Reducing data access latency in SDSM systems using runtime optimizations

Javier Bueno[1], Xavier Martorell[1], Juan José Costa[1], Toni Cortés[1], Eduard Ayguadé[1], Guansong Zhang[2], Christopher Barton[2], and Raul Silvera[2]

[1]Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
c/ Jordi Girona, 31 - 08034
Barcelona (Spain)
{jbueno, xavim, jcosta, toni, eduard}@ac.upc.edu

[2]IBM Toronto Lab,
8200 Warden Avenue
Markham, ON, L6G 1C7, Canada
{guansong, kbarton, rauls}@ca.ibm.com

Abstract

Software Distributed Shared Memory (SDSM) systems offer a convenient way to run applications developed for shared memory systems on distributed systems with no changes to them. However, since SDSM systems add an extra layer of abstraction to the memory hierarchy, applications may suffer performance problems when running on top of them.

Our main research interest is to develop a set of compiler and runtime system techniques that widen the range of applications that can efficiently run on SDSM systems. Currently we are targeting OpenMP applications due to the ease of use this programming model provides.

In this paper we show the performance of a set of regular applications that perform well on our SDSM system. They were adapted from OpenCL codes provided by ATI, and re-written in OpenMP. When trying to exploit more complex applications with different data access patterns, we find more difficulties from a DSM system. As an example, we show the performance evaluation of the NAS MG benchmark, and two techniques we have developed to improve its data locality.

Our SDSM infrastructure is composed of NanosDSM, an everything-shared SDSM developed at the Technical University of Catalonia (UPC) and the Barcelona Supercomputing Center (BSC), and the IBM XL SMP Runtime to allow the execution of the OpenMP applications.

1 Introduction

Cluster systems have become an important platform for high performance computing, currently being the most widely used systems where large computation power is needed.

However, in order to fully get the benefits of distributed memory systems, a message passing programming model is required to fully exploit the parallelism that they offer.

Message passing programming models have a major drawback when compared to shared memory programming models - they add the burden of having to explicitly program the data movement between processors. This usually means having to rethink the whole application structure.

In contrast, shared memory programming models offer a friendlier programming model than the message passing model. In the case of OpenMP [6], applications can be easily parallelized by adding a few directives to the original code. Since it is an standard, currently most commercial compilers support OpenMP, making it very appealing as a parallel programming model.

Due to the advantages of OpenMP versus message passing paradigms, the possibility of running OpenMP applications on clusters has always been interesting. One of the ideas to achieve this, is having a software system that emulates a virtual address space distributed across different machines.

But the big disadvantage of this kind of systems is the overhead caused by this extra layer of virtualization, which limits the set of applications that can be efficiently run on top of such environments.

The main contribution of this paper is a set of techniques that allow applications to achieve better performance when running on SDSM systems. This has been the result of a project between the Universitat Politècnica de Catalunya and the IBM Toronto Laboratory that aims to run OpenMP applications efficiently using NanosDSM, our SDSM system, and the XL SMP runtime.

We have used several applications to measure the performance of our SDSM. The set of applications includes a *Matrix Multiply*, several OpenCL applications that have been ported to OpenMP and the MG from the NAS Parallel Benchmarks. These applications show different behaviors when running on our SDSM and thus offer a good insight into the benefits and problems of such systems.

With all the experience obtained from studying these applications we have developed two runtime optimizations that improve the locality of memory accesses when running on our SDSM system. These optimizations have been developed using the NAS MG benchmark as a study case, because MG was the application which had more problems when running on our system.

The rest of the paper is organized as follows: in section 2 we take a look at related work on SDSM systems. In section 3 we describe some features of the SDSM system that we have used in our project, NanosDSM. In section 4 we show how the integration of NanosDSM and the XL runtime has been possible. Section 5 discusses the experiences we found when running applications with our system. In section 6 we comment the techniques that have been developed to improve the performance of the NAS MG benchmark while running on our SDSM environment. In section 7 we describe the analysis done to evaluate the current performance of the system. Sections 8 and 9 conclude this paper, and present the future work.

2 Related work on SDSMs

There have been many research projects on SDSM systems, many of them being oriented to support OpenMP applications.

Usually, the main trend followed when designing a SDSM system is taking advantage of the relaxed consistency of the OpenMP memory model. This model specifies that the whole memory is shared, but write operations to memory are not required to be finished while execution continues, synchronization points define when the main memory is required to be up to date. The OpenMP *flush* directive serves as a explicit memory synchronization point, meaning that every memory operation issued before a *flush* has to be visible to all threads after the execution of that *flush*; memory operations after a *flush* are not allowed to start until it completes. This relaxed consistency has been believed to be needed in order to achieve a good performance in SDSM systems, thus many SDSM projects have implemented such memory model.

Another typical feature that has been implemented is limiting the shared memory area to a zone specified by the user. In addition, dynamic memory allocation in shared areas can not be done using the standard *malloc* or *free* calls, so new routines are needed to replace them in cases where dynamic memory is required to be shared. When using OpenMP, new directives might be needed to overcome this limitation. This might involve making changes in the source code of the parallel applications and extra effort by the compiler to support the new directives and features.

JIAJIA [12] is an example of SDSM system that uses a lock-based protocol for scope consistency, but it does not have explicit OpenMP support. HAP [8] is another SDSM that relaxes the memory model by using a Lazy Release Consistency. Brazos [5] also uses a scope consistency memory model and supports multithreading to take advantage of SMP servers.

SDSMs that have been designed to run OpenMP applications include the OpenMP translator [13], the SCASH system [9] and ParADE [14]. The OpenMP translator allows applications to run applications on top of TreadMarks [3], its main task is dealing with the limitations that the TreadMarks system has due to his implementation (limited shared area and relaxed consistency). The SCASH system uses a release consistency memory model with a multiple writers protocol to avoid false sharing, it requires the Omni compiler [10] to transform the applications to run on it. ParADE also uses the Omni compiler to run OpenMP applications, it is based on a lazy release consistency memory model with home migration and uses MPI as a communication system. Also, Intel has included SDSM support for OpenMP in its compiler suite. The runtime, called Cluster OpenMP [7], is based on the TreadMarks system. It adds the new directive *shareable* to specify data that will be shared through the SDSM mechanism, thus it requires some modifications in the source code of the applications to be executed with it. The compiler manages the dynamic memory allocation and the implications of having a lazy release consistency.

However, the limitations discussed previously are not really a requirement when designing a SDSM system, and everything-shared SDSMs with restrictive memory models do exist. As an example of such systems, we have the SDSM that is being used in the development of our project, NanosDSM. NanosDSM is an everything shared SDSM, meaning that the whole space address (global data, stacks, code) is shared. The system also provides sequential consistency as the memory consistency model, implying that memory operations are visible to other threads at the same time they are executed. Both features might seem as a performance killer for many applications, and of course the system has to pay some performance penalty for them, but there are optimization techniques that have been applied successfully on some programs, achieving a performance as good as other SDSMs with the restrictions seen above. Furthermore, both of these features ease the porting of applications to this system, since the view of the architecture provided is almost identical to a hardware shared memory environment.

3 NanosDSM

In this section we will take a closer look at the SDSM system that we have used in this project, NanosDSM.

NanosDSM is a part of the Nanos project [1], whose purpose is to support research on enabling the efficient use of high performance architectures. Currently, Nanos is composed of three main software components, the OpenMP runtime, *nthlib* [11], the OpenMP source-to-source Mercurium compiler for Fortran and C/C++ [2], and the SDSM runtime [4].

3.1 User view

NanosDSM is an everything shared SDSM that offers a sequential consistency memory model. Its characteristics are the same as the ones that any developer can expect while programming for a hardware shared memory environment using a threads library such as *pthreads*. With this, the NanosDSM API was designed to be similar to a regular threads library. We can classify the main features as follows:

Thread management Threads are created when the application starts, and sleep until the user assigns some code to them. A thread is identified by its node and the thread number within the node. Multi-threading on the same node is supported to take advantage of SMP nodes.

Synchronization directives NanosDSM offers a mutex-like variable type (called *lock*) as a more efficient alternative to standard shared memory implementations.

Support for thread specific data is not offered with explicit calls (like *pthread_{getspecific, setspecific}*) but the use of ELF TLS variables (using the *__thread* qualifier) is supported.

In addition, NanosDSM has some extra features to allow applications to be DSM-aware and ease the implementation of OpenMP runtimes that optimize the execution of parallel programs considering the underlying SDSM layer. This features include:

Pre-sends and pre-invalidations The application can explicitly request the transfer of memory pages to a given node. This can improve the performance of the applications by allowing communication to overlap the computation.

Pagefault callbacks A user function can be registered to be called when a page fault on a given page occurs. This is useful when an application wants to keep track of the movement of certain data.

Dynamic memory A call with similar behavior as *mmap* is implemented to offer the possibility of allocate dynamic shared memory from any thread of the application. In addition, the standard calls *malloc/free* are allowed to be executed in any thread.

Message passing Information can be sent directly to other threads through the network. This can sometimes be better for the application performance than using a solution based on shared memory.

NanosDSM has been implemented on i386 and PowerPC (32 and 64-bit) architectures running the Linux 2.6 operating system.

3.2 Implementation details

NanosDSM has two main components, a dynamic library and the *ndsm_server*. The library contains most of the services offered by NanosDSM and any parallel application must be linked to it to run. The *ndsm_server* is a daemon that runs on the nodes where remote threads of a parallel application will be executed. At the beginning of the execution it can be seen as an image of the application with all of its address space being not physically present in that node.

The memory consistency is kept at page level. In a given node, memory pages can be in three possible states during the execution:

RW readable and writable: read and write accesses are allowed. A page can only be in a RW state on one node at a given time, all other nodes have the page marked as invalid (I).

R readable: only read accesses to this page are allowed. Many nodes can have a page in this state at a given time but no nodes can have it in a RW state if an R state exists in a node.

I invalid: no accesses are allowed, meaning that an up-to-date version of the page is not physically present in the node.

These three states are similar to those defined by the MSI protocol. States are set using the *mprotect* system call. When a thread tries to access a page, but the current state of the page does not allow it, a SIGSEGV is delivered by the operating system to the application. This signal is captured by NanosDSM and is handled properly. This usually requires changing the state of the page causing the signal to allow the memory access to be completed correctly. Changes to the state of a page are always made being consistent with the rules described above. For example, if a page needs to be promoted to a RW state from a R state, all pages in a R state in all other nodes must be downgraded to a I state. This mechanism will provide the sequential consistency for all memory accesses of the application. Page information is stored in one node, the *master* node of the page. This node has the responsibility for handling the state transitions of a page. This can become a performance problem if only one node is the responsible for this task. To improve the performance, NanosDSM implements a *migration* system where the *master* node of a page can change during the execution, allowing for a better load balance of requests across the nodes.

To handle all the system features described, an extra thread is needed in every node. This thread is called *infoserver* and it is the responsible for receiving page fault requests, managing the state transitions of pages and offering all the extra features offered by the NanosDSM.

The communication system between nodes is implemented using the *Myrinet Express* (MX) API. It is the most advanced driver available for *Myrinet* NICs, and offers a higher performance than using the standard UNIX sockets interface on top of these NICs (which is also supported by NanosDSM).

4 Running the XL SMP runtime over NanosDSM

Given the features of our SDSM system, the integration with the XL SMP runtime was a straightforward process. NanosDSM can be seen as a regular threads library since its API includes calls to allow thread creation, management and termination and a few synchronization directives. The POSIX Threads library has similar calls so we can port to NanosDSM any application that uses it with minimal changes to the code structure. This has been the main idea that we used when doing the porting of the XL SMP runtime to our SDSM system (figure 1).

Still the XL SMP runtime had a few more requirements to be able to run. Memory allocation using the standard *malloc* call was needed in some parts of the code. NanosDSM provides some calls to dynamically allocate memory, but they resemble the *mmap/munmap* specification. We decided to support *malloc* in order to allow applications that use it to run without having to replace their dynamic memory allocations. To implement this feature, we took advantage of the *malloc hooks* that the GNU libc supports. This allows NanosDSM to capture the *malloc* calls from the application and forward them to the master node, where the requests will be processed and the memory allocation will occur. This implementation can cause some significant overhead in applications that make heavy use of dynamic memory allocation with malloc due to the big number of messages that one single node might have to process. However we are not worried about it since we have not worked with such applications and the current implementation can be improved with a small amount of effort.

With this, NanosDSM simply had to take over pthreads functionality to enable the XL SMP runtime to run over the SDSM. Nevertheless, some parts need to be reimplemented to be SDSM-aware and to avoid loss of performance on some critical features such as barriers.

5 Running OpenMP applications on NanosDSM

Given the infrastructure described in section 4, running OpenMP applications in our SDSM system is a simple process, it just requires compiling standard OpenMP code with the XLC or XLF IBM compiler, enabling the support for OpenMP and linking the resulting binary with our runtime libraries. No changes are needed to the original code given the features of our SDSM system.

The ability to be able to execute applications directly on the distributed systems is appealing from the point of view of productivity. However, this process is not always as easy as it looks like since, depending on the application, the performance that can be achieved may be unacceptable. Using a SDSM system adds an extra layer of abstraction to the memory hierarchy, this greatly increases memory latency in some cases, since the data can be physically in a remote machine.

We have seen that the impact of this new layer depends on the application. Some parallel applications are able to perform well on this system while others perform worse even

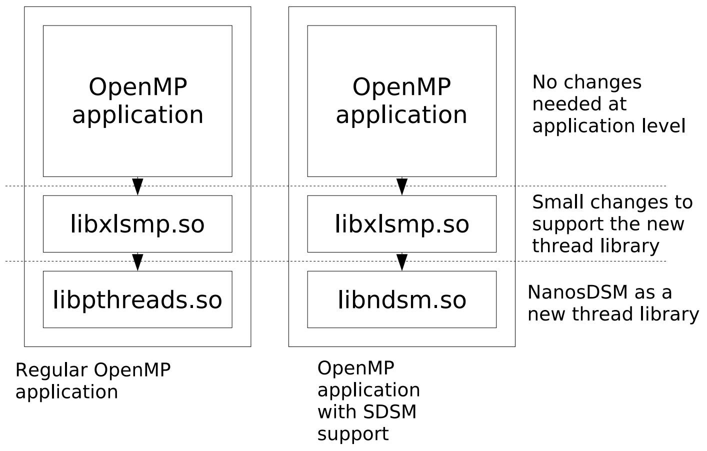

OpenMP application
libxlsmp.so
libpthreads.so
Regular OpenMP application
OpenMP application
libxlsmp.so
libndsm.so
OpenMP application with SDSM support
No changes needed at application level
Small changes to support the new thread library
NanosDSM as a new thread library

Figure 1: XL and NanosDSM integration structure

when running with more threads. Since the key factor is the memory latency, applications that exhibit more data locality and/or less data sharing between threads are the best suited for SDSM systems.

Our application set was formed by a *Matrix-Multiply* implemented in OpenMP, three applications originally implemented in OpenCL but ported to OpenMP (*Perlin*, *MonteCarloAsian* and *FloydWarshall*) and the *MG* benchmark from the NAS parallel benchmarks. As shown in section 7, these applications are good examples of what we stated previously.

OpenCL applications are easily ported to OpenMP since the computation expressed in them is parallel (the OpenCL kernels do not share data). Thus, it can be expressed in OpenMP by a simple combination of *parallel* regions and *work sharing* constructs. This schema of data independent kernels running in parallel fits perfectly with the idea that data sharing among different threads will be low during the execution, which will not impair scalability.

6 Improving the NAS MG benchmark with runtime optimizations

We chose to focus on the NAS MG benchmark (version 2.3, implemented using the C programming language) since, as it is shown in section 7 it was having performance problems when running on our SDSM system, thus we wanted to explore which techniques we could develop using the runtime system in order to improve the MG performance. We had previously measured performance using a slightly different environment (using the NANOS OpenMP runtime instead of the IBM XL SMP runtime) using the EP and CG benchmarks [4]. These results demonstrated that our SDSM system achieves good performance using runtime optimization techniques such as automatically triggered pre-sends and pre-invalidations.

6.1 Data allocation

In order to improve the data locality and to help the implementation of the other techniques that will be discussed later on this section, the benchmark required some changes in the allocation of its data structures. The original MG code allocated a series of three dimensional arrays using a simple loop to call the *malloc* function to obtain dynamic memory. This code successfully generated a contiguous area where the whole data structures were placed, but it mixed data and pointers. This created many areas of read-only data (pointers, since they were not modified during the whole execution of the benchmark) and read-write data (the real contents of the arrays) when running the benchmark with our SDSM environment. It also created some false sharing, since the allocation was not aware of the page granularity, and data and pointers could be placed on the same page. We rewrote the allocation process to first set up the space required to place all pointers, and then compute the size for the rest of the data and allocate it with a single call to *malloc*. We also used this as an opportunity to pad the outer dimension of each array so different elements of the outer dimension would never be placed on the same physical page. This avoids some false sharing in some loops and also eases the design decisions while implementing and adding the techniques previously mentioned.

6.2 Scheduling reuse

The main goal when designing this new scheduling policy was to minimize the data transfers between threads. The NAS MG benchmark has fairly similar loops. By similar we mean that some loops have almost the same number of iterations (some go from 0 to $N-1$, others go from 1 to $N-1$, etc), access the same data structures, and are parallelized across the same dimension. Figure 2 shows an example of this, the *subA* function iterates a given array from 0 to $N-1$, while *subB* performs a similar loop that goes from 1 to $N-2$. Eventually *subA* and *subB* access the same data structure. With this scenario, static scheduling may offer a reasonable good scheduling on an SMP environment. However, on our SDSM environment, it has some side effects. We can see the iteration distribution as a partition of the data across the different threads (nodes) of the application and each thread owns a partition. Accesses to partitions of different threads are costly but accesses owned by the thread are fast. With this, we want to achieve a schedule that keeps the same partitions during the execution of the entire program. This is achieved by registering the distribution used in the different loops of the execution and when a new loop is reached the runtime looks for a similar loop and adjusts the new distribution according to the previously scheduled loop. Since we assume that loops are accessing the same data structures, changes in the number of iterations must come from changes to the lower bound or the upper bound of the loop. Thus, adjustments to match

```
void subA(double *array)
{
    /* ... */
#pragma omp parallel for
    for (i = 0; i < N; i++)
        array[i] = 0.0;
}

void subB(double *array)
{
    double tmp[M];
    /* ... */
#pragma omp parallel for
    for (i = 1; i < N-1; i++)
        tmp[i] = array[i];
}
```

Figure 2: Two sample loops.

the correct number of iterations are made in the first and in the last chunk of iterations. This ensures that the partitions will remain the same for every thread executing the benchmark.

6.3 Pre-sends and pre-invalidations

Pre-sends and pre-invalidations were added to the benchmark code in order to speed up the data transfers between threads. The code needed to perform these operations is simple and can be easily added using a directive, however we aim to have them automatically added by the compiler. Adding the pre-sends and pre-invalidations was a straightforward process given the new scheduling implemented that provides an “owner computes” model to the benchmark. The pre-sends and pre-invalidations are required on the loops where threads accessed data owned by other threads. These loops were easily identified by analyzing how the induction variables were used. If a induction variable was used in a expression of the form $i+C$ (where i is the induction variable and C a constant value) that was indexing an access to an array. In that case, that was present in the majority of the loops of the MG benchmark, the thread would access data owned by the next thread (or the previous, if the constant value was negative) If the data is written, then a pre-send is needed to send the data back to its owner thread. Otherwise if the data is read, then a pre-invalidation is needed to invalidate the local copy of the data to have it upgraded to read-write permissions in its owner thread. With this we avoid most of the true sharing problems of the MG benchmark using a criteria that, we believe, can be easily adapted to an automated process or offered to the programmer in the form of compiler directives.

7 Experiments

In this section we will show the experiments done with a selected set of applications and the results that we have obtained. A brief description of each application follows:

MatrixMultiply It performs a dense matrix multiplication of two square matrices of dimension 3000x3000.

Perlin A perlin noise generator, it generates perlin noise on an image of 4096x1024 pixels. It was originally coded for OpenCL, we ported it to OpenMP.

MonteCarloAsian Monte Carlo analysis are a class of computational algorithms that rely on repeated random sampling to compute their results, in this case it it computes the asset price of Asian Option Calls, which follows the classic Black-Scholes model. Also it was ported from an OpenCL code to OpenMP.

FloydWarshall It computes the shortest path between each pair of nodes in a graph defined by its adjacency matrix. It is a dynamic programming approach that iteratively refines the adjacency matrix of the graph in question until each entry in the matrix reflects the shortest path between the corresponding nodes.

NAS MG From the NAS Parallel Benchmarks, the MG performs a *multigrid method* which is an example of a hierarchical algorithm. These algorithms can be found at the core of large scale scientific applications. We have used the C implementation of the benchmark selecting the class B data set.

The platform we used to evaluate our system is a distributed machine with each node running Linux 2.6, containing 2 PowerPC 970MP @ 2.3GHz processors, 4GB of physical memory, and connected by a Myrinet network. Up to 16 nodes were used on our experiments. All applications were compiled using the IBM XL C/C++ V8.0 compiler, using -O3 optimization level. We allocated one application thread on each node when running the benchmarks so each thread was physically a different machine.

We also measured the performance of the applications in a shared memory environment, to compare the performance loss in terms of speed-up between our SDSM system and a real hardware shared memory one. The shared memory platform used was a SGI Altix 4700 with 256 processors, and 2.4 Terabytes of memory. The benchmarks were compiled with the Intel C Compiler (O2 optimization level).

Our tests compared the performance when running our SDSM system with 2, 4, 8 and 16 OpenMP threads (so 2, 4, 8 and 16 nodes). The main metric measured is execution time of the benchmark but we also measured the number of page faults produced by the benchmarks. Speed-ups are measured against the sequential execution of the benchmark. The sequential time measured for every benchmark on each architecture is shown in table 1.

We achieved good performance with *MatrixMultiply*, *Perlin* and *MontecarloAsian*. In the next figures, we compare the performance obtained in the NanosDSM environment (label *NDSM*), with the performance obtained in the Altix environment (label *SMP*). *MatrixMultiply* achieved a 6.4 speed-up running on 8 nodes and 11.7 speed-up running on 16 nodes (figure 3(a)). *Perlin* showed a perfectly linear speed-up (figure 3(b)), also achieved in the Altix environment. These are examples of applications that can perform well on a SDSM environment, they have little communication among threads and do high computational work keeping the job of hiding the communication simple and not too costly to the DSM layer.

MontecarloAsian, while not showing as good performance as the previous applications, also achieved a decent performance when running on 4 and 8 nodes, measuring 3.4 and 5.8 speed-ups respectively (figure 3(c). With 16 nodes it only achieved a modest 8.4 speed-up. The figure also shows that this benchmark suffers the same amount of performance degradation in the Altix environment, compared to the *MatrixMultiply* benchmark.

The performance loss observed is caused by the same problem that also affects *FloydWarshall*. The problem with *FloydWarshall* was caused by the fact that it spawns and joins a parallel region for each node of the graph (4096 nodes in the executions tested). The high number of spawn/join messages between nodes, which in our DSM implementation has a cost dependent on the number of nodes, causes the performance degradation. In the case of *MontecarloAsian* the number of spawn/join requests was not as high as in *FloydWarshall*, thus allowing the application to perform better but not as good as we expected.

Figure 3(e) shows the performance obtained when running the MG application without applying any optimizations (label *NDSM*), except for the new data layout described in section 6.1, since otherwise the address space became too fragmented, and reached the limit of mappable areas of the Linux OS. As expected, the execution time increases when we increase the number of threads. The true data sharing of the application causes too much overhead when accessing to memory because the network is not fast enough to make the page fault system mechanism perform at an acceptable level.

Figure 3(e) also shows the impact of the optimizations presented in this paper (label *NDSM opt*). With the addition of the pre-sends and pre-invalidations, the benchmark achieves some speed-up, but it stops improving when using more than four threads. The figure also shows the scalability of MG in the Altix environment (label *SMP*), as a baseline reference. Note that this version does not achieve much more overall speedup.

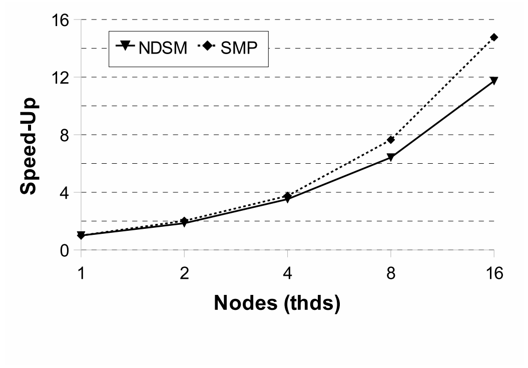
In figure 4 we can see the effects of our optimizations. The chart shows, on a per-loop basis, the percentage of page faults when running the MG benchmark with different threads. The values are normalized to the non optimized versions. The total number of page faults is greatly reduced in all cases, achieving the best reduction whith 8 threads (32% of page faults compared to the non optimized version). Some parallel loops benefit more from presends than others due to having more data locality, which means that decreasing the page faults in a given loop does not necessarily improve the execution time and scalability of it.

As a consequence of the reduction in the number of page faults, the speedup obtained in a per-loop basis depends on the specific loop conditions regarding data exchange among nodes. Figure 5 shows the speedup obtained in each of the 6 parallel loops of MG. We found out that several of the loops (*comm3*, *interp*, *zero3*) do not provide enough computational work to compensate for the cost of executing them on a distributed environment, since they send and receive several messages across the nodes. The result is that these loops do not scale at all. In fact *comm3*, and *zero3* degrade their performance so much that they spend more execution time than their serial versions. Reducing the page faults in these loops does not help because communication is not hidden by computation, this exposes the cost of the presends which is nearly the same as the page faults themselves. We also tried to run these small loops on a single node, but the performance is not better, as then they find the data they need already distributed, causing a large memory movement for them to compute, and an extra movement to later redistribute data for the subsequent parallel loops.

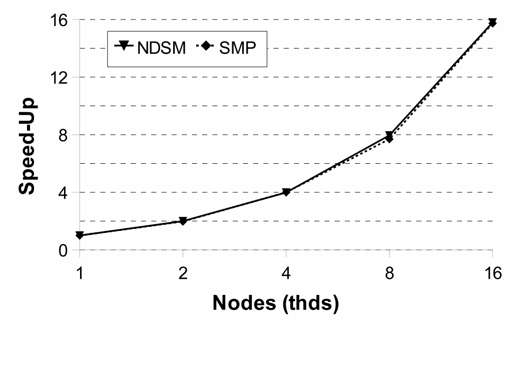
The other three loops in the application, *psinv*, *resid*, and *rprj3* are large enough to compensate the spawn/join operation and to benefit from the effects of the presends which greatly reduce the number of page faults on all of them. This allows them to achieve a speedup of about 3 on 8 nodes.

8 Conclusions

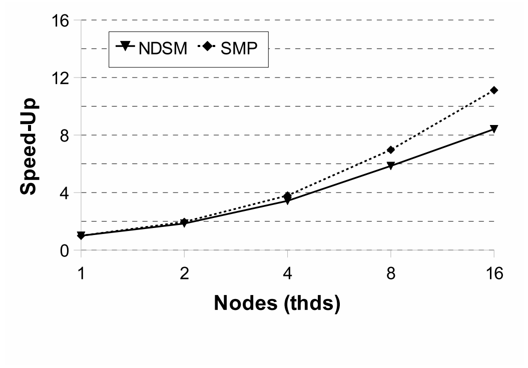
We have presented our SDSM environment using the XL SMP runtime to enable standard OpenMP applications to run on the top of our NanosDSM SDSM library. Standard OpenMP applications can run correctly on this infrastructure without any changes to the original



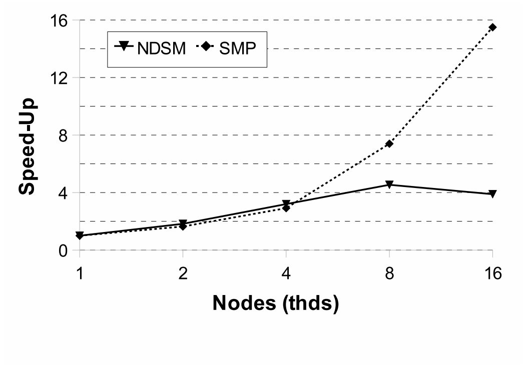
(a) MatrixMultiply



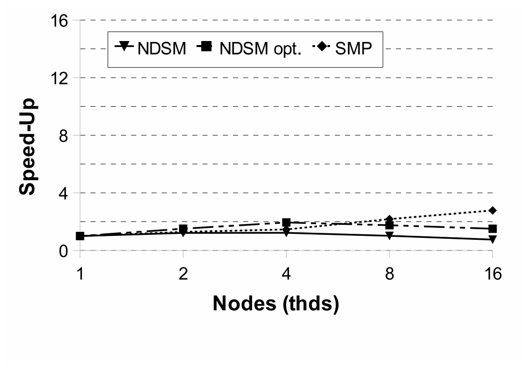
(b) Perlin



(c) MonteCarloAsian



(d) FloydWarshall



(e) NAS MG class B

Figure 3: Evaluation results for all the applications.

Application	PowerPC 970MP	SGI Altix 4700
MatrixMultiply	1644.50 secs.	869.81 secs.
Perlin	1294.78 secs.	700.06 secs.
MonteCarloAsian	56.38 secs.	62.37 secs.
FloydWarshall	338.52 secs.	1497.70 secs.
NAS MG	23.82 secs.	30.90 secs.

Table 1: Sequential execution time of the different benchmarks on our two architectures used.

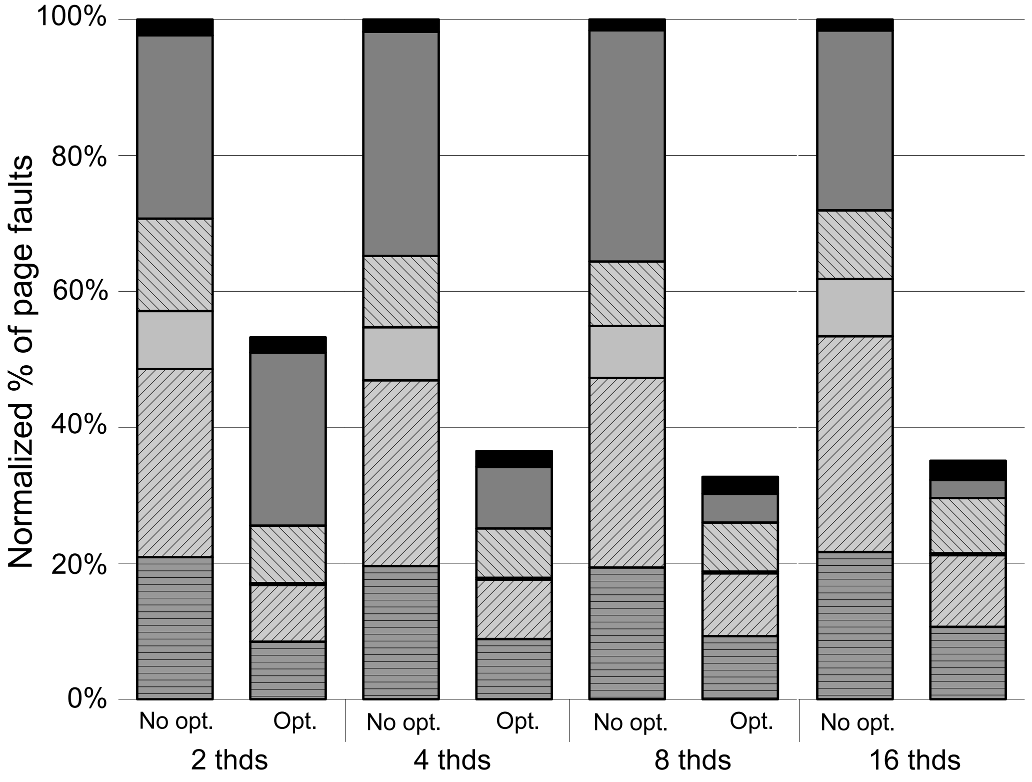


Figure 4: MG class B, percentage of page faults of the optimized version normalized to the non optimized version.

source code, but the performance obtained depends on how sensitive the application is to memory latency.

Applications that require a ratio of computational power vs memory latency biased towards the first factor are the ones that are more suitable for SDSM systems. OpenCL applications are a good example of this, since they can be parallelized in a way that the computation does not share many data between threads, thus making the accesses fast on a SDSM, since data ends up being physically on the node where the thread is running.

On the other hand, applications which require this memory performance to be able to scale properly need from optimizations in order to improve the performance.

We have developed a couple of such techniques, taking the NAS MG benchmark as a representative application. A new scheduling policy and the addition of pre-send and pre-invalidations have helped to improve the performance of the benchmark. However, the experiments done show that there are some parts of the application that can not benefit from these optimizations, since the loops are computationally too small, limiting the amount of overlapping of computation and communication that can be achieved. This sets a limit on these loops scalability, and also on the application scalability.

9 Future work

As we have seen in Section 7, applications generally need help in the form of optimizations in order to run efficiently on SDSM environments. Pre-sends and pre-invalidations are examples of techniques that can have a positive impact on performance but currently they have to be

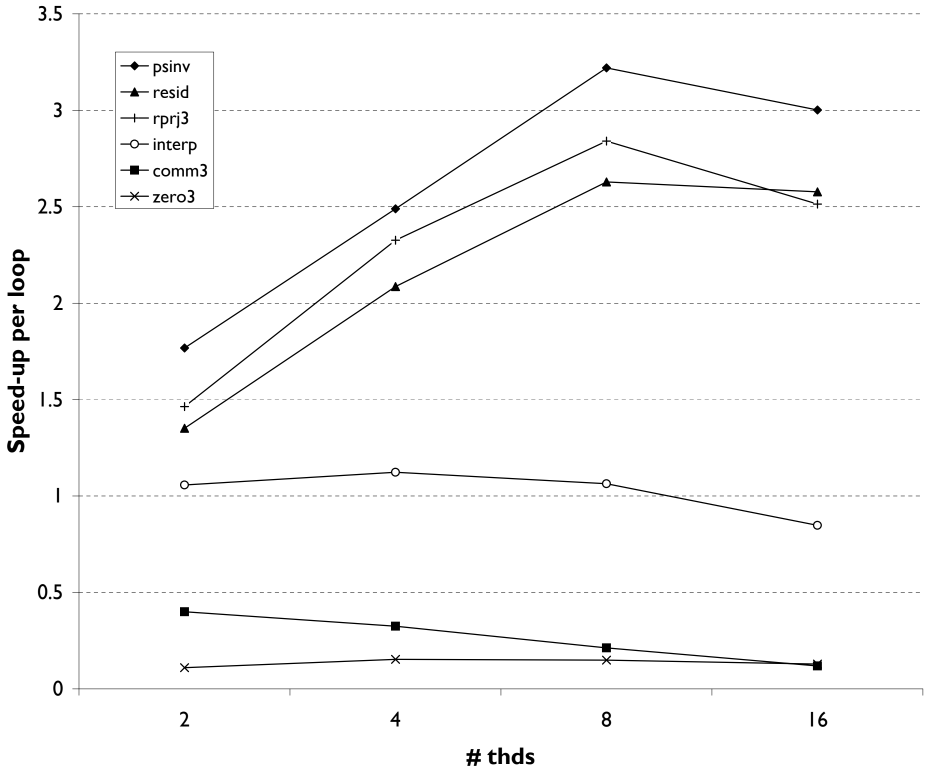


Figure 5: MG class B, per loop speed-up.

manually placed by the programer. This goes against one of the most important design principles of NanosDSM, a SDSM that was conceived to run OpenMP applications without having to modify the source code.

We believe that these pre-sends and pre-invalidations can be placed automatically by the compiler using data flow analysis. However this can be challenging and may not be possible in some cases. For now a more realistic goal may be adding directives to aid the placement of these pre-sends and pre-invalidations without having to make significant changes in the original source code.

Acknowledgements

This research has been supported by the IBM Center for Advanced Studies (CAS), Toronto, the Spanish Ministry of Education (contract no. TIN2007-60625), and the European Union (under the HiPEAC-2 Network of Excelence, FP7/ICT 217068). We also would like to thank the Barcelona Supercomputing Center for the use of their machines.

About the Authors

Javier Bueno is a Ph.D. student from the Department of Computer Architecture at the Technical University of Catalunya (UPC), Spain. He received his degree in Computer Engineering at the same university. His research is focused on Distributed Shared Memory systems but other interests include parallel computing and high performance architectures.

Xavier Martorell received the M.S. and Ph.D. degrees in computer science from the Technical University of Catalunya (UPC) in 1991 and 1999, respectively.

Since 1992 he has lectured on operating systems, parallel runtime systems and OS administration. He has been an associate professor in the Computer Architecture Department at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, compilers and applications for high-performance multiprocessor systems. He spent one year working with the BG/L team in the IBM Watson Research Center. He is currently the Manager of the Parallel Programming Models team at the Barcelona Supercomputing center.

Juan José Costa is a Ph.D. student at the Technical University of Catalunya (UPC), 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 Barcelona Supercomputing Center.

Toni Cortés is the manager of the storage-system group at the BSC and is also an associate professor at the Technical University of Catalunya (UPC). He received his M.S. in computer science in 1992 and and his Ph.D. also in computer science in 1997 (both at the UPC). His research concentrates in storage systems, programming models for distributed systems and operating systems. He has published more than 40 papers in international conferences and journals. Toni has been general chair for the Cluster 2006 conference and belongs to the editorial board of the Cluster Computing Journal. Finally, he has also participated in EU funded projects such as Paros, Nanos, POP, and XtremeOS.

Eduard Ayguadé 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 parallel programming models. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. He is currently associate director for research on Computer Sciences at the Barcelona Supercomputing Center (BSC). His research interests cover the areas of multicore architectures, and programming models and compilers for high-performance architectures.

Guansong Zhang received a PhD. degree from HIT, China in 1995. Worked on HPF and Java compilers as a Research Scientist in NPAC at Syracuse University from 1995. Joined IBM Toronto Lab in 1999 Implemented the major OpenMP 2.0 features in the IBM XL C/C++ and XL Fortran compilers' backend and runtime. Participates language committee discussions for the OpenMP 2.5 and 3.0 standards as one of the IBM representatives.

Dr. Christopher Barton works with the TPO Development group at the IBM Toronto Laboratory. He leads the SMP group, which focuses on compiler optimizations for automatic parallelization as well as explicitly parallel programs. He is also interested in high performance computing with a focus on languages and compilers for large scale machines. He received his Ph.D. in Computing Science from the University of Alberta (2009) and his M.Sc. also from the University of Alberta (2003).

Raul Silvera is a senior developer at the IBM Toronto Lab. After joining IBM in 1997, he has focused on compilation technology, including optimization and parallelization. He has made contributions on the areas of loop transformation, locality analysis, profile-directed optimization, automatic and user-directed parallelization, interprocedural optimizations and others. He is currently the technical lead for TPO, the mid-level optimizer used in the IBM XL Family of compilers.

References

[1] NANOS project overview. http://nanos.ac.upc.edu.

[2] J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Sixth European Workshop on OpenMP*, Stockholm, Sweden, 2004.

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

[4] JJ Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. *Journal of Parallel and Distributed Computing*, 66(5):647–658, 2006.

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

[6] OpenMP Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. 1997.

[7] J. P. Hoelflinger. Extending OpenMP to Clusters. 2006.

[8] L. Whately, R. Pinto, R. Bianchini, C. L. Amorim. Adaptative Techniques for Home-Based Software DSMs. *13th Symposium on Computer Architecture and High Performance Computing*, 2001.

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

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

[11] X. Martorell, J. Labarta, J.I. Navarro, and E.Ayguadé. A library implementation of the nano-threads programming model. In *Lecture Notes in Computer Science, Euro-Par'96*, pages 644–649, Springer-Verlag, August 1996.

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

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

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