Running OpenMP applications efficiently on an everything-shared SDSM

J.J. Costa * T. Cortes X. Martorell E. Ayguade J. Labarta

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona

Contact Author: J.J. Costa

Mailing address:

Departament d’Arquitectura de Computadors
UPC - Campus Nord (C6-E201)
c/ Jordi Girona 1-3
08034 Barcelona
Spain

Phone: +34 934011650
Fax: +34 934017055
e-mail: jcosta@ac.upc.edu

1 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
* Corresponding author

Email addresses: jcosta@ac.upc.edu (J.J. Costa), toni@ciri.upc.es (T. Cortes), xavim@ac.upc.edu (X. Martorell), eduard@ac.upc.edu (E. Ayguade), jesus@cepba.upc.es (J. Labarta).

Preprint submitted to JPDC 30 August 2005
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 present are demonstrated as very useful to overcome the limitations of the current SDSM systems.

Key words: OpenMP, Software Distributed Shared Memory, Scalability, Adaptability, Cooperation

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 (as in Figure 1 (a) that represents the address space of a real shared memory architecture where the code, data and stacks of the threads are shared). In order to improve performance, all implementations change the semantics of the shared memory to something easier to implement [1–5]. 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. An illustration of the resulting address space is shown in Figure 1 (b) where three nodes have their private address spaces and a specific memory area is shared.
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. This is not an approach with future.

Our proposal (see Figure 1 (c)) 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.

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. [6], OpenMP on the SCASH system [4,7], and ParADE [8].

Hu et al. [6] 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 [4,7]. This approach uses the Omni compiler [9] 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 [8] is implemented on a lazy release consistency (HLRC[5]) 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 Main philosophical ideas

As we have already mentioned, our environment (NanosDSM) offers an everything-shared SDSM. Actually, this is just one issue that differentiates our work from
previous work done. In this section we will try to explain the main issues in our design that are different to what has been done so far.

- **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.

- **Adapt parallel structure instead of implementing fancy consistency models.** NanosDSM 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). This approach has normally not been followed for performance reasons. We propose to use it and solve the performance problem by adapting the structure of the parallelism as will be seen later. This changes will be done by the runtime and will be completely transparent to the programmer that will only see OpenMP applications.

- **Sender initiated transactions instead of demand driven.** In order to improve performance, issues such as prefetching are normally implemented. These mechanisms are always started from the node that will need the information, and this may be done at wrong time for the node that has the information. We propose to design mechanisms that are initiated by the sender when it knows to be the right time to send the information.

- **Cooperation between layers.** Finally, but probably the most important issue, is that all layers have to cooperate in order to make the execution efficient. For instance the SDSM layer has to inform about some events to the OpenMP runtime, while the OpenMP runtime has to inform the memory layer about structure and semantics of the application.

How these philosophical ideas are finally implemented will be described in the rest of this paper, but by now it is important to notice that our system will offer a shared memory equal to a hardware shared memory (as far as semantics are concerned) and thus applications that run on a real shared-memory machine will run, with no modification, in our system.
3.2 NanosDSM: An everything-shared SDSM

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 and at any time. The current policy is that the first node to modify a 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, shown in Figure 2, 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. At this moment, we only need to know that a call is offered for the runtime to specify which pages have to be moved from which nodes to which other nodes. The NanosDSM also offers a call for a node to voluntarily invalidate a page it has (preinvalidation).

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

Communication

Another important component of NanosDSM is the communications subsystem. Besides providing a mechanism 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.

On the one hand, we have implemented what we call remote queues. These queues are a communication mechanism used by the runtime to send information from one node to another. In the shared memory version, this information is placed in a queue, but as this would be too costly due to page movements, we send a message with the information to the receiver’s queue using sockets. It is important to notice that this mechanism is used to improve the efficiency of the runtime, but that user or OpenMP applications will not see it.

On the other hand, if the runtime has to write a few bytes to a page that knows for sure in which node it resides, then we have also implemented a direct write mechanism that allows a node to modify a few bytes on a remote page without having to fault it. Once again, this mechanism is only available for the runtime that knows how to use it without loosing the sequential consistency, but not for user applications.

Later in the paper, we will describe how the OpenMP runtime uses these communication mechanisms for thread creation and synchronization.

Other implementation details

The current version of NanosDSM is able to run on both TCP and UDP. In the later case (UDP), it executes a control mechanism that is able to handle UDP packet losses. In order to increase the potential usage of NanosDSM, we have also ported it on top of PadicoTM [10,11], which is a layer designed and implemented at IRISA with the objective of decoupling the network interface from the higher layers.
The current version is also thread safe, which means that can be run in SMP clusters and take advantage of the different processors that each SMP has.

Finally and 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.

3.3 Nanos OpenMP runtime

In our environment, OpenMP applications are parallelized using the NanosCompiler [12,13]. This compiler understands OpenMP directives embedded in traditional Fortran codes, such as the NAS benchmarks 2.3 [14] 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 [15,16] 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 first milestone was achieved (with terrible performance results, as expected), we started the adaptation of the services in Nthlib to take advantage of the message layer in NanosDSM.

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

Nthlib spawns parallelism using an abstraction called work descriptor. A thread sets up a work descriptor and 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 sent through the message queues described in previous section to reach the other threads. This solution allows to distribute work among different nodes avoiding any page fault while spawning parallelism. It is true, that this mechanism would not allow a simple or efficient implementation of work stealing scheduling policies, but as locality is one of the most important issues, we believe that such policies would not be adequate regardless of the work distribution mechanism implemented.

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.

In the same way as barriers, spin locks have been reimplemented for this new environment. 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.

Generally, reductions are done directly on the reduced variable, or to avoid cache problems in a vector that is finally reduced by the master. In a SDSM environment, this cannot be implemented directly in any of these ways as the page with the variable or the vector would produce too many page faults. To solve this problem, we implement the vector version of reductions, but instead of writing normally, we use the direct write mechanism described in the NanosDSM section that allows a node to write on a remote page (via message passing) without faulting the page. This mechanism speeds up reductions a lot avoiding unnecessary page faults.

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.

3.4 Nanos compiler

The last piece in this environment is the compiler. Most of the time, it works regardless of the fact that the application will be executed on a SDSM, but in order to make applications run efficiently there is only one issue that have to be taken into account: the alignment of vectors. In order to avoid false sharing, vectors (and matrices) should be aligned to page boundaries and should not share pages with other data. This modification in the compiler is very easy to
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 iterations 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. This problem is shown in Figure 3 (a), where the distribution of iterations between two processors produces a sharing with the $p$ page.

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, as shown in Figure 3 (b). 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 4.

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. This is specially important when using a single writer protocol like us. This problem can be better seen in the Figure 5 where two threads appear. One of them prefetchs the page while the other node is still writing it. This causes a ping-pong effect moving the page between the two threads unnecessarily.

**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 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 sends 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 6.

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.
Table 1
Detailed parameter values for the different benchmark classes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Param.</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>m</td>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>CG</td>
<td>na</td>
<td>14000</td>
<td>75000</td>
</tr>
<tr>
<td></td>
<td>nonzer</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Ocean</td>
<td>n + 2</td>
<td>1024</td>
<td>5000</td>
</tr>
</tbody>
</table>

4.4 Integration of all techniques

As the reader may have guessed, the process of finding the border to be aligned cannot be done at the same time as the learning of which pages have to be present and preinvalidated. The effects of one of the techniques affects the results of the other. In order to solve this problem, the learning is done in two steps. In the first steps, presending and preinvalidation are deactivated and only border alignment is active. Once the borders have been found and are stable, the presend and preinvalidate mechanisms are activated.

This two step mechanism has not produced any problem in the presented experiments because the borders were found in one or two iterations.

5 Methodology

5.1 Benchmarks

In order to test our proposal we have executed three standard OpenMP applications. Two of them are NAS benchmarks [14] (EP and CG) and the third one is the Ocean kernel from the Splash2 benchmark suite [17,18]. The size of the different classes used are specified in table 1, where the different parameters of each benchmark is detailed by each problem class.

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.
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 less than some tolerance level. The main parallel loop of this benchmark can be seen in Figure 7.

**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 2 presents their characteristics. For availability reasons, we have been able to execute them on as much as 7 nodes.
Table 2
Platforms used in our tests

<table>
<thead>
<tr>
<th>Kandake</th>
<th>Crossi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>8</td>
</tr>
<tr>
<td>Available nodes</td>
<td>6</td>
</tr>
<tr>
<td>Processors per node</td>
<td>2</td>
</tr>
<tr>
<td>(Hyper threaded)</td>
<td></td>
</tr>
<tr>
<td>Processor speed</td>
<td>266MHz</td>
</tr>
<tr>
<td>RAM per node</td>
<td>128Mbytes</td>
</tr>
<tr>
<td>Network</td>
<td>Myrinet</td>
</tr>
</tbody>
</table>

6 Performance results

The results of the different benchmarks will be shown in this section. The speedups are calculated using as the base case the sequential time of the original OpenMP application. Sequential time means that the original OpenMP code is compiled using any available Fortran compiler without OpenMP support.

All the execution times are shown in appendix A.

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 8 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 is 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.
Fig. 8. Speedups obtained by EP

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 [7].

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 later 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 9 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 10) 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 11). 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 12). 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 10). 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 copied/moved.

When executing the same benchmark but using a smaller dataset on the same machine (Figure 11), 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 12). 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 DSM. 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.

6.4 Shared-everything overhead

The NanosDSM is a shared-everything DSM, a consequence is that the pages containing the code section of the application are faulted by the remote nodes instead of being moved using an \textit{rsh} command. From a functional point of view, this approach is interesting because it doesn’t need any further action rather than launch the OpenMP application at the master node. All the remote nodes will get the code without any user interaction.

On the other side, the overhead of this approach must be really small in order to maintain the performance.

In order to calculate this overhead, the different pages containing the code section of the different benchmarks have been identified. These code pages are faulted only once by all threads, and they are faulted at the beginning of the execution time.

The percentage of time solving these page faults are measured in table 3 using a pessimistic average time per page fault of 2 milliseconds, and it shows that this overhead is really small compared to the total execution time.
Table 3
Measured overhead of shared-everything SDSM.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Pages</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A</td>
<td>15-16</td>
<td>0.033</td>
</tr>
<tr>
<td>EP.A</td>
<td>10</td>
<td>0.003</td>
</tr>
<tr>
<td>Ocean.A</td>
<td>9</td>
<td>0.036</td>
</tr>
</tbody>
</table>

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 node 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 [14] and the SPEComp 2001 [19]. 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


Juan Jose Costa is a Ph.D. student at Universitat Politecnica de Catalunya (Barcelona – Spain) since 2002. He obtained his M.S. degree in computer science at the same university. His main interests are distributed shared memory and cluster computing. Currently, he is doing his research at the CEPBA-IBM Research Institute.

Toni Cortes is an Associate Professor at Universitat Politecnica de Catalunya (Barcelona – Spain) since 1999. He obtained his M.S. and Ph.D. degrees in computer science at the same university, and is currently the coordinator of the single-system image technical area in the IEEE Task Force on Cluster Computing (TFCC). His main interests are cluster computing and parallel I/O. Dr. Cortes has also been working on several European industrial projects such as Paros, Nanos, and POP. As a result of his research, both in basic research and technology transfer, Dr. Cortes has published more than 35 papers in international journals and conferences, has published 2 book chapters and has edited a book on parallel I/O (“High Performance Mass Storage and Parallel I/O: Technologies and Applications”, IEEE press). He also acts a program committee for many international conferences and journals.

Xavier Martorell received the Engineering degree in Computer Science in 1991 and the Ph.D. degree in Computer Science in 1999, both from the Technical University of Catalunya (UPC), Spain. Since 1992 he has been lecturing on operating systems and parallel run-time systems. Currently, and since 2001, he is an associate professor of the Computer Architecture Department at UPC. His research interests cover the areas of operating systems, run-time systems and compilers for high-performance multiprocessor systems. He has participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs. Currently, he is a visiting scientist at the IBM T.J. Watson Research Center.

Eduard Ayguade received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the Technical University of Catalunya (UPC), Spain. Since 1987 he has been lecturing on computer organization and architecture and optimizing
compilers. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. His research interests cover the areas of parallelizing compilers and runtime systems for high-performance multiprocessor systems. He has published more than 125 papers in these topics and participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs. Currently he is one of the project managers of the CEPBA-IBM Research Institute.

**Jesus Labarta** is full professor at the Computer Architecture department of UPC (technical University of Catalonia) since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, metacomputing tools and performance analysis and prediction tools. He has leaded the technical work of UPC in 15 industrial R+D projects. Since 1995 he is director of CEPBA where he has been highly motivated by the promotion of parallel computing into industrial practice, and especially within SMEs. In this line he has acted as responsible of three technology transfer cluster projects where his team managed 28 subprojects.

His major directions of current work relate to performance analysis tools and OpenMP. The most representative result of his work on tools are Paraver and Dimemas. The work on OpenMP and OS scheduling is reflected in the NANOS platform. He actively participates in the OpenMP ARB Futures Committee proposing and evaluating the potential of future extensions to the standard.

Since 2000 he is strongly committed in carrying out and promoting productive research co-operation with IBM as part of the CEPBA-IBM
## A Benchmark’s execution time

Table A.1
Execution time (seconds) for the Benchmark CG Class A at kandake compared with TreadMarks.

<table>
<thead>
<tr>
<th></th>
<th>OnlyDSM</th>
<th>Align</th>
<th>Presend</th>
<th>TreadMarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>89.0</td>
<td>89.0</td>
<td>89.0</td>
<td>89.0</td>
</tr>
<tr>
<td>2</td>
<td>79.85</td>
<td>56.7</td>
<td>52.15</td>
<td>60.93</td>
</tr>
<tr>
<td>3</td>
<td>60.13</td>
<td>46.39</td>
<td>41.16</td>
<td>43.04</td>
</tr>
<tr>
<td>4</td>
<td>70.46</td>
<td>43.29</td>
<td>37.27</td>
<td>35.53</td>
</tr>
<tr>
<td>5</td>
<td>203.8</td>
<td>42.74</td>
<td>37.36</td>
<td>32.19</td>
</tr>
<tr>
<td>6</td>
<td>255.75</td>
<td>42.17</td>
<td>37.20</td>
<td>31.47</td>
</tr>
</tbody>
</table>

Table A.2
Execution time (seconds) for the benchmark CG Class A at Crossi.

<table>
<thead>
<tr>
<th></th>
<th>OnlyDSM</th>
<th>Align</th>
<th>Presend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>2</td>
<td>7.03</td>
<td>5.47</td>
<td>4.34</td>
</tr>
<tr>
<td>3</td>
<td>7.37</td>
<td>5.25</td>
<td>4.21</td>
</tr>
<tr>
<td>4</td>
<td>22.66</td>
<td>5.47</td>
<td>4.35</td>
</tr>
<tr>
<td>5</td>
<td>22.13</td>
<td>5.65</td>
<td>4.54</td>
</tr>
<tr>
<td>6</td>
<td>24.19</td>
<td>6.00</td>
<td>4.76</td>
</tr>
<tr>
<td>7</td>
<td>26.28</td>
<td>6.22</td>
<td>4.97</td>
</tr>
<tr>
<td>8</td>
<td>22.20</td>
<td>6.67</td>
<td>5.38</td>
</tr>
</tbody>
</table>
Table A.3
Execution time (seconds) for the benchmark CG Class B at Crossi.

<table>
<thead>
<tr>
<th>Seq</th>
<th>OnlyDSM</th>
<th>Align</th>
<th>Presend</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1421.19</td>
<td>1421.19</td>
<td>1421.19</td>
</tr>
<tr>
<td>3</td>
<td>-.-</td>
<td>-.-</td>
<td>-.-</td>
</tr>
<tr>
<td>4</td>
<td>378.94</td>
<td>371.32</td>
<td>350.14</td>
</tr>
<tr>
<td>5</td>
<td>343.22</td>
<td>310.18</td>
<td>302.66</td>
</tr>
<tr>
<td>6</td>
<td>-.-</td>
<td>-.-</td>
<td>-.-</td>
</tr>
<tr>
<td>7</td>
<td>297.11</td>
<td>280.10</td>
<td>267.79</td>
</tr>
</tbody>
</table>

Table A.4
Execution time (seconds) for the benchmark EP.

<table>
<thead>
<tr>
<th>Seq</th>
<th>EP.A Crossi</th>
<th>EP.B Crossi</th>
<th>EP.A Kandake</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>101.81</td>
<td>381.05</td>
<td>532.7</td>
</tr>
<tr>
<td>3</td>
<td>58.63</td>
<td>-.-</td>
<td>307.89</td>
</tr>
<tr>
<td>4</td>
<td>39.28</td>
<td>-.-</td>
<td>205.17</td>
</tr>
<tr>
<td>5</td>
<td>29.5</td>
<td>117.64</td>
<td>156.81</td>
</tr>
<tr>
<td>6</td>
<td>19.9</td>
<td>-.-</td>
<td>104.81</td>
</tr>
<tr>
<td>7</td>
<td>16.98</td>
<td>78.08</td>
<td>-.-</td>
</tr>
</tbody>
</table>

Table A.5
Execution time (seconds) for the benchmark Ocean.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7.36</td>
<td>45.38</td>
<td>54.13</td>
</tr>
<tr>
<td>3</td>
<td>4.34</td>
<td>24.64</td>
<td>28.86</td>
</tr>
<tr>
<td>4</td>
<td>3.21</td>
<td>17.15</td>
<td>21.32</td>
</tr>
<tr>
<td>5</td>
<td>2.58</td>
<td>13.18</td>
<td>16.45</td>
</tr>
<tr>
<td>6</td>
<td>2.21</td>
<td>10.84</td>
<td>13.94</td>
</tr>
<tr>
<td>7</td>
<td>1.98</td>
<td>9.25</td>
<td>12.29</td>
</tr>
<tr>
<td></td>
<td>1.8</td>
<td>8.17</td>
<td>-.-</td>
</tr>
</tbody>
</table>