this update until the difference for all points is

```
!Weighted average computing

        diff = 0.0;
C$OMP   PARALLEL DO PRIVATE (i,j, tmp)
C$OMP&   REDUCTION (+:diff)
        do j = 2, n+1
           do i = 2, n+1
              tmp = A(i,j)
              A(i,j)=0.2*(A(i,j)+A(i,j-1)
                     +A(i-1,j)+A(i,j+1)+A(i+1,j))
              diff = diff + abs(A(i,j) - tmp)
           enddo
        enddo
```

Figure 2. Main parallel loop found in Ocean

less than some tolerance level. The main parallel loop of this benchmark can be seen in Figure 2.

The CG benchmark kernel

The CG benchmark kernel uses a conjugate gradient method to compute an estimate to the largest eigenvalue of a symmetric sparse matrix with a random pattern of nonzeros. The problem size of the benchmark class depends on the number of rows (na) of the sparse matrix and the number of non-zero elements per row (nz). We use the classes A and B as distributed in the NAS benchmarks suite for our experiments.

This kernel have the following four consecutive parallel loops: i) matrix-vector product, ii) dot-product, iii) AXPY/Dot-product combination and iv) axpy.

5.2 Testbed

All the experiments presented in this paper have been run in two different clusters: Kandake and Crossi. Table 1 presents their characteristics. For availability reasons, we have been able to execute them on as much as 7 nodes.

Table 1. Platforms used in our tests

	Kandake	Crossi
Nodes	8	24
Available nodes	6	7
Processors per node	2	2 (Hyperthreaded)
Processor speed	266MHz	2.4GHz
RAM per node	128Mbytes	2Gbytes
Network	Myrinet	Myrinet

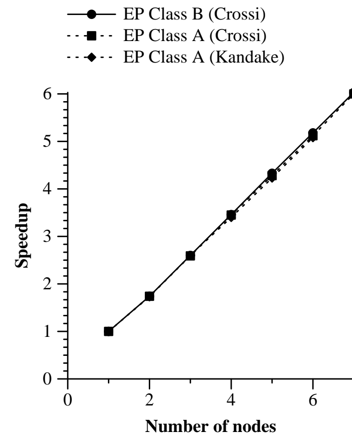

Figure 3. Speedups obtained by EP

6. Performance results

6.1 EP

As we have mentioned while describing the benchmark, this is the best possible case for any kind of SDSM. It shares no pages between the different nodes and thus a SDSM does not penalize its executions except for the first copy of the data.

As this benchmark does not modify shared data, page alignment does not make sense and thus our mechanism detects it and maintains the original static schedule. Regarding the presend mechanism, there is nothing we can presend, because the parallel loop is executed only once and after its execution there is no exchange of data among the nodes.

Figure 3 presents the speedups obtained by this benchmark in both machines and using both sizes (classes A and B). In Kandake we only run class A because class B was too big (this happens with all benchmarks).

Observing the graph, we can observe that, as expected, the speedup obtained is quite good. A perfect speedup it not achieved due to the reductions that need to be done at the end of the loop and because the schedule used (STATIC) is not fully balanced. Some nodes have some more work to do than others.

Although we have not been able to do a direct comparison of our results with other SDSM system, we can tell that similar results were obtained by Müller et al. using a relaxed consistency SDSM [5].

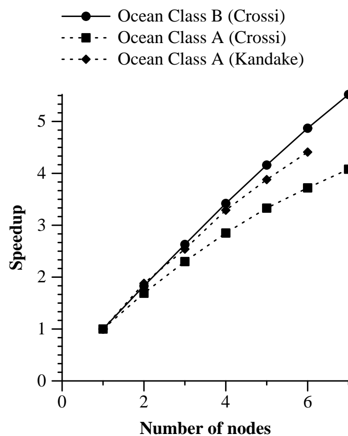

Figure 4. Speedups obtained by Ocean

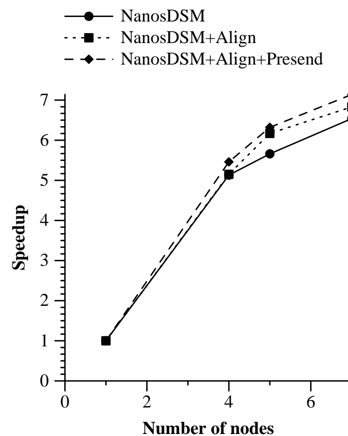

Figure 5. CG class B on Crossi

6.2 Ocean

In this benchmark, we have a potential horrible situation for a SDSM, which is a true sharing among nodes. Many different cells in the array are read by a node and written by another. This implies that there are no boundaries because no matter how we split the computation, some elements on one side will be written by the nodes assigned to the other side. Once again our mechanism detects it and does not align. In a similar way, the presend mechanism is not useful because pages are either only accessed by a node, or are read and written by several nodes. This last case cannot be taken into account by the presend as we may invalidate a page that may be written latter on in the same node. Our granularity is the loop and we cannot use a smaller granularity within the loop. In this case, the OpenMP runtime also avoids to do any presend.

Figure 4 presents the speedup obtained by this benchmark, and once again we can see that they are very good ones. Although there is potential true sharing among nodes, when a node needs a page, the other nodes are not using it. This behavior is quite frequent due to the order in the iterations. It is clear that if the granularity becomes too small and the nodes conflict in the true sharing area, then the performance will be degraded significantly. Nevertheless, the results presented here show that in all tested cases, the speedup obtained is good enough.

6.3 CG

The last benchmark presented in this paper is the CG. This benchmark does not run efficiently on an everything-shared SDSM if there is no cooperation between the layers. The most important reason is that the elements of a vector are read by some nodes in a loop and written by different nodes in another loop. This situation is perfect for the presend and alignment mechanisms.

In order to present a more detailed study of the behavior of this benchmark, we present three different graphs. The first one (Figure 5) shows the behavior of this benchmark class B on Crossi. Then, we present the behavior of the same benchmark in a smaller class (A) on the same machine (Figure 6). This will help us to see the effects of the different proposals when the granularity is smaller and thus will give us an idea of how well this application will scale. Finally, we re-execute CG class A on Kandake and compare its speedup with the one obtained by TreadMarks (Figure 7). This experiment will show us how well our automatic mechanism does compared to a version specifically written for TreadMarks and using a relaxed-consistency SDSM.

The first experiment (CG class B on Crossi), shows that a good speedup can be achieved (Figure 5). It also shows that as the number of nodes grows, the alignment and presend mechanisms become more important. This makes sense because as we increase the number of nodes, we also increase the number of boundaries and the number of pages that have to be

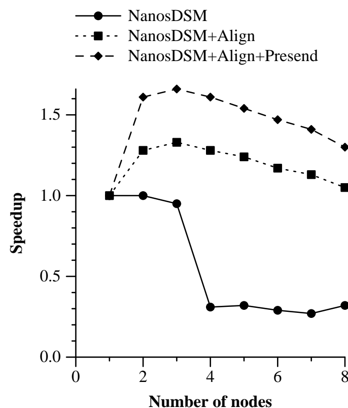

Figure 6. CG class A on Crossi

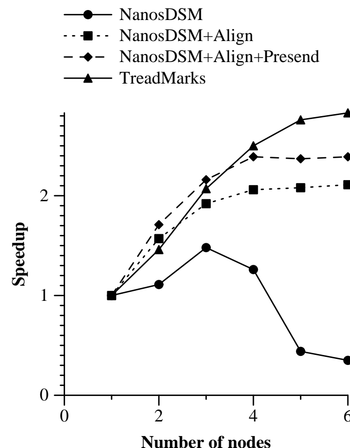

Figure 7. CG class A on Kandake

copied/moved.

When executing the same benchmark but using a smaller dataset on the same machine (Figure 6), we clearly see that the alignment and the presend are necessary if some speedup is to be achieved. We can also see that this speedup stops when more than 3 nodes are used. The reason behind this behavior is the presence of two variables *alpha* and *beta*, which are written in sequential and read in parallel, producing a big contention. This could be solved if the compiler detects this situation and informs the other threads with the written value avoiding any page fault (and we are currently working on it). Even though the load balance has improved the performance a lot, as we divide iterations on a page basis, a given node has all the iterations that modify a page or none. This limits the possibility of load balancing and thus if very few pages are used, a good schedule will be impossible. For instance, if the dataset has as many pages as nodes plus one, we will have all nodes with the iteration of one page and one node with the iteration of 2 pages, which means that it will have twice as many iterations (and thus work) than any other node.

Finally, we repeated the execution of the benchmark on Kandake (Figure 7). The objective was to compare our speedup with the one observed when the "same" application is run on TreadMarks. We could only test the TreadMarks version on this machine because we only have a license for this machine.

The first thing we can see is that it has a similar behavior (speedup wise) than the execution on Crossi. We also see that this speedup stops growing after 4 nodes and the reason is also the same as in the previous experiment.

When comparing our behavior with the one achieved by TreadMarks, we can see that we do as well as they do but using a relaxed-semantic SDSM. In addition, we should remember that the CG executed in TreadMarks is not the OpenMP version, but a version specially coded for TreadMarks. Finally, we can also observe that TreadMarks continues to increase its performance when the number of nodes grows beyond 4. Observe also that, even when using TreadMarks and relaxed consistency, the speedup is limited to 2.5 on 4 processors, confirming the point about the small size of the class A of CG.

7. Conclusions

We have presented some applications that have achieved very good speedups. The ones that did not achieve it have been compared to the execution on top of other SDSM systems such as TreadMarks observing a very similar behavior.

Finally, we have also detected the main limitation in our approach. As we have to distribute work on page bases, when the data needed by each nodes reached the size of just a few pages, then our alignment mechanism will not be able to build a good load balance and thus the performance will be penalized. On the other hand, we will be able to run our applications on a system much more similar to what we have on a hardware shared memory system.

Our future work is to evaluate this proposals using

more benchmark applications, both from the NAS Parallel Benchmarks [8] and the SPEComp 2001 [2]. The experience taken from them will then be used to improve the execution environment with new proposals oriented to solve the performance problems that we could find with them.

8 Acknowledgments

We thank Ernest Artiaga, Josep Maria Vilaplana and Sebastia Cortes, for their help in developing part of the software presented in this paper. We also thank Prof. M. Müller for letting us use one of his clusters and for his help.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. *Lecture Notes in Computer Science*, 2104, 2001.

[3] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In *In Proceedings of the 1st European Workshop on OpenMP, Lund, Sweden*, October 1999.

[4] M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism Exploitation in OpenMP. *Concurrency: Practice & Experience*, (12):1205–1218, October 2000.

[5] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster. In *Workshop on OpenMP Applications and Tools (WOMPAT'02)*, August 2002.

[6] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *Proceedings of the High Performance Computing and Networking (HPCN'99)*, volume LNCS 1593, pages 463–472, Amsterdam, Netherlands, Apr. 1999. Springer.

[7] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[8] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[9] Y. Kee, J. Kim, and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Supercomputing 2003 (SC'03)*, November 2003.

[10] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta. Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors. In *13th International Conference on Supercomputing (ICS'99)*, june 1999.

[11] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A Library Implementation of the Nano-Threads Programming Model. In *Euro-Par, Vol. II*, pages 644–649, August 1996.

[12] M. Sato, H. Harada, and Y. Ishikawa. OpenMP Compiler for a Software Distributed Shared Memory System SCASH. In *WOMPAT2000*, July 2000.

[13] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *EWOMP '99*, pages 32–39, Sept. 1999.

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

[15] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proc. of the USENIX Windows NT Workshop*, 1997.

[16] L. Whately, R. Pinto, M. Rangarajan, L. Iftode, R. Bianchini, and C. L. Amorim. Adaptive Techniques for Home-Based Software DSMs. In *13th Symposium on Computer Architecture and High Performance Computing*, September 2001.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.