Efficient OpenMP over sequentially consistent distributed shared memory systems

Juan José Costa Prats

Advisors: Toni Cortés Rosselló

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Thesis submitted in fulfillment of the requirements of the degree:

Doctor per la Universitat Politècnica de Catalunya

June 2011

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: .....

Autor de la tesi: .....

Acorda atorgar la qualificació de:

No apte
Aprovat
Notable
Excel·lent
Excel·lent Cum Laude

Barcelona, …..…… de/d'…..……..… de .....….

El President

.....
(nom i cognoms)

El Secretari

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

Abstract

Nowadays clusters are one of the most used platforms in High Performance Computing and most programmers use the Message Passing Interface (MPI) library to program their applications in these distributed platforms getting their maximum performance, although it is a complex task. On the other side, OpenMP has been established as the de facto standard to program applications on shared memory platforms because it is easy to use and obtains good performance without too much effort.

So, could it be possible to join both worlds? Could programmers use the easiness of OpenMP in distributed platforms? A lot of researchers think so. And one of the developed ideas is the distributed shared memory (DSM), a software layer on top of a distributed platform giving an abstract shared memory view to the applications. Even though it seems a good solution it also has some inconveniences. The memory coherence between the nodes in the platform is difficult to maintain (complex management, scalability issues, high overhead and others) and the latency of the remote-memory accesses is highly increased due to the interconnection network, which can be orders of magnitude greater than on a shared bus.

Therefore this research improves the performance of OpenMP applications being executed on distributed memory platforms using a DSM with sequential

[1]This work has been supported by the Spanish Ministry of Education (TIN 2004-07739-C02-01, TIN2007-60625), by the Generalitat de Catalunya (2009-SGR-980), by the European Union (under the HiPEAC2 Network of Excelence FP7/ICT 217068 and the POP: Portability of OpenMP Performance project Future and Emerging Technologies IST-2001-33071) and by IBM through the CAS program. We would also like to thank Mathias Muller and the Barcelona Supercomputing Center for the use of their machines.

consistency evaluating thoroughly the results from the NAS parallel benchmarks.

The vast majority of designed DSMs use a relaxed consistency model because it avoids some major problems in the area. In contrast, we use a sequential consistency model because we think that showing these potential problems that otherwise are hidden may allow the finding of some solutions and, therefore, apply them to both models.

The main idea behind this work is that both runtimes, the OpenMP and the DSM layer, should cooperate to achieve good performance, otherwise they interfere one each other trashing the final performance of applications.

We develop three different contributions to improve the performance of these applications: (a) a technique to avoid false sharing at runtime, (b) a technique to mimic the MPI behaviour, where produced data is forwarded to their consumers and, finally, (c) a mechanism to avoid the network congestion due to the DSM coherence messages.

The NAS Parallel Benchmarks are used to test the contributions. One of the results of this work is that the false-sharing problem is a relative problem, and it depends on each application. For example, there are cases where the false sharing does not affect the final application performance.

Results also show that it is really important to move the data flow outside of the critical path and techniques that forwards data as early as possible, similar to MPI, benefits the final application performance.

Another interesting result is that this data movement is usually concentrated at single points and, due to the limited bandwidth of the network, affects the application performance. Therefore it is necessary to provide mechanisms that allows the distribution of this data through the computation time using an otherwise idle network.

Finally, results shows that the proposed contributions improve the performance of OpenMP applications on this kind of environments.

Acknowledgements

Since the beginning of this document, this has been a long journey. A journey where I have met very different people whom I am really grateful and possibly never I have told them. Let me redeem myself.

First of all Xavier Martorell who made the Operating Systems classes a really interesting topic during my undergraduate courses. Xavier also give me an opportunity to enter the now extinct Center of Parallelism in Barcelona (CEPBA) for continuing the research on distributed shared memory systems.

My research needs to acknowledge a lot of people. First of all, Sebastià, who started all this work, and afterwards all the people that help to grow the final result: Pimpam, Ramon, JBueno, JVaquero and Ernest. Thank you, without your contributions and support this thesis could not be possible.

During my research I followed one of the indications from Jesus Labarta, main director of the CEPBA, who think that to improve the performance of OpenMP programs on the distributed shared memory world *you should feel the pain*(meaning that a sequential consistency model should be used). Now that I am finishing this thesis, I can say, that, as usual, he was right... but it really hurts :).

I need to thank Xavier (again) and Eduard Ayguadé for their support in the moments were the work seems stuck at some point and their comments helped to solve or circunvent the problems.

I can not forget about Toni, my thesis advisor, he is without any doubt one of the persons who has dedicated more time to me. He has been there during the best moments, and also during the worst, giving hope and always showing me the good side of the situations. Thank you (no, really, Thank You).

Aside from people related to my research, I need to thank also all those who have had the time to share a good coffee in the campus... well, maybe not so good, but you made it better. I would like to thank Alex specially for finding holes in his incredibly busy agenda ;).

Finally, I need to give a special thank you to my partner, because she has been always there, giving support and helping me to believe that this day would arrive, taking care of the good and bad moments (specially the bad ones). She is the best thing I could have ever wished and together we have made our best achievement. Thank you.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

DON'T PANIC
The Hitchhiker's Guide to the Galaxy
Douglas Adams

Abstract

This chapter introduces the research topic, the motivation and the contributions of this thesis. Finally, the structure of the remaining document is outlined.

1.1 A distributed world

Distributed memory platforms, like clusters, are one of the most used architectures for high performance computing nowadays. According to data extracted from the top500 list [top], clusters represent a 83% of the 500 most powerful machines installed in the world. In fact, their progress have been growing exponentially since the end of the 2000 year. This exponential progress is explained because they are more cost effective than its shared memory counterparts.

On one hand, the cost of these shared memory machines is explained by the extra hardware needed to maintain the global shared address space. The memory should be coherent across all the processors, and they are connected through an interconnection network that should deliver a high bandwidth and a low latency.

On the other hand, clusters have a distributed address space, where each node has its own memory address space, and the different nodes are connected through commodity networks simplifying the design and lowering the costs.

Any programmer using one of these distributed machines normally use a distributed memory programming model, such as MPI message passing [mpi, For94], where each node has a private address space and thus a node must communicate any produced data explicitly to the nodes that will use it. The problem with this programming model is that a programmer needs to create an algorithm taking care of all the burden of the data distribution between the nodes in the machine to achieve a functional parallel program.

Shared memory machines have a globally shared addressable memory, and programming models like OpenMP[1] [Boa04] allow a straightforward parallelization of sequential applications without too much effort.

The ease of programming of shared memory computers when compared with the complex task of programming a distributed memory one, makes us wonder if it could be possible to program a distributed one with the same complexity as the shared one.

[1]OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this product/publication may have been derived from the OpenMP Language Application Program Interface Specification

1.2 Motivation

The use of a shared memory programming model like OpenMP on top of a distributed memory platform, like clusters, has already been researched and some of these works are presented in the next Chapter 2.

We focus our research in the execution of OpenMP applications on top of a distributed system using a distributed shared memory layer. This layer offers a shared addressable memory space across the nodes in the cluster, allowing the execution of OpenMP applications.

But the OpenMP programming model, thought for shared memory machines, has some issues when used on distributed platforms that impact in the final application performance.

Basically, clusters have non-uniform memory accesses. A processor in this architecture has access to: i) local memory, located in the same bus as the processor; and ii) non-local or remote memory, accessible through an interconnection fabric or network. The local memory accesses take just a couple of nanoseconds, but an access to memory on a remote node can take orders of magnitude more.

OpenMP is a model which hides the subjacent hardware from the programmer, so it helps the programmer to focus in parallelizing the algorithm and the access pattern. The problem is that the cost of the different memory references is also hidden, and the programmer does not have any control over it. It is the runtime responsibility to minimize these accesses to remote memory, or at least to hide their communication overhead.

On the othe hand, a programmer using a distributed programming model, like MPI, is conscious of these different costs, and he explicitly avoids (or minimizes) these remote accesses to improve application performance.

In this research we try to address some of the problems that arise when using a DSM to execute OpenMP applications by enabling a tight cooperation between the OpenMP runtime and the DSM software.

1.3 Thesis and Goals

The thesis for this work is that it is neccessary to have a tight cooperation between the different layers to improve the performance of OpenMP applications when they are executed on distributed environments using a distributed shared memory software,

For this work we have used a DSM with sequential consistency, and, to accomplish this thesis we think that the following goals should be solved:

1. **Tolerate false sharing adapting the application at runtime**
2. **Apply lessons learnt from MPI**
3. **Avoid network congestion**
4. **Proposal of OpenMP extensions**

The research presented here tries to avoid the modification of the OpenMP application source code or, at least, modify it without affecting the initial algorithm. For example, with the addition of new directives.

1.4 Organization of this thesis

This work is organized in 7 chapters as follows:

- Chapter 2 introduces some related concepts needed for understanding this work. It also shows the environment used to execute OpenMP applications on top of a distributed platform.
- Chapter 3 shows the first contribution of this work: the boundary alignment. Describes its philosophy, its design and obtained results.
- Chapter 4 presents the second contribution of this work: a mechanism to overlap communication with computation that tries to imitate the MPI behavior. It describes the design of this mechanism and evaluates its performance.

- Chapter 5 explains a mechanism to distribute coherence data during the computation time to avoid network congestion when this coherence data is send at once after the computation. This chapter discusses its design issues, and it evaluates its performance.
- Chapter 6 presents an extended evaluation of the three contributions applied on the NAS Benchmark.
- Finally, Chapter 7 concludes this work with a summary of all contributions and the future work.

Chapter 2

Background and related concepts

Abstract

This chapter presents some concepts to aid in the understanding of this thesis work, the research being done in areas similar to our, the benchmarks used in the performance evaluation and our environment.

2.1 Introduction

The research presented here is focused on distributed-memory multiprocessors, where individual nodes containing one or more processors and some memory are connected through some interconnection network as Figure 2.1 shows. These machines are commonly known as *clusters*.

The nodes inside these clusters are usually *symmetric multiprocessors* (SMPs) where the processors connects to a single shared main memory. These nodes are also known as *uniform memory access* (UMA) architectures because they have a uniform access time from any processor to the memory.

2.1.1 Memory address spaces

Different architectures have different memory address spaces and Figure 2.2 shows some of them.

In first place it shows the hardware shared address space, present in SMPs, where three threads share the same code, data and heap, but different locations in the shared address space for the different stacks.

In second place, it shows the memory address space available in a distribute platform, present in clusters. There are three different nodes, each one with its own private address space, and so the application is replicated between the nodes. Any communication should use some mechanism of message-passing. Due to the distributed address space, the three threads

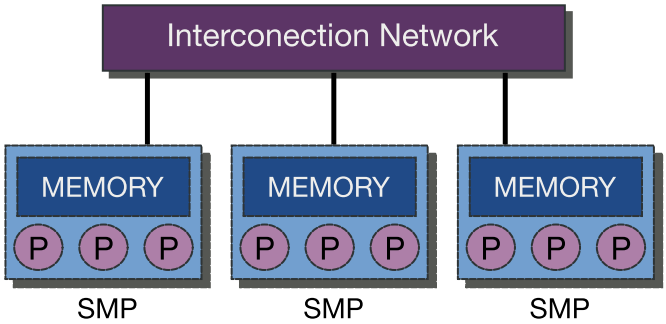

Figure 2.1: Basic architecture for a distributed-memory multiprocessor consisting on individual nodes containing one or more processors, some memory and an interface to an interconnection network that connects all the nodes.

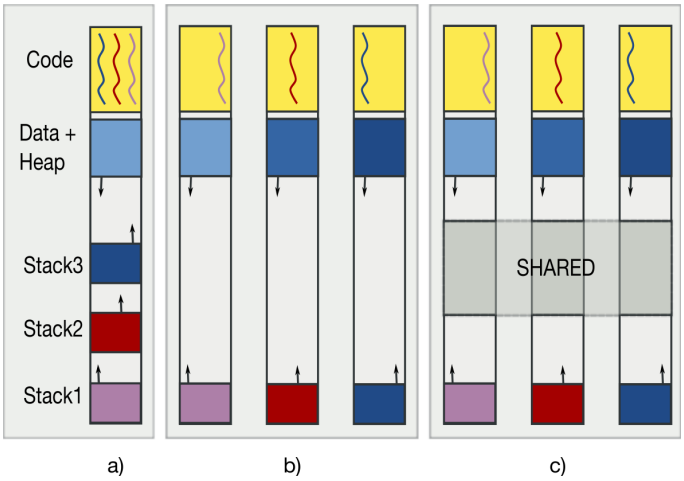

Figure 2.2: Different memory address spaces: (a) shared memory, (b) distributed memory and (c) distributed shared memory.

can use the same addresses for their stacks.

Finally, it shows the address space of a distributed platform that uses a DSM system. As in the distributed case the code is replicated and the data and heap are completely private but now it allows the explicit sharing of a specific memory region.

In contrast to this distributed view, NanosDSM, the DSM used in our environment, offers an everything-shared address space similar to the hardware shared address space offered by the SMPs. This DSM is explained later in section 2.7.

2.1.2 Coherence and Consistency

Whenever two or more processors share a common area of memory and they have copies of the same memory value in their local caches, the cache coherence problem appears. How to maintain both memory values coherent?

Adve et al. [AG96] explains that, on one hand, the usually referred as *cache coherence protocol* is the mechanism to propagate a newly written value to all the cached copies of the modified location. This propagation is usually done by *invalidating* or *updating* all cached values.

On the other hand, the *consistency model* defines a policy for when this

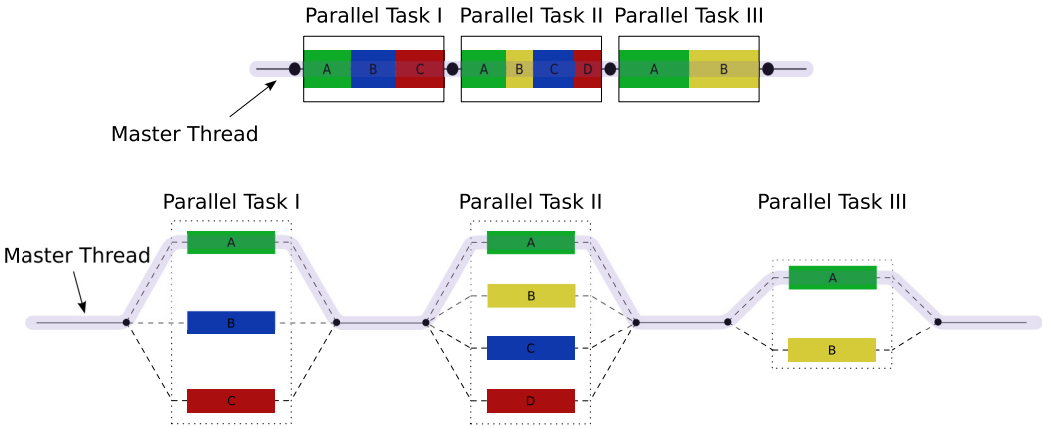

Figure 2.3: Illustration showing the OpenMP fork-join parallelism.

written value should be notified to all caches, giving an upper and lower bound.

2.2 OpenMP

OpenMP [Boa04] is a paradigm designed to parallelize C/C++ or Fortran programs in a shared memory environment. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The programmer annotates the sequential code with these directives or pragmas, giving hints to the OpenMP runtime about, for example, parallel regions or synchronization events.

OpenMP follows a fork-join parallelism (see Figure 2.3[1]), there is a master thread executing an application and when it enters a parallel task it creates a fixed number of threads, *forks*, and divides the task among them. When the task finishes, the threads *join* back to the master thread continuing the execution.

2.2.1 Directives

In this section we present some of the OpenMP directives encountered in the benchmarks used in the performance evaluation. The information presented

[1]Extracted from http://en.wikipedia.org/wiki/File:Fork_join.svg (accessed April 2011)

here is a summary extracted from the OpenMP specification to ease the reading of next chapters, but the source should be consulted to get more details. Only the Fortran API is presented.

parallel Construct

The *parallel* directive is the mechanism to spawn parallelism in a program. When a thread encounters this directive creates a team of threads to execute the parallel region.

```
1 !$omp parallel [clause[[,] clause]...] ]
2   structured_block
3 !$omp end parallel
```

Loop Construct

The *loop* construct distributes the iterations of the loop across the team of threads inside a parallel region, which will be executed in parallel.

```
1 !$omp do [clause[[,] clause]...] ]
2   do_loops
3 [!$omp end do [nowait] ]
```

We use the term *orphaned loop* when the *loop* construct is not nested within another construct than can determine the execution context. For example, having a *loop* construct inside a subroutine without a parallel construct.

master Construct

The structured block after the *master* directive is executed by the master thread of the team only.

```
1 !$omp master [clause[[,] clause]...] ]
2   structured_block
3 !$omp end master [nowait]
```

single

The *single* directive executes the structured block inside the single region by only one thread in the team.

```
1 !$omp single [clause[[,] clause]...] ]
2   structured_block
3 !$omp end single [end_clause[[,] end_clause]...] ]
```

barrier

All threads in a team that finds this construct will execute a barrier, and no thread is allowed to continue the execution before the others have finished all their tasks.

```
1 !$omp barrier
```

2.3 Message Passing Interface (MPI)

Message Passing Interface (MPI) [mpi] library is the most used method to parallelize applications in distributed platforms. It offers a collection of communication functions to send and receive data between two or more computers.

It follows a SPMD (Single Program Multiple Data) model, meaning that MPI allocates all the processes that the user wants and all of them execute the same program but on different data.

2.4 Distributed Shared Memory (DSM)

Distributed Shared Memory (DSM) is a layer on top of a cluster offering an abstract view of a globally shared memory. All locations in the global memory are accessible by all nodes in the cluster but the content of the memory is distributed across the nodes. Due to that distribution some memory is local to the node and the rest is remote, meaning that there are different access times to memory depending on the location. This is known as a *Non Uniform Memory Access*(NUMA) architecture.

Hardware implementations of DSM exist like the Stanford DASH [LLG+92] or Star-T Voyager [ACRA98] but the software implementations are more

usual. The main differences between them being the consistency model [AG96] implemented, the memory access granularity or the platforms supported.

2.4.1 Consistency models

This section describes the main consistency models used to maintain the memory consistent between different nodes in a DSM: Sequential and relaxed consistencies.

Sequential Consistency

Lamport [Lam79] defines sequential consistency as *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* This means, that a system will be sequentially consistent if the result of any execution is the same as obtaining a code mixing all instructions executed by all processors and executing it in an uniprocessor system.

Since the first reference to a DSM [LCBZ97] several other works implementing sequential consistency have appeared. For example, Murks [PR01] which implements a multithreaded page-based DSM, Jackal [VBB01] that uses a Java implementation and so an object-based DSM or Millipede [ISW96, NS01] which adapts the data granularity by creating different views of the same memory region.

NanosDSM, the DSM used in this work, is another example that uses this consistency offering an everything-shared DSM.

Relaxed Consistency

Relaxed consistency, in contrast to sequential, relax the restriction imposed by Lamport which forces that a change in a memory value should be immediately seen by all other processors. This relaxation is usually implemented by making a twin of the modified memory-page and sending the differences between this twin and the modified page at some point. In any case, relaxing the consistency has been extensively studied, because lowers the data

granularity and reduces the quantity of data transfer, at the cost of higher computation to detect exactly what has been modified and giving the responsibility of controlling the coherence to the programmer.

Different types of consistency models have been studied which basically modifies the exact instant where the calculated differences should be sent: release consistency [ACD+96b], lazy release consistency [KCZ92], scope consistency [ISL98], entry consistency [ACD+96a] or even transactional consistency models [SSS+99, STS98].

These consistency models are implemented by, for example: Brazos [SB97] which implements scope consistency on top of Windows NT, Munin [BCZ89, CBZ91] which uses release consistency, Quarks [SSC98] release consistency, Treadmarks [KCDZ94] using lazy release consistency, CVM [Kel96a], Jiajia [HST99] who uses an scope consistency and a bigger shared address space containing the sum of all memory nodes or MOME [Jég00, Jég03] which also uses both consistency models (strong and weak) but it sends whole pages instead of calculating the differences.

2.5 OpenMP on distributed memory

OpenMP has been designed to use a shared memory environment, but there are different approaches to execute an OpenMP application on a distributed memory environment.

Transform the source code. At compile time, OpenMP source code can be translated to other languages suited for distributed platforms, MPI or global arrays for example.

Use a distributed shared memory. Source code could be unmodified and use a DSM at runtime offering the abstraction of a shared memory between the different nodes.

A combination of both. Where smaller granularity is handled at compile time by transforming the code to MPI for example, and coarser granularity is handled at runtime with a DSM.

2.5.1 Transform the OpenMP code

One of the techniques used to execute OpenMP applications in a distributed environment is to analyze the accesses to the shared data and transform them to other languages. Basumallik et al. [BE05, BE06] transform the OpenMP code to use the MPI message-passing library. This kind of transformations are feasible when the application makes regular accesses, but when the accesses are irregulars then the transformation becomes trickier.

Huang et al [HCL05] make the same transformation, but instead of using MPI, they used global arrays (GA).

2.5.2 Execute OpenMP on top of DSM

Another technique is to use a DSM that offers a shared memory layer between the nodes executing the OpenMP application. This is the approach followed in this thesis, the OpenMP source code is compiled with our Nanos Compiler which generates a binary linked with the OpenMP runtime and the NanosDSM library.

Usually, available software DSMs needs to modify the application source code to mark the variables that should be shared, TreadMarks [KCDZ94] for example.

To avoid this annoyance some compilers transforms an annotated OpenMP source code to its equivalent code to be used in an specific DSM automatically. For example, Sato et al. [SSKT99, SHI00, SHH01, OSHI03] presented the Omni compiler which detects the variables to share and inserts code to be executed on top of the SCASH [HIH+00] DSM. Their results are comparable to the results obtained with direct MPI [HJMR02].

Similarly, Intel has integrated TreadMarks [KCDZ94] inside its compiler [Hoe06], with promising results for small applications but for larger applications too much tweaking was necessary [TMSW08].

In our case it is not necessary to mark the shared variables, because the NanosDSM is an everything-shared DSM, meaning that all application memory is shared by default.

2.5.3 Hybrid programming

Finally the last option is to use both components at compile time and at runtime. For example, Parade [KKH03] uses this kind of hybrid programming, it uses MPI for synchronizations and small data-structures and a DSM for the rest of the code (HLRC). Min et al [ME08] presents a similar approach focused on improving applications with irregular accesses.

2.6 Benchmarks

2.6.1 NAS Benchmarks

The NAS Parallel Benchmarks (NPB) [BBB+94, JFY99] are a set of benchmarks developed at the NASA Advanced Supercomputing (NAS) division to measure and evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three simulated CFD applications. We have not used the integer sort benchmark (IS).

Embarrassingly parallel (EP)

This kernel generates pairs of Gaussian random deviates according to a specific scheme. This is a really parallel benchmark, where all the data in the loop is private and it finally does a reduction. This benchmark is useful to establish the reference point for peak performance of a given platform.

Conjugate gradient (CG)

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.

Fourier Transformation (FT)

FT computes a 3D fast Fourier Transformation (FFT), performing three consecutive 1-D FFTs in each of the three dimensions.

```
1  call setup
2  call fft(1)
3  do step=1, niter
4    call evolve
5    call fft(-1)
6    call checksum
7  enddo
```

Multigrid (MG)

This is a kernel that uses a V-cycle MultiGrid method to compute the solution of the 3D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement.

```
1  do step=1, niter
2    call rprj3
3    call psinv
4    call interp
5    call resid
6    call psinv
7  enddo
```

Block tridiagonal solver (BT)

The BT benchmark is an application for a typical problem on computational fluid dynamics codes (CFD) that solves 3-dimensional compressible Navier-Stokes equations. It updates a 3-dimensional array of points successively in the x-, y-, and z-direction solving a system of equations per planar grid point.

The algorithm iterates through five basic functions: i) compute the right hand side matrix (rhs), solve the equations in the ii) x-, iii) y-, iv) and z-direction, and finally v) accumulate the results. These 5 functions contains 15 parallel loops in the version used (NPB 3.3). The rhs function has 11 parallel loops, and the remaining functions have one parallel loop each. All loops are

parallelized using the outermost dimension (z) except five of them, where the second outermost dimension (y) is used due to their data dependencies.

```
1  do step=1, niter
2    call compute_rhs
3    call x_solve
4    call y_solve
5    call z_solve
6    call add
7  enddo
```

The loops basically modify eight large shared structures: us, vs, ws, qs, square, rho i, u and rhs. Six 3-dimensional and two 4-dimensional matrices. The interesting part of this benchmark is how these structures are read or written at each loop. Most of these variables are written once, at the rhs1 and add loops, and read at the remaining loops. The unique exception is the structure rhs, which is written in almost all of them.

Pentadiagonal solver (SP)

This simulated CFD application has a similar structure to BT. The solution is calculated using the the Beam-Warning approximate factorization instead of the Alternating Direction Implicit.

```
1  do step=1, niter
2    call compute_rhs
3    call txinvr
4    call x_solve
5    call ninvr
6    call y_solve
7    call pinvr
8    call z_solve
9    call tzetar
10   call add
11 enddo
```

LU Solver (LU)

LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting

from finite-difference discretization of the Navier-Stokes equations in 3D by splitting it into block Lower and Upper triangular systems.

```
1  do step=1, niter
2     call compute_rhs
3     call jacld
4     call blts
5     call jacu
6     call buts
7     call add
8  enddo
```

2.6.2 SPLASH Benchmarks

Ocean

The Ocean application is one of the SPLASH benchmarks [WOT+95], that 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.

```
1 !Weighted average computing
2           diff = 0.0;
3 C$OMP     PARALLEL DO PRIVATE (i,j, tmp)
4 C$OMP&    REDUCTION (+:diff)
5           do j = 2, n+1
6              do i = 2, n+1
7                 tmp = A(i,j)
8                 A(i,j)=0.2*(A(i,j)+A(i,j-1)
9                        +A(i-1,j)+A(i,j+1)+A(i+1,j))
10                diff = diff + abs(A(i,j) - tmp)
11             enddo
12          enddo
```

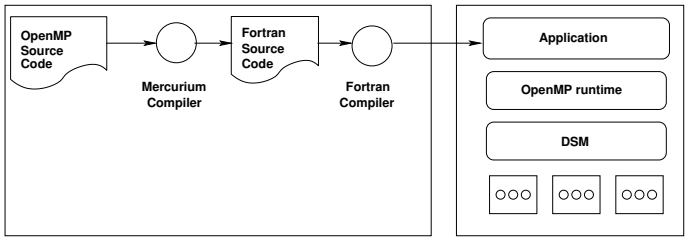

Figure 2.4: Components for running an OpenMP source code application on top of a distributed architecture.

2.7 Our environment

In order to execute an OpenMP application on a distributed environment like a cluster, we will go through different phases. As shown in Figure 2.4, we start from an OpenMP application source code, coded in Fortran for our examples. This application is compiled through a source-to-source compiler, Mercurium, which translates OpenMP directives into plain Fortran code with added calls to our Nanos OpenMP runtime. The resulting Fortran code is compiled through the native Fortran compiler present in the machine to obtain an application object. And, finally, this application object is linked against the OpenMP and DSM libraries to obtain the final application binary.

2.7.1 Mercurium compiler

In our environment, OpenMP applications are parallelized using the Mercurium compiler [BDG+04]. This compiler understands OpenMP directives embedded in traditional Fortran codes, such as the NAS benchmarks 2.3 [JFY99] 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.

2.7.2 OpenMP Runtime

Nthlib [MLNA96, MAN+99] is our runtime library supporting the Fork-Join parallel codes.

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.

2.7.3 NanosDSM

NanosDSM is the software that offers the shared memory abstraction to an application being executed in a distributed environment. It shares the whole memory of a node between all members in the cluster nodes.

It has two components: 1) a user-level library, to be linked with the application, which offers functionalities of a thread library; and 2) a server daemon, that will be responsible to maintain the memory coherence by distributing and gathering memory pages to and from nodes.

The library has different functions to send work to a specified thread in a node, to synchronize threads and to communicate data between threads.

Each node in the cluster sharing the memory will execute a serve daemon: the *infoserver*. This daemon has two features: 1) handle requests from local threads for remote pages, and 2) handle requests from other infoservers.

The node starting the application is known as the *master node*. The master node has all the application memory, and the remaining nodes do not have any access to it. When a thread accesses an address that is not valid, it will generate a trap that will be captured by the operating system (a SIGSEGV). The signal handler will request a copy of the page containing the offending address and the working thread will be blocked until this page

Table 2.1: Platforms used in our tests.

	Kandake	**Crossi**	**Marenostrum**
Nodes	8	24	2560
Available nodes	6	7	8
Processors per node	2	2	2
Processor type	Intel	Intel (Hyper threaded)	PowerPC (Hyper threaded)
Processor speed	266MHz	2.4GHz	2.3GHz
RAM per node	128Mbytes	2Gbytes	4Gbytes
Network	Myrinet	Myrinet	Myrinet Gigabit

arrives.

The request will be directed to the page master node who has all the information about the page, and it will update the coherence graph, solving the request and answering with the requested page.

After receiving the page, the infoserver will put the page in the local memory with the right protections and it will unblock the working thread, continuing the application execution.

Sequential consistency

The implementation of the memory coherence follows a sequential consistency model as explained in Section 2.4.1.

Memory coherence protocol

To maintain the memory consistency, NanosDSM uses a memory coherence protocol similar to the MSI protocol.

A single node is allowed to write a page in the same instant, but multiple readers are allowed.

2.7.4 Testbed

Three different systems have been used to test and develop the present research: (a) Kandake, (b) Crossi and (c) Marenostrum.

Their characteristics are summarized in Table 2.1.

Kandake

Kandake was a Pentium based cluster located at the Center for European Parallelism in Barcelona (CEPBA) facilities. 8 nodes were interconnected through a Myrinet network. Each node has a dual-processor Pentium II at 266 Mhz with a total of 128Mb of RAM.

Crossi

Crossi was an Intel based cluster located at the High Performance Computing Center Stuttgart (HLRS). It had 24 nodes, each one with a dual hyperthreaded processor Xeon at 2.4Ghz and 2 gigabytes of main memory. The nodes were interconnected through a myrinet network.

Marenostrum

MareNostrum is a cluster of PowerPCs at the Barcelona Supercomputing Center (BSC). It is a supercomputer with 31 racks, each one with 6 BladeCenters, and each BladeCenter with 14 JS21 nodes. The JS21 node has two dual hyperthreaded processors PowerPC 970MP at a frequency of 2.3 GHz, and 8Gb of shared memory between both processors. All nodes are connected through a fast Myrinet network and a full-duplex Gigabit.

Due to the global availability of this machine and its shared use, the tests in this research are executed in the same BladeCenter to avoid interferences with other users jobs. We have used a maximum of 8 nodes.

Chapter 3

Boundaries alignment

With great power there must also come
– great responsibility!

Amazing Fantasy #15 (August 1962)
The first Spider-Man story.
Stan Lee

Abstract

False sharing is a known problem plaguing DSMs. It is produced when two nodes access two different offsets in the same cache line, and at least one of them is a write. A typical solution to cope with this problem is to relax the memory consistency. We think that this is unnecessary, and it can be avoided with a tight cooperation between OpenMP and DSM runtimes. This chapter presents the design, implementation details and evaluation for a mechanism that tolerates at runtime the false sharing in an application without relaxing the memory consistency. The idea is to modify the iteration space of parallel loops to avoid the false sharing. Results show that this mechanism is suitable for regular applications working with linear data structures.

```
1 !$omp parallel do
2     do i=1, N
3         A[i] = ...
4     enddo
```

Figure 3.1: Example of a Fortran loop parallelized with OpenMP.

3.1 Motivation

OpenMP is one of the most used programming models for shared memory systems, mainly due to its ease of use and its more than acceptable performance. One of its features is to automatically divide the iteration space of parallel loops among the processors that will execute them in parallel.

Figure 3.1 shows a simple Fortran Loop that writes all positions of an array (`A`) parallelized with an OpenMP directive. This example can be executed on top of any parallel system with shared memory, like an SMP system. Each processor of the SMP has a local cache to improve the accesses to global shared memory. In our example, each cache contains a part of the `A` array, thus when two processors write to the same memory cache line, the sharing problem appears. This sharing increases the memory traffic, because the memory system, who must guarantee the cache coherence, invalidates the cache line and forces a request for the new values.

Two sharing situations are possible: *true sharing* and *false sharing*. *True sharing* happens when two processors access exactly the same offset at the same cache line and at least one of the accesses is a write. *False sharing* happens when two processors access different offsets at the same cache line.

True sharing is produced by the programmer and therefore it can not be avoided at runtime, because the programmer (possibly) knows what he is doing.

In false sharing, data is not really shared but it produces cache invalidations that would be avoided if data was placed in different cache lines. This kind of sharing is produced by accessing different positions in an array (like our example), known also as *self-variable false sharing*; or by different variables that fits in the same cache line, known as *cross-variable false sharing*.

Both false sharings are due to a mismatch between the size of the data and the size of the hardware cache line. With bigger cache lines, more data can fit inside them, and so the probability of issuing false sharing is also bigger. On SDSM systems, this is specially problematic, because the cache line size is usually bigger than processor caches. In page-based DSMs it corresponds to a whole physical page size of 4096Kb.

Our work focuses on removing at runtime the self-variable false sharing on OpenMP loops, because cross-variable false sharing can be easily solved by the compiler.

3.2 Thesis

A popular solution to the false sharing 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.

A typical false sharing problem appears when a loop is parallelized by the OpenMP runtime. The runtime ignores the fact that the application is being executed in a distributed environment and so it just distributes the loop iteration space between the available threads, without any care about the data placement. In the worst case, the last iterations executed by one thread and the first iterations of the following thread will share the same page, and so, there will be false sharing.

Our proposal is to give more information to the OpenMP runtime about the underlying layer to be able to distribute the iteration space in a smarter way without relaxing the consistency. This proposal needs a tight cooperation between the runtime and the DSM layer.

3.3 Related work

The effects of the false sharing, increase of memory traffic due to the "ping-ponging" of cache lines, have been studied largely in the literature [TLH90, GP91, EJ91]. There are two main ways to solve the false sharing problem: at compile time or at runtime. On one hand, the source code of the application can be modified to avoid the sharing between the processors. On the other, the false sharing can be tolerated by allowing the sharing and merging the results at the end.

3.3.1 Compile time

Chow and Sarkar [CS97] categorizes the solutions to reduce or eliminate false sharing at compile time in four groups:

Changing loop structures Changes in the parallel loop structures can avoid the false sharing. For example, transforming the program in such a way that iterations in a parallel loop accesses different cache lines (e.g., by blocking, alignment or peeling) [WL91, GP91, Gra93, KRC99, KCRB03]

Changing data structures The layout of the data structures can also be rearranged. For example, changing the starting address or the size of one of the dimensions of an array (array alignment and padding) [TLH90, BCJ+94, KRC99, HCZ00, JE95]

Copying data Copying the original data to be updated by the loop to a temporary area which does not suffer from false sharing and is well suited to the data access pattern [TGJ93, EJ91, LRW91]. After executing the loop, the data at the temporary area is copied back to its original placement (even though this last step may show false sharing).

Changing schedule parameters Schedule the loop iterations so that concurrently executed iterations access different cache lines [SSMBL94].

They also think that the false sharing can be eliminated changing the schedule and thus they add three parameters (chunksize, chunkstride and peel) to the DOALL fortran library implementation to eliminate it.

Granston et al. [GW93] presented the idea of align the iteration space to page boundaries. Later, Bodin et al. [BGG+95] proved that it is feasible, showing the results of this theory.

Nikolopoulos et al. [NAAL01] presented the idea of reusing a schedule to exploit spatial locality between different parallel loops.

3.3.2 Runtime

The most used technique to tolerate the false sharing problem at runtime is to relax the memory consistency, using a multiple writer protocol (LRC) [KCZ92, Kel96b]. Different DSM implementations with this protocol exists: TreadMarks [ACD+96b], CVM [Kel96a], Quarks [SSC98], ... Amza et al. [ACRZ97] showed that a large consistency unit (like a hardware page) is not detrimental of performance if a relaxed consistency and a multiple writer protocol is used.

Protocol writer-owns [FA96] improves the performance of LRC by remapping subpages in a page, such all sub-pages are written by the same process.

Instead of relaxing the consistency, Itzkovitz and Schster [IS99b, IS99a] also use sequential consistency. They detect pages with false sharing and map the conflictive sections of the same page to different virtual pages at runtime, avoiding the false sharing completely. This technique is known as the EmFiGS approach. But this technique has been proven ineffective by Kudlur and Govindarajahn [KG04] due to the overhead of managing the extra virtual pages and the reduced exploitation of spatial locality.

Alexander et al. [ACCL00] developed a DSM called ULTRA that uses dynamic granularity to eliminate false sharing. The accesses to the shared address space are sequentialized with the use of tokens that represent a specific region of memory.

Our work uses the idea of scheduling loops taking into account the affinity

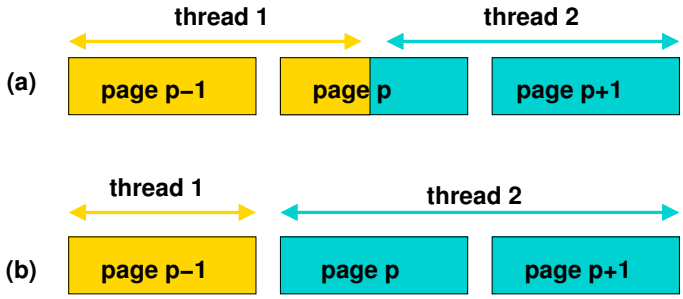

Figure 3.2: Iterations per thread (a)before and (b)after aligning to the boundaries.

to local data [Mar94] and it is heavily influenced by the work of Granston and Nikolopoulos, extending them to the runtime arena.

3.4 Mechanism

As we have already mentioned, our approach does not consist on modifying the source code 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 one of the cooperations we have already implemented and tested.

3.4.1 Boundaries alignment

As most parallel loops are executed more than once, our proposal consists of scheduling the iterations 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 a side effect this schedule produces some load unbalance. Figure 3.2 (a) shows an example, where two threads execute a static schedule and a page `P` is shared between both processors. After the alignment (Figure 3.2 (b)) the page `P` is not shared anymore. 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.

3.4.2 Design issues

In this section, we explain how to use the new *ALIGN* scheduler and the features needed in the DSM and in the runtime.

User: SCHEDULE directive

In order to use the new scheduler in a parallel loop, the OpenMP programmer uses the *SCHEDULE* directive. We propose the following syntax:

```
1 !$omp SCHEDULE ( ALIGN, <schedule>, <operation> )
```

The `schedule` parameter is a number identifying the schedule to be used in the following parallel loop. It can be a new or previously calculated one. The `operation` parameter manages what to do with the schedule: 1) *TRAIN*, to learn the iterations per page and detect the page boundaries to build a new schedule that will be reused later by the current or other loops, or 2) *REUSE*, to reuse a schedule calculated previously.

Example

Figure 3.3 shows an excerpt from the CG source code modified to use the *align* scheduler.

Two parallel loops are being executed 25 times. Both loops use the same schedule, which is calculated in the second loop after a warm-up of 2 iterations of the outermost loop. The first parallel loop uses a static scheduling in its first execution because the schedule is not yet calculated. The schedule clause uses a variable (`operation`) to select what operation to do: *train* or *reuse*. This is useful to do some warm-up (2 iterations in this case), learn the right schedule and reuse it afterwards.

Runtime: The ALIGN scheduler

The *ALIGN* scheduler follows these steps to compute a schedule of iterations taking into account the page boundaries:

```
1         operation = TRAIN
2        do cgit=1, 25
3 !$omp  parallel do default(shared) private(j,k,sum)
4 !$omp& schedule(ALIGN, 1, REUSE)
5           do j=1,lastrow-firstrow+1
6              sum = 0.d0
7              do k=rowstr(j),rowstr(j+1)-1
8                 sum = sum + a(k)*p(colidx(k))
9              enddo
10             q(j) = sum
11          enddo
12
13 !$omp  parallel do default(shared) private(j)
14 !$omp& schedule(ALIGN, 1, operation)
15          do j=1, lastcol-firstcol+1
16             p(j) = r(j) + beta*p(j)
17          enddo
18
19          if ( cgit .eq. 2 ) then
20             operation= REUSE
21          endif
22        enddo
```

Figure 3.3: Fortran code example for the Align scheduler use.

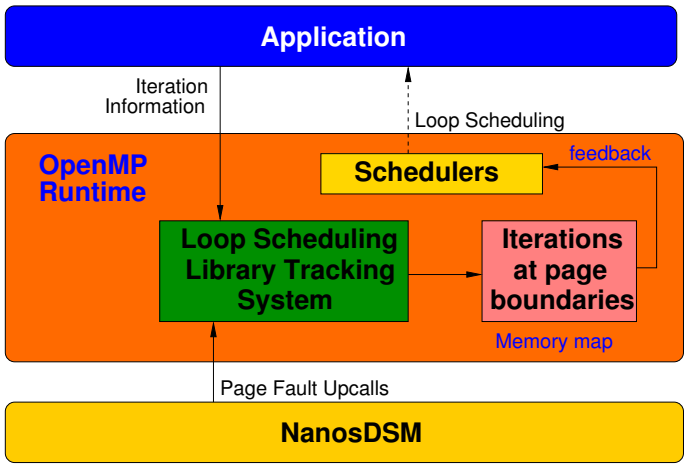

Figure 3.4: Components that take part in the alignment mechanism.

1. Register the memory regions where write accesses are done, so it can detect the false sharing and avoid it. 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. The idea is to establish a communication between the SDSM layer and the OpenMP runtime in a way that the SDSM notifies the runtime about a page fault and its associated information. This notification mechanism is what we have defined as the *upcall mechanism* and it is explained later in Section 3.4.2. In order to avoid the overhead of being notified at all the memory page faults, the mechanism offers functions to register a specific region of memory where faults should be notified.

2. When a page fault occurs, the SDSM sends an upcall, 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 used 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. Also, this schedule calculated for an specific loop can be applied to other parallel loops which will reuse the same mapping between iterations and threads exploiting any temporal locality available.

The modules in the Nanos OpenMP runtime taking part into the alignment mechanism are presented in Figure 3.4. There is a module that will gather the page faults and the iteration information, building a memory map for the current parallel loop. This information is used to detect the iterations at page boundaries and construct a new schedule accordingly to be reused later.

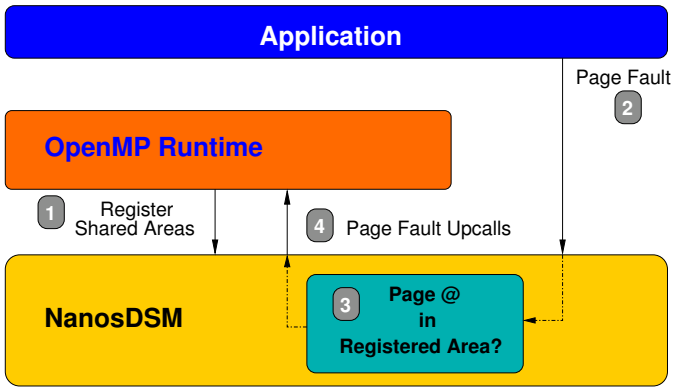

Figure 3.5: Components that take part in the upcall mechanism.

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

DSM: Support to allow cooperation with higher levels

The most important feature to support the cooperation consists on offering upcalls [Cla85]. This mechanism, shown in Figure 3.5, 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 these 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.

This upcall mechanism is not being thought to be used by regular programmers, but by runtime implementors, compiler developers, etc. This mechanism should be transparent to regular applications.

Register a shared memory region. To notify page faults in a shared memory region, the region should be registered with a function to be called whenever the page fault occurs. The register function has the following interface:

```
1 int msm_set_region_pf_notification( char *region_start, long size,
2                         void (*upcall) (char *, char *, int, int))
```

where `region_start` and `size` are the region starting address and its length in bytes; and `upcall` is the address of the upcall function to be called.

A list of registered regions is maintained and the register function will return a positive value with the new assigned region identifier, or a negative value meaning that the region overlaps a previously registered region.

Page fault in a registered region. When a page fault occurs, the list of regions is checked. If the page fits in a region, the corresponding upcall function will be called with the address being faulted, the instruction code that produced the fault, its type (a read or a write fault), and the selected region id.

3.5 Evaluation

3.5.1 Methodology

The evaluation of the page alignment scheduler is divided in two parts. First, a preliminary evaluation of the mechanism with a couple of benchmarks just to prove that the mechanism works. And, second, a more comprehensive evaluation with more benchmarks at chapter 6.

The benchmarks used in this section are the Conjugate Gradient (CG) from the NAS benchmarks [JFY99], and the Ocean kernel from the Splash2 benchmark suite [WOT+95,SWG92]. They are executed in the MareNostrum cluster. On one hand, the CG benchmark is an application which suffers from false sharing that can be solved with the align scheduler. On the other hand, the Ocean benchmark also has false sharing, but it can not be solved with this mechanism. It is an example showing the worst case and that the mechanism is able to detect when it will not be useful and it simply uses an static schedule adding very low overhead.

For each benchmark two versions are evaluated:

Static The unmodified original source code of the benchmark, which acts as the baseline.

Align The static version extended with the align scheduler in the parallel

	CG	
	Class A	Class B
na	14000	75000
nz	11	13
niter	25	75

Table 3.1: Parameter values for A and B classes of CG benchmark.

Data structure	Size
p	na
q	na
r	na
z	na

Table 3.2: Main data structures for CG benchmark.

Loop	p	q	r	z
MVP	R	W		
DP	R	R		
AXPYDP	R	R	RW	RW
AXPY	RW		R	

Table 3.3: Summary of CG data structures modified by each loop.

loops that suffer false sharing.

CG Benchmark

The CG benchmark kernel has four consecutive parallel loops (i) matrix-vector product (MVP), ii) dot-product (DP), iii) AXPY/Dot-product combination (AXPYDP) and iv) axpy (AXPY)) that are executed a fixed number of times (niter) determined by the benchmark class. The experiments uses the classes A and B as distributed in the NAS benchmarks suite, Table 3.1 summarizes the values for the *na, nz* and *niter* parameters.

These loops modifies the main data structures (Table 3.2) following the access pattern presented in Table 3.3, where for each loop we show if the corresponding variable is read (R), written (W) or both (RW). The table shows that the `p` and `q` variables are the most used structures, which are written in a loop and then they are read by the others, AXPY and MVP loops respectively.

	CG			
	Class A		Class B	
Nodes	Static	Align	Static	Align
Seq	12.82	12.82	1399.71	1399.71
1	14.04	13.57	1533.74	1526.76
2	11.81	10.89	524.96	523.43
4	9.70	8.16	375.04	366.88
8	9.27	8.33	329.42	323.75

Table 3.4: Execution time (seconds) for A and B classes of CG benchmark.

Ocean Benchmark

The Ocean benchmark uses a square matrix `A` of 2048x2048 float elements to represent the set of 2D point planes. The benchmark uses one parallel loop (MAIN) to calculate the weighted average and this loop is executed until the difference for all points arrives to an specified tolerance level.

Testbed

The benchmarks have been executed on top of the MareNostrum cluster (see Section 2.7.4) with 8 nodes using the Full-Duplex Gigabit Ethernet network. Even we could use a maximum of 4 threads per node, just one thread will be used for all benchmarks, because the problem already appears. All benchmarks have been run with 2, 4 and 8 nodes.

3.5.2 Results

CG Benchmark

The execution times for the CG benchmark are summarized in the Table 3.4.

The speedup for the CG class A is shown in Figure 3.6. The speedup for the static version is quite low, arriving at a maximum of 1.38 with 8 nodes. The small size of the matrix provides small computation time per thread. Threads can access few pages in this time, and therefore the boundary pages (pages that share the last iterations from one thread and the first iterations

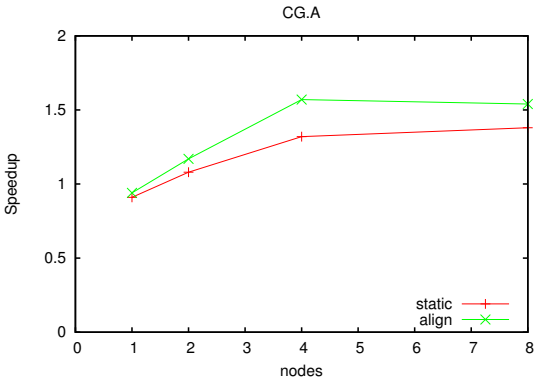

Figure 3.6: Speedup of CG class A for original and aligned versions.

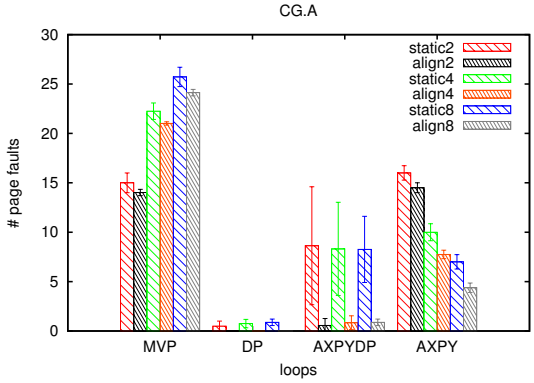

Figure 3.7: Number of page faults per loop for static and align version of the CG Class A benchmark.

of another) are accessed at the same time, making threads to compete for them. This ping-pong effect trashes the execution time.

Figure 3.7 shows the average number of page faults for the four main benchmark loops. For each loop, it presents the static and align versions using 2, 4 and 8 nodes respectively. Versions are grouped together to ease their comparison.

As the figure shows, the third parallel loop in the static version has a great number of page faults with a big variation, basically due to the iteration-page boundaries mismatch. These page faults represents, on average, a 42% of the pages faulted in the first loop.

The align scheduler learns a new schedule at the fourth loop, when a single structure is written, and the resulting schedule is reused in the remaining loops. This new schedule is able to completely eliminate the false sharing in the third loop and to exploit a better spatial locality in the other loops. It removes the 92% of the faults in the third loop, getting an average improvement of an 11% in the execution time of this class. Even though the number of page faults has been greatly reduced, the impact on the final execution time of the application is not as important, this is easily explained because the number of page faults is not the only thing to consider. Figure 3.8 shows the average time used by both versions at each loop, it is divided between

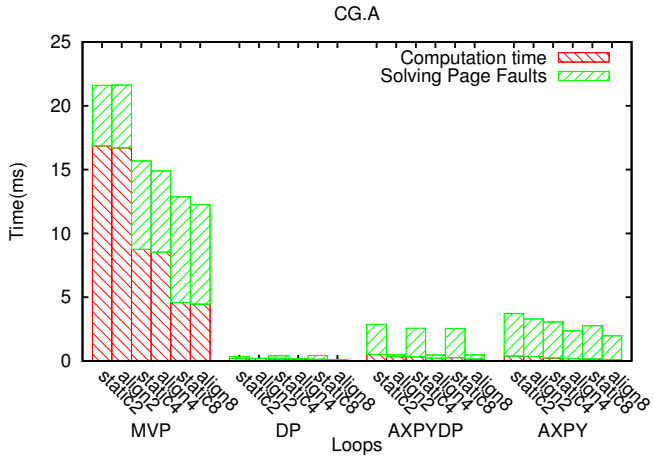

Figure 3.8: Average time used in the computation phase and solving page faults at the different loops when executing the CG class A benchmark.

time computing data and time solving page faults. As the figure shows, the third loop suffers from false sharing, but it is not the most consuming one.

To better understand the false sharing problem and the effect of the align scheduler, Figure 3.9 shows the faults in the memory address space of the application for static and align versions when executed with 4 nodes. The address space shown includes the four main variables `p, q, r` and `z`. The figure shows two Paraver [PLCG95] histograms for the static (upper) and the align (lower) versions. Each column represents a memory address, and each row represents a node. A gradient of color represents the number of faults for each address by each node (darker color means more faults).

Page faults shows that one of the structures is used by both versions, while the other three structures are faulted at the boundaries in the static version. But these faults disappear completely in the align version due to the alignment of the iteration space to these page boundaries.

In contrast, both versions have similar speedup when they use the bigger class B (Figure 3.10). There are two things that explain this similarity. On one hand, the number of page faults in the third loop does not change as much as in the smaller class, meaning, that even the false sharing problem exists, there is enough computation time between consecutive faults from different threads to avoid the ping-pong effect. The page faults per loop

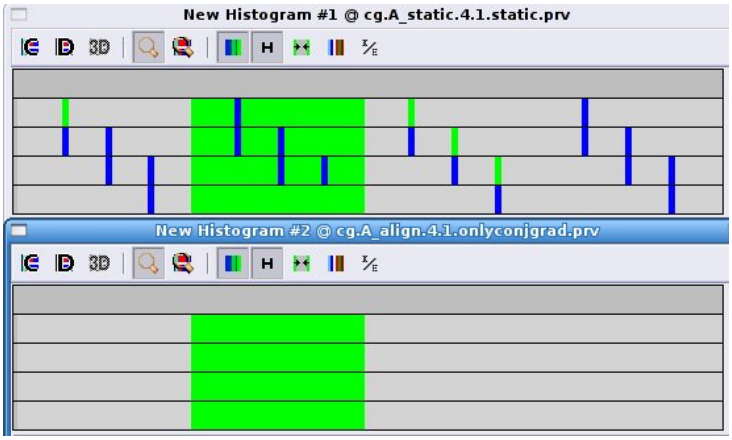

Figure 3.9: False sharing effects in the number of page faults for the CG benchmark executed with 4 nodes: light green corresponds to few page faults and dark blue to many page faults. The static version shows 4 structures being accessed, all of them with false sharing, while the aligned version do not have this problem.

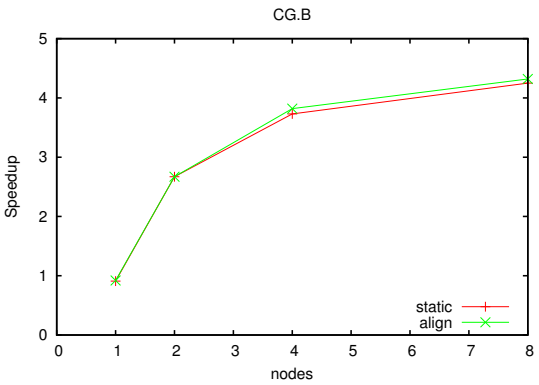

Figure 3.10: Speedup of CG class B for original and aligned versions.

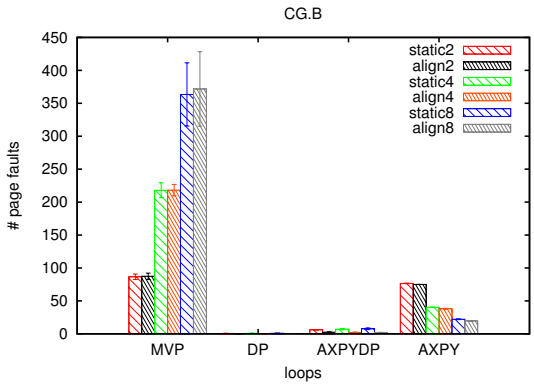

Figure 3.11: Number of page faults per loop for static and align version of the CG Class B benchmark.

corroborate this theory (Figure 3.11). On the other hand, the third loop has a small number of page faults compared with the first one (which has the highest number), representing just a 4% in average. So, even the scheduler solves the false sharing problem, it does not improve the final execution time, because it is not a relevant problem anymore.

Ocean Benchmark

The Ocean benchmark presents a potential horrible situation for a SDSM, which is a true sharing among nodes. Many different cells in the array are read by one node and written by another. This implies that there are not explicit boundaries at all, because no matter how we split the computation, some elements on one side will be written by the nodes assigned to the other side. Our mechanism is able to detect this situation and it disables the alignment, using a simple static schedule.

Even a static schedule is used, the speedup of this benchmark is quite good, with a maximum of 6 with 8 nodes (Figure 3.12). This is easily explained with the regularity of this benchmark. After the first execution of the main loop, the placement of main part of the pages remains stable, except for pages suffering from true sharing, which are the ones that keep faulting. Figure 3.13 shows the average number of page faults per loop, showing the static and align versions executed with 2, 4 and 8 nodes. As expected, the number of page faults increases with the number of nodes, because there are more boundaries. But, at least, the number of faults per node remains stable. We can also observe that the number of faults do not change when using the align scheduler, because the same static schedule is used.

Execution times for both versions are shown in Table 3.5. As a curiosity, the align mechanism seems to get better speedup than the static version alone, even they do exactly the same schedule. A closer look to the generated code showed that the align version uses more parameters, and so it has a different alignment in the stack frame. This difference in the stack alignment explains the slight difference in the speedup, a deeper research is needed to understand why is this behavior happening.

	Ocean	
Nodes	Static	Align
Seq	15.18	15.18
1	15.25	14.29
2	7.92	7.49
4	4.41	4.21
8	2.59	2.5

Table 3.5: Execution time in seconds for original and align versions of the Ocean benchmark.

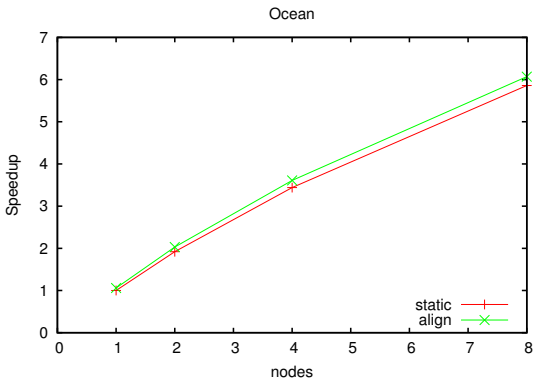

Figure 3.12: Speedup of Ocean benchmark for original and aligned versions.

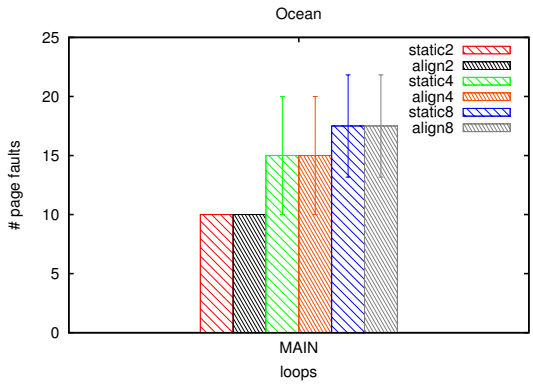

Figure 3.13: Number of page faults per loop for static and align version of Ocean benchmark.

3.6 Conclusions

In this chapter, we have presented the false sharing problem: when two processors, in a parallel system, access the same cache line at different offsets and, at least, one of them is a write. We have shown that the false sharing in a sequential consistency DSM is a problem when different threads try to access the same page at the same time, producing a ping-pong effect that may trash the final execution time.

A typical solution to solve this problem at runtime is to relax the memory consistency, and therefore the false sharing vanishes due to the multiple writer protocol. We think that there are cases where the complexity and the cost necessary for this relaxation are not worth it.

The problem is that this false sharing is due to the OpenMP runtime who decides an iteration distribution without taking into account the underlying DSM. We have proven that in these cases it is not necessary to relax the consistency but to obtain a better distribution of the iterations.

Therefore we have designed a new *Align* scheduler for OpenMP parallel loops that avoids this false sharing problem at runtime.

The scheduler needs a tight cooperation between DSM and OpenMP runtime layers, and we propose a mechanism of upcalls to communicate with the DSM layer.

The scheduler has been used in a couple of benchmarks, showing that, on one hand, when the false sharing produces trashing it is able to avoid it and it reduces the execution time by an 11% on average. On the other hand, when the scheduler is not able to find a better schedule, it adds small overhead and it obtains similar execution times than an static schedule. We have also shown that the false sharing is a problem if the computation time available is small, and that if the computation time is big enough its effects are not relevant.

The results of this contribution have been published in [CCM+04,CCM+06].

Chapter 4

Apply lessons learnt from MPI

La mejor defensa es un buen ataque

Sun Tzu

Abstract

One of the main problems when executing an OpenMP application on top of a distributed platform with a DSM is the overhead of the remote memory accesses. In this chapter we design and evaluate a new mechanism to overlap the computation of an OpenMP parallel loop, with the communication of its referenced memory accesses. We emulate the data flow from MPI, where a node forwards data it owns to the nodes that will need this data after finishing a parallel loop. Results show that a sequential consistency DSM with this technique achieves similar performance results than a relaxed consistency DSM like TreadMarks.

4.1 Motivation

In a DSM environment, an access to an address that is not locally accessible has a high overhead. It involves a segmentation fault signal captured by the DSM layer, a request to the manager of the page containing that address and wait the network latency for the response.

A typical approach to minimize this overhead is by avoiding the fault in the first place, for example, by placing the data before the offending access: the *prefetch* technique.

The idea is simple, imagine an OpenMP application with the code presented previously where a parallel loop traverses an array:

```
1 !$omp parallel do
2     do i=1, N
3         A[i] = ...
4     enddo
```

When this application is executed, each thread accesses different memory pages due to the organization of the memory in physical pages. Each accessed pages that is not in the local memory will generate a page fault with the corresponding overhead to receive the page content. Figure 4.1 a) shows the timeline of one thread where pages `P1, P2` and `P3` are accessed sequentially.

This specific pattern of page accesses in an application parallel region can be learnt by each thread, and when this pattern is detected again, the whole set of accessed pages is brought locally to the thread avoiding posterior page faults, as shows Figure 4.1 b) where the previous pages are gathered after faulting the first page.

This way, the first access has a slightly bigger overhead, but the following extra memory accesses are avoided.

This solution follows a consumer-driven model, where the consumer requests necessary data from the producers. Even though this approach has been proven as a successful model, it has some limitations: (1) data may be requested before all nodes working on it have finished, producing some unwanted results; and (2) the most important, data is not consumed immediately after being produced, so there is still place to reduce latency.

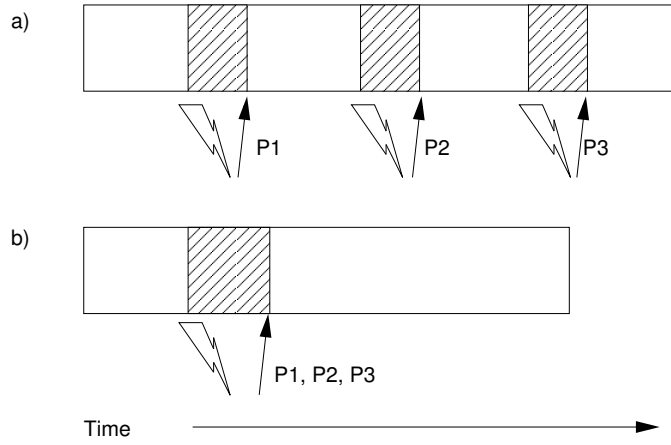

Figure 4.1: Prefetch technique in a loop.

4.2 Thesis

We think that the performance of an OpenMP application being run on top of a DSM could be improved if we follow a producer-driven model, similar to MPI, where producers send their new data directly to the consumers.

4.3 Related Work

A lot of research has been done to reduce the high memory latencies to access remote nodes through the interconnection network, but they reduce to three main topics: overlap the communication with the computation, relax the consistency or transform the code.

4.3.1 Overlap communication and computation

One of the main ideas at hardware architectures with different levels of memory hierarchies to overlap data communication with computation is to fetch data locally before accessing it, the *prefetch* technique [VL00]. The tecnique described here is used by programmers to specify which data needs to be fetched.

Jégou [Jég00] also opts for using extracted information from the compile-time analysis to predict variable accesses and include prefetch requests. This

is different from our proposal where the prediction is done at runtime and the data is sent from the producers instead of requested from the consumers.

Instead of moving the data to the nodes that will do the computation, Saltz et al. [SBW91] propose to change the schedule so each node computes on its local data, the inspector-executor method. A loop is divided into two code segments: the inspector and the executor, where the inspector preprocesses the loop to gather the best schedule and then the executor executes the loop using the calculated schedule.

Instead of using this inspector-executor method, data accesses information can be generated at compile-time and the runtime can precompute the set of pages that will be accessed, fetching them before each iteration execution as Lu et al. [LCD+97] prove to improve irregular applications.

The main problem with distributed memory is that when nodes accesses remote memory, a lot of coherence messages are sent and received to update the local memory view. Keleher et al [Kel99] proposes a mechanism that captures this coherence traffic inside a region of code and it allows to replay it afterwards in next executions of the same region, the Tapeworm. This is similar to our solution with the difference that is the data consumer who requests the data instead of the producer as we propose. Other minor differences is that we use the page content instead of the coherence messages and we use sequential consistency.

4.3.2 Relax consistency

Another idea to reduce the network latency is to relax the consistency, delaying all memory updates till a synchronization point is reached. Treadmarks [KCDZ94, Hoe06, TMSW08] implementing a Lazy Release Consistency with multiple writers is the most successful DSM relaxing the consistency. Additionally, Carter et al [CBZ95] presents a similar idea as our pre-invalidation mechanism, but they use a timeout and therefore they could make a wrong decision.

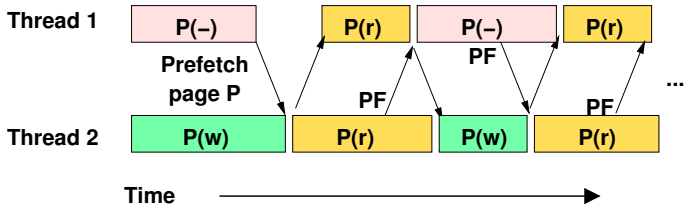

Figure 4.2: Prefetch problem, the prefetched page is still being used.

4.3.3 Transform the code

Finally the last option is to transform the source code to a real distributed environment like MPI. This is the option used by Basumallik et al [BME02] who transforms the code mixing an SPMD with Treadmarks, and pointing out that data-forwarding should be preferred over prefetch, because the former is a one-way communication (producer forwards to all consumers) whereas the latter is two-way (consumers request data and producers respond). But finally, he opts for transforming the whole code to MPI [BE05, BE06].

4.4 Mechanism

In this section we present the idea of two mechanisms to achieve the sender initiated data flow and also the issues needed to design these features in our environment: the presend and the preinvalidation.

4.4.1 Presend

In order to overlap data movement with computation, it is necessary 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 4.2 where two threads appear. One of them prefetchs the p 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.

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 upcall 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.

4.4.2 Preinvalidation

A very similar problem consists on invalidating the copies of a page once a node wants to modify them. This task is also time consuming and it would

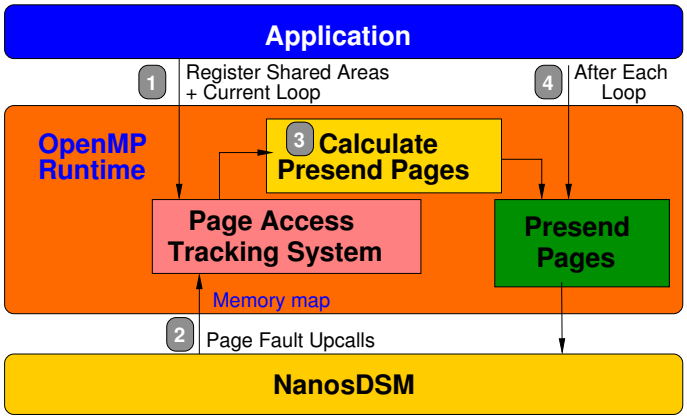

Figure 4.3: Components that take part of the presend mechanism.

be desirable 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 this copy and inform the page master that this copy does not exist 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 their copies, 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.

4.4.3 Design issues

The three main components of the presend mechanism are shown in Figure 4.3: the OpenMP application, the OpenMP runtime and the software DSM. This section presents what needs to be done at each component to implement the presend mechanism.

```
1 !$omp  parallel do default(shared) private(j,k,sum)
2 !$omp& PRESEND
3           do j=1,lastrow-firstrow+1
4              sum = 0.d0
5              do k=rowstr(j),rowstr(j+1)-1
6                 sum = sum + a(k)*p(colidx(k))
7              enddo
8              q(j) = sum
9           enddo
```

Figure 4.4: Fortran code example using the Presend directive.

User: PRESEND directive

We propose a new OpenMP directive `PRESEND` to enable the presend/preinvalidation technique on a parallel loop with the following syntax:

```
1 !$omp PRESEND
```

Only the threads that execute a parallel loop annotated with this directive will detect page faults during its execution and, at the end of the loop, will presend or preinvalidate all detected pages.

On one hand, the directive adds flexibility to the OpenMP application, allowing the user to decide if the technique should be applied or not to each parallel loop.

On the other, there are situations where the presend technique is not desirable. For example, there are parallel loops where:

1. the presend can be useless, because the results of the presend can not be exploited; for example if the loop is executed just once or the accessed data is completely different at each iteration, avoiding any benefit from the technique.

2. the presend can be unnecessary, because the computation time is too small or the accessed data is not relevant.

We think that these reasons justify the use of a directive per parallel loop instead of applying the presend technique to all of them.

Example

Figure 4.4 shows an OpenMP parallel loop using the PRESEND directive extracted from the CG NAS benchmark source code. It is a loop that updates all positions of an array (`q`) with the result of summing up some positions of another array (`p`) through indirect accesses. The directive enables the monitoring of all memory references inside the parallel loop and presend them as needed at the end of the loop.

Runtime: Parallel loops

As the figure 4.3 shows, the OpenMP runtime, which has a tight cooperation between the application and the DSM, needs to:

1. **Learn the sequence of parallel loops.** The parallel loops are identified and a simple one level detector is used to detect their sequence order. A parallel loop delimits a region of code where page faults can be associated, we use the term *context* for this region, and Section 5.4.2 in next Chapter gives more details.

2. **Build the list of pages faulted inside each annotated parallel loop.** The upcall mechanism will notify each thread with all pages faulted inside a monitored loop. All threads will send this list of pages to a master node when the threads finish the parallel loop. This master node will build a global list with these lists, containing the pages faulted and their protections for each parallel loop.

3. **Build the list of pages that each thread should presend/preinvalidate.** Finally, with these global lists, the master node will send to each thread the set of pages that should be pre-sent or pre-invalidated after the current loop. For each page in the loop, the master node calculates which is the next loop that uses that page for reading or writing and so it calculates a presend or a preinvalidate request accordingly.

DSM: Presend

The presend mechanism allows the content of a page to be sent asynchronously to a list of destination nodes. Each thread can use it to distribute a page it owns (meaning that it has read or write permissions for this page) to other nodes. This is specially useful to distribute new produced content to its consumers.

A thread issuing a presend on a page sends a request to the master node of that page. The master node will get a copy of that page (in case the master did not have any copy) and it will send copies to all nodes in the destination list. Copies will not be sent if the destination node already has the page, this can be the case if the node faulted the page before the presend request.

DSM: Preinvalidation

In a sequential consistency DSM with a write-invalidate protocol, all nodes that have read the value of a variable will receive an invalidation for the page containing that variable when this variable is updated by other node. The preinvalidation mechanism allows a node to invalidate the local copy of a page in advance, notifying the master node that his copy is not valid anymore.

4.5 Evaluation

4.5.1 Methodology

We used the CG benchmark, explained in the previous chapter at section 3.5.1, to test these mechanisms.

We evaluate the results from three different versions:

Original The original version without modifications.

Align This version corresponds to the *Original* version with the SCHEDULE directive added to avoid false sharing (like in chapter 3).

Presend *Presend* version is the *Align* with the directive PRESEND enabled.

The different versions are executed in Kandake and Crossi clusters (see section 2.7.4), with one thread per node.

4.5.2 Results

The CG 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 written by some node in a loop and read 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, four different graphs are presented. The first one (Figure 4.5) shows the average number of page faults at each parallel loop for original and presend versions. Afterwards, Figure 4.6 shows the behavior of the class B of this benchmark on Crossi. Then, the behavior of the same benchmark in a smaller class (A) on the same machine (Figure 4.7) is presented. 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, CG class A is re-executed on Kandake and compare its speedup with the one obtained by TreadMarks (Figure 4.8). 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.

Figure 4.5 presents the average number of page faults in the main parallel loops of the CG class A benchmark. As the figure shows, the presend clearly reduces the number of page faults in the first and last parallel loops (MVP and AXPY), which are the loops that takes more execution time, as it will be clearly shown in Chapter 6.

The CG class B on Crossi shows that a good speedup can be achieved (Figure 4.6). 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.

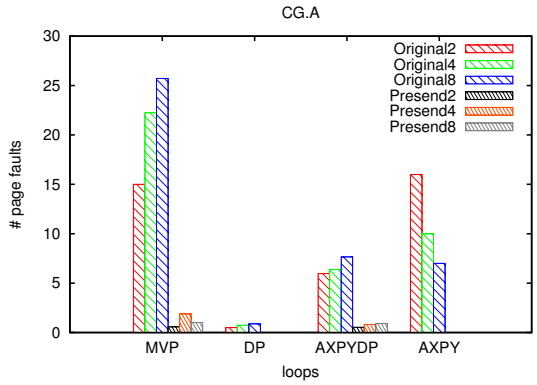

Figure 4.5: Number of page faults at each parallel loop in the CG class A benchmark for original and presend versions when they are executed with 2, 4 and 8 nodes.

	Execution time (seconds)		
Nodes	Original	Align	Presend
Seq	1421.19	1421.19	1421.19
2	–.–	–.–	–.–
3	–.–	–.–	–.–
4	378.94	371.32	350.14
5	343.22	310.18	302.66
6	–.–	–.–	–.–
7	297.11	280.10	267.79

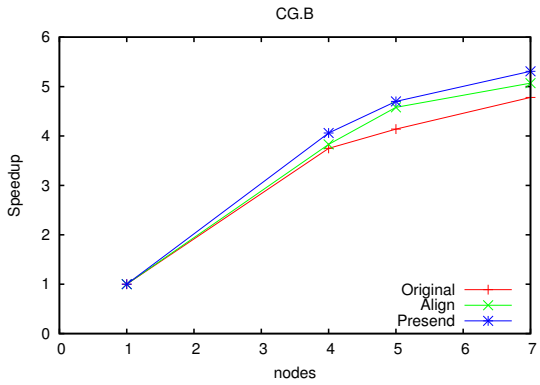

Figure 4.6: CG class B at Crossi.

	Execution time (seconds)		
Nodes	Original	Align	Presend
Seq	7.0	7.0	7.0
2	7.03	5.47	4.34
3	7.37	5.25	4.21
4	22.66	5.47	4.35
5	22.13	5.65	4.54
6	24.19	6.00	4.76
7	26.28	6.22	4.97
8	22.20	6.67	5.38

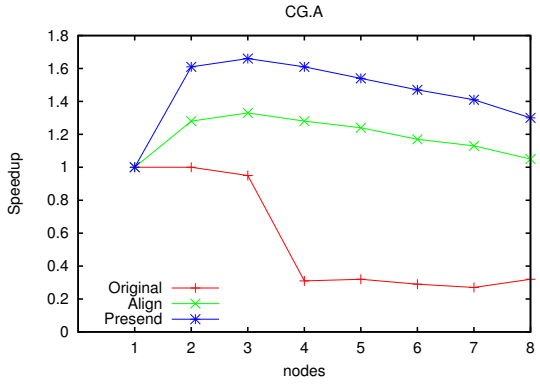

Figure 4.7: CG class A at Crossi.

When executing the same benchmark but using a smaller dataset on the same machine (Figure 4.7), 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 is solved in Chapter 6 where this situation is detected and the reading threads are informed with the written value avoiding any page fault. Even though the load balance has improved the performance a lot, as iterations are divided 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, the execution of the benchmark is repeated on Kandake (Figure 4.8). The objective was to compare the *presend* speedup with the one observed when the "same" application is run on TreadMarks. The TreadMarks version has been tested only on this machine because a license was available for this machine. Its source code included a version of the CG benchmark. This version is similar to the MPI version of the NAS benchmark, but it has been modified to be executed on top of Treadmarks. It is also modified to do a preliminary redistribution of the data and some other tricks to improve its performance. It is not fair to compare this version with the original OpenMP used for the tests, but it will give a reasonable idea of the results that can be accomplished.

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

When comparing the *presend* behavior with the one achieved by TreadMarks, it is easy to observe that the presend does it as well as they do but without using a relaxed-semantic SDSM. In addition, it is important to remember that the CG executed in TreadMarks is not the OpenMP version,

	Execution time (seconds)			
Nodes	Original	Align	Presend	TreadMarks
Seq	89.0	89.0	89.0	89.0
2	79.85	56.7	52.15	60.93
3	60.13	46.39	41.16	43.04
4	70.46	43.29	37.27	35.53
5	203.8	42.74	37.36	32.19
6	255.75	42.17	37.20	31.47

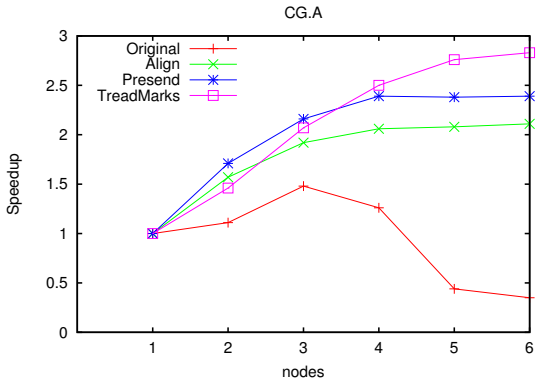

Figure 4.8: CG class A at kandake.

but a version specially coded for TreadMarks. Finally, the Treadmarks increases 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.

4.6 Conclusions

In this chapter, we have designed a mechanism that follows a producer-consumer model similar to MPI, where the producers send data to the consumers before they need it. The goal of this mechanism is to reduce the latencies of the accesses to remote pages.

The mechanism offers two different functions: presend and preinvalidation. Presend sends copies of a page to a list of nodes, and preinvalidation invalidates the local copy of a page.

The OpenMP runtime uses these functions together to forward all data produced in a parallel loop to the nodes that will need this data in the next loop. So, the consumer loop avoids the majority of page faults because the data will be already there. The communication of these pages overlaps with the computation of the current loop.

The experimental results show that the number of page faults is highly reduced compared to the baseline. It also shows that the performance of a sequential consistency DSM using a sender initiated communication is similar to the performance obtained by a relaxed consistency DSM like TreadMarks.

The contributions from this chapter have been published in [CCM+06, CCM+04].

Chapter 5

Avoiding network congestion

No dejes para mañana lo que puedas hacer hoy

Refranero español

Abstract

A typical problem when running OpenMP applications on top of a DSM is that, at synchronization points, the network can saturate when the system sends high amounts of control messages and pages trying to maintain the memory coherence. The limited bandwidth of network and the effect of concentrate communication at single points in time produces network congestion. This situation may limit or degrade the final performance of the application.

This chapter discusses a method to distribute this memory coherence messages during the computation phase and send them as early as possible avoiding the congestion.

5.1 Motivation

The main problem of page-based DSMs is that they need to communicate a high amount of pages through the network in order to maintain the coherence of the global shared memory across the nodes in a cluster.

In OpenMP applications, this communication is usually produced at the OpenMP synchronization points, namely implicit and explicit barriers or locks.

At these points, the required bandwidth to send these pages may be greater than the available in the network, generating some temporal congestion. The number of network messages is directly related to the number of memory modifications, because they are basically coherence messages. So, applications that modifies a high amount of data will send a high amount of network messages.

The TCP protocol tolerates this congestion using a *congestion avoidance* algorithm [APS+99] and the data will be eventually sent but it will not be transferred immediately as needed. Even though this algorithm is effective, it affects negatively the final application performance because it limits the quantity of data sent at the same time, reducing bandwidth and adding some timeouts.

The underlying problem is that the quantity of data to be transferred at these points is greater than the data that the network can transmit and this affects negatively the final performance results.

5.2 Thesis

The solutions to network congestion problem are mainly two: (1) reduce the quantity of data sent, or (2) distribute the data to avoid the congestion.

The main idea in this work is to avoid the transient network congestion situations by distributing coherence messages that produces the congestion as early as possible during the computation time, taking profit from the potentially idle network.

5.3 Related Work

The topic of network congestion has been extensively researched in the network area, like the TCP congestion algorithm [JK88, APS+99, AFP02], where a congestion window with the acknowledged messages is used to slowdown the client. The work presented here is somewhat related but instead of slowing down the client, we identify earlier points in time where data could be distributed.

As far as we know, in the DSM literature nobody else has been doing any research on distributing data delivery across time to avoid network congestion. However, there are different works that allow programmers to synchronize explicitly different regions of memory and therefore they could be used to distribute manually the coherence messages.

The OpenMP specification offers a *flush* operation [HDS08] to enforce consistency between a thread's temporary view and global memory, it affects the whole memory or just the data specified in the parameters. Our proposal extends this operation to automatically detect what needs to be synchronized without the user intervention.

Another idea is developed by Iftode et al [ISL98] who allow the synchronization of all modified data inside a *scope*, where the term *scope* refers to a region a code. They call it the Scope Consistency model. Inconveniently, an improper use of this technique produces an incorrect code.

There are other studies, like Gornish et al. [GGV90] which shows mechanisms to find at compile time the earliest time to start prefetching data taking into account data and control dependences, but the question of how to avoid the network congestion remains unanswered.

5.4 Mechanism

This section presents the *chopper*, a mechanism that can detect when a thread access a specified page by the last time.

5.4.1 Chopper

Network congestion appears because a high number of network messages are sent at the same time, therefore to avoid it, it is necessary to distribute these network messages during the computation phase.

These network messages are directly related to coherence messages, and in our DSM these coherence messages corresponds to pages because the memory granularity is a page.

The ideal situation to distribute the coherence messages would be if the DSM was able to detect the exact time when a thread finishes working on a page. After finishing a page, the DSM could send the coherence data messages automatically, without waiting for any synchronization point. The problem is that DSMs usually do not know when a thread finishes working on a page.

The solution: Our proposal gives a solution to this problem: DSMs can detect the exact interval of time where a thread finishes working on a page, allowing the distribution of coherence messages before synchronization points.

The idea is to divide a given interval of execution time into smaller sections and monitor the page faults inside each section. Therefore, we can detect which is the last section using any page and send the coherence messages for a page before the usual synchronization points, exactly at the end of the last section using it.

How is the solution implemented: The starting point for our mechanism is an OpenMP application that has some regions which suffer network congestion at synchronization points and therefore the chopper algorithm follows these steps:

1. Mark the regions suffering network congestion.
2. Learn the sequence of regions.
3. Detect when a thread has finished working on a page, by dividing a region into sections and detecting all page faults inside the sections.
4. Calculate pages that can be sent after each section.

5. And finally, distribute the list of pages using the sections.

Actually, our mechanism needs some manual modifications in the source code to mark these regions and to divide a region into smaller sections where the network messages can be distributed. We use the pre-send techniques explained in previous Chapter 4 to execute this distribution.

Marking regions with network congestion

The regions which suffer network congestion are marked and identified. This allows the page-fault detection inside the regions and the posterior distribution of their related coherence data.

These chopper regions can contain different parallel loops and sequential sections, in contrast to the previous chapter where the regions where limited to parallel loops.

These regions correspond to contexts inside the DSM and they are used by the pre-send mechanism later.

Detecting when a thread has finished working on a page

It is necessary to find a mechanism that detects when a thread finishes working on a page in order to avoid the congestion and send the page modifications as soon as possible.

The DSMs have information about the first reference to a page after a synchronization point and the next synchronization point, but nothing else is recorded in between. For example, after a synchronization, a thread, with an invalid copy of a page that wants to update it, accesses the page and a page fault is generated. The page fault notifies the DSM that a thread wants to access that page. But, once the DSM retrieves that page with the right protections and gives a copy to the thread, the DSM will not be notified anymore about any posterior access to this page done by this thread.

Given this scenario, the thread usage of a page is limited by the first time the thread references the page generating a page fault and the synchronization point after the page fault.

Our proposal consists in dividing the computation phase suffering the network congestion into smaller sections. These smaller sections will allow the detection of sections where the page is used and, even more important, in which sections the page is unused.

The ***virtual synchronization points*(VSP)** are the proposed mechanism to divide a region into smaller sections. The OpenMP runtime will use them as extra points to distribute coherence data earlier than the congestion point.

Each VSP behaves like a synchronization directive, so it synchronizes all data that is not referenced anymore inside current region or it is silently ignored. They are simple hints to the OpenMP runtime which will decide their final semantics.

5.4.2 Design issues

As in the previous Chapters, this mechanism requires a tight cooperation between the OpenMP and the DSM runtimes. This section presents the changes needed to implement the chopper mechanism.

User: Directives

We propose the directives shown in Figure 5.1 to use the chopper mechanism. The user annotate an OpenMP application with the `start_region` and `stop_region` directives to mark and identify a region of code, and the `vsp` directive to insert a new VSP inside the code, slicing the current region in two sections or *chops*.

```
1 !$omp start_region(id)
2 !$omp stop_region(id)
3 !$omp vsp
```

Figure 5.1: Proposed OpenMP directives for the chopper mechanism.

A region of code enclosed between the `start_region` and `stop_region` and identified by the parameter `id` is monitored during the execution. All referenced pages inside this region are forwarded to the next region at the

end of the region or at any available VSP, which are used as additional points to distribute pages.

We show two examples for using these directives: a typical approach where a parallel loop sends too much data and it uses the VSP to distribute the data; and another one where the chopper is used to group different parallel loops into a bigger region.

Example: Dividing a parallel loop into smaller sections

Figure 5.2 shows a typical parallel OpenMP loop which suffers network congestion. It is annotated as a region and a VSP is used at each 10 iterations.

```
1  !$omp parallel
2  !$omp start_region(id)
3  !$omp do
4      do i=1, N
5          A[i] = ...
6  !$      if ( i % 10) {
7  !$omp       vsp
8  !$      }
9      enddo
10 !$omp stop_region(id)
11 !$omp end parallel
```

Figure 5.2: Dividing a parallel loop into smaller sections.

Example: Grouping different parallel loops into a bigger region

There are situations where it is desirable to join different regions into a bigger one instead of dividing them, for example, in cases where the presend technique is not applicable because the computation time is too small. Figure 5.3 shows an example where two different loops are considered as a single context for the DSM by enclosing both inside the same region.

Runtime: Marking chopper regions

From the runtime point of view, an OpenMP application is a sequence of sequential and parallel regions, with some synchronization points in the middle.

```
1 !$omp parallel
2 !$omp start_region(id)
3 !$omp do
4     do i=1, N
5         A[i] = ...
6     enddo
7 !$omp do
8     do j=1, N
9         A[j] = ...
10    enddo
11 !$omp stop_region(id)
12 !$omp end parallel
```

Figure 5.3: Grouping two parallel loops into a bigger region.

This limits the runtime potential to modify the behavior of the application. Specially, if our goal is to distribute the coherence messages that are concentrating in a single synchronization point.

For this reason, we have modified the Nanos OpenMP runtime to offer functions to delimit a region of the application that suffers network congestion and another function to divide a region into smaller ones to distribute those messages:

int start_region (unsigned int id) This function marks the beginning of a new region of code. This region is identified with the `id` identifier. It also creates a link between this region of code and a new context inside the DSM layer.

int stop_region (unsigned int id) It marks the finishing point of the region of code identified with the `id` number.

int vsp() The Virtual Synchronization Point is the function that divides a bigger region into smaller sections (what we call *chops*), allowing the distribution of the network coherence messages.

Parallel loops are automatically annotated by our runtime at the beginning and at the end of the loop with `start_region` and `stop_region` respectively.

Nested regions are allowed but the inner ones will be silently ignored. This is useful, for example, to group different parallel loops together in a single region (see Section 5.4.2).

The implementation of these routines follows the next algorithm, where each thread will:

- After entering a region by the first time:
  - Record all accessed pages inside the region using the upcall mechanism.
  - If a VSP is found inside the region, mark all previously recorded pages as invalid to generate new faults on new accesses. This is essential to detect which VSP uses a page by the last time.
- After finishing a region by the first time:
  - Calculate the list of page addresses that are not faulted any more inside the region for each VSP.
  - Calculate a list of the page addresses faulted at the last section for the end of the region.
- After this first time, when a VSP or the end of the region is found, the list of page addresses is sent to the DSM which will pre-send or pre-invalidate each page to the next regions.

Runtime: Monitoring chops

The runtime monitor from previous chapter has been modified to monitor chops instead of parallel loops, and a parallel loop is redefined as a sequence of chops with a starting chop and an ending chop.

DSM: Support to allow cooperation with higher levels

From the DSM point of view, an OpenMP application is just a sequence of page faults without any knowledge about the context of these page faults.

We add some intelligence to the DSM to correlate a set of page faults to a specific region into the application and the relations between different regions. We have used the term *contexts* for these regions.

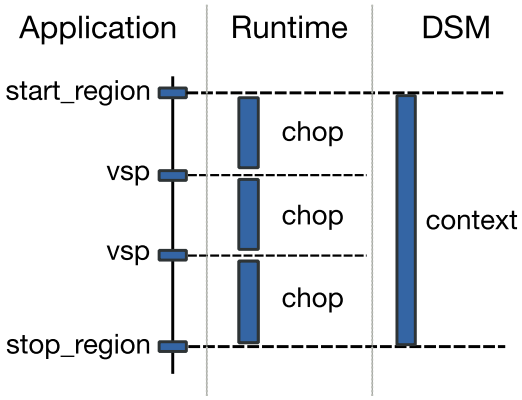

Figure 5.4: Chopper directives create *chops* in the OpenMP runtime, and these chops are grouped inside a DSM *context*.

Figure 5.4 summarizes the relation between the different concepts explained till this moment through the different layers: application, OpenMP runtime and NanosDSM. The `start_region` and `stop_region` directives define a DSM *context*, while the `vsp` directives create new *chops*.

The state of each page (page protections at each node) is recorded at each context and, therefore, DSM knows what actions (pre-send or pre-invalidate) need to be taken to change the page state in the current context to the next one.

Context

As stated before, a context is a region of code delimited by a starting and an ending points where page faults can occur. Each context is identified by an unique number. The DSM API offers two functions to define a context:

int msm_begin_ctxt (void * ctxt) This function marks the beginning of a context identified with the *ctxt* number. All pages faulted after this point will be associated with this context identifier.

int msm_end_ctxt (void) This function marks the end of the current context, identified with a previous call to *msm_begin_ctxt*. All page faults after this point will be annotated without any context, and so, they will not be considered in the context state calculation.

State of pages inside contexts

Each page in the DSM will record its state at each different context encountered. The state of a page at a region consists of a page protection: Read or Write; and the list of nodes that have a copy of the page. This state is built during runtime at the master page node with each page fault. On one hand, a write page fault sets the protection to write, resets the list of nodes containing the page and sets the node that faulted the page. On the other, a read page fault sets the protection to read and adds a new node containing a copy of the page.

In order to send data as early as possible, the states of a page with read protections between two writes can be accumulated. This way, after finishing a context where a page has been written, this page can be pre-sent at once to all nodes that will require the page in the following contexts. It is important to remark that all pages faulted outside a context will not be considered in this calculation for the context page states.

Context predictor

A simple 1-level predictor is used to detect the sequence of contexts, and to predict the next context after a given one. For each context, it stores which is the next context and a counter to ensure its validity.

The predictor is notified when a new context starts, inserting the context into the predictor if it was not present or updating its values if the saved prediction for the previous context matches (or not) the new one. The pseudocode is shown at Figure 5.5.

It is able to detect cycles in the sequence, but it will not detect when the cycle has finished, nor different context options due to branches.

Marking pages

In order to detect accesses to pages that are already owned by a node, DSM offers a couple of functions to the upper layers:

int msm_mark_page(void *page) This function marks a page so it will

```
1    function update_predictor( current )
2        if (current is new)
3            add current to predictor
4        if (previous.next = current) /* HIT */
5            previous.counter ++
6        else /* MISS */
7            if (previous.counter > 0)
8                previous.counter --
9            else
10               previous.next := current
```

Figure 5.5: Pseudo-code for the context predictor update function.

generate a page fault if accessed. It remembers the current protection of page *page* and invalidates its local copy. Any posterior access to the page will generate a page fault, restoring its previous protections and executing any registered upcall. After that fault the marking is disabled.

int msm_mark_page_range(void *start, unsigned long size) This function marks all the memory pages in the memory address interval between `start` and `start+size` rounded up.

This is used in conjunction with the upcall mechanism, where a function has been registered to be called whenever a page fault occurs inside a delimited memory region.

Automatic pre-send/pre-invalidate

A node can inform the DSM layer with a list of page addresses that may be forwarded to the next context using the pre-send/pre-invalidate mechanism.

Each node can build a list of page addresses. The construction of the list uses a couple a functions:

int msm_release(void *page) It adds the page address `page` to the current list. It can be called multiple times without sending any network message. A fixed size buffer is used to store the list, so in case the buffer fills, a `msm_release_all` call is issued, clearing the buffer and starting over.

int msm_release_all(void) This function sends the list of page addresses to the DSM and clears the list for posterior use. Each page address has a master node, so the list of page addresses is split into different lists, one list per master node, with all the page addresses corresponding to the same master node. Finally, each list is sent with a message to its master node.

When this list is received by a master node, for each page address it:

1. Predicts the next context that will use the page. This context will has the page state with the nodes that will need it and their protections.

2. Pre-sends the page to the nodes that will read it, or pre-invalidate the page and upgrade the page protections at the writer node if possible.

5.5 Evaluation

5.5.1 Methodology

In first place we evaluate the bandwidth behavior of the gigabit network in an environment similar to the DSM, trying to find its limits. Afterwards we use a synthetic and the BT benchmarks to evaluate the impact of the chopper mechanism in the performance results. And, finally, we evaluate the impact of the number of chops inside the same synthetic benchmark.

Bandwidth evaluation

To evaluate the network bandwidth we use a client-server application. The client sends a set of consecutive messages to all the nodes, and the server at each node waits for a message, acknowledges that message by resending it and waits for more messages.

The bandwidth is measured as the total number of bytes sent and retrieved by the client divided by the time it used:

$$bw = \frac{2 * MAX * m * nodes * sizeof(message)}{time}$$

```
1     do it=1, niter
2  !$omp parallel do default(shared) private(j)
3  !$omp& PRESEND
4           do j=1,na
5              v(j) = v(j) + 1
6              call heavy_calculation(3, delay)
7  !$          if (mod(j,  chop) .eq. 0) then
8  !$omp          vsp
9  !$          endif
10          enddo
11
12 !$omp parallel do default(shared) private(j,d)
13 !$omp& PRESEND
14          do j=na, 1, -1
15             d = d + v(j)
16             call heavy_calculation(d, delay)
17 !$          if (mod(j,  chop) .eq. 0) then
18 !$omp          vsp
19 !$          endif
20          enddo
21    enddo
```

Figure 5.6: Algorithm for the synthetic benchmark, annotated with the presend and the chopper.

, where `MAX` is the number of times that the test is repeated to minimize any outlayer effect; `m` is the number of messages send; `nodes` is the number of nodes involved in the execution; `sizeof(message)` is the size in bytes of the message sent in the request; and `time` is the measured time in seconds at the client side between the beginning of the test and the last message received.

Synthetic benchmark

The synthetic benchmark tries to exploit the fact that too many coherence requests produce congestion in the network affecting the final performance.

The synthetic benchmark algorithm iterates over two parallel loops. The first writes all positions of an array (`v`) and the second reads the same array in reverse order (see Figure 5.6). This is intended to avoid the same data placement and generate page faults at each parallel loop. Both loops have an added delay to simulate a longer computation time. And, finally, there is an instruction calling the VSP routine each `chop` iterations. By default, this `chop` value has been calculated so the application has a maximum of 400 chops (see Section 5.5.2 for a discussion on this number).

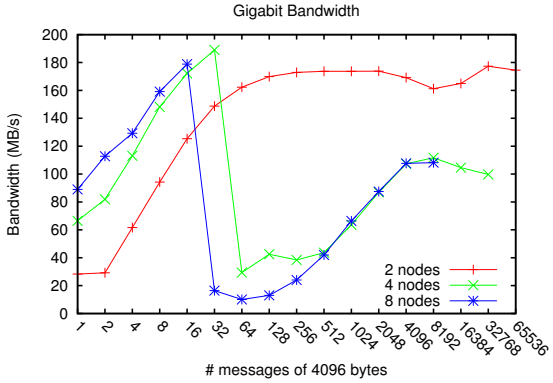

Figure 5.7: Bandwidth used for different number of messages of 4096 bytes.

The array `v` has 2000000 doubles (`na`), using about 4000 pages, big enough to be representative.

Three versions for this benchmark are presented: (a) *Original*, is the baseline version, without pre-send nor chopper; (b) *Presend*, where the pre-send is used at the end of each parallel loop; and (c) *Chopper*, which uses pre-send and chopper to avoid the network congestion, its source code is shown at Figure 5.6.

VSP overhead

To calculate the chopper overhead we have evaluated the effects of not using the chopper at all, which corresponds to the *Original* version; and different versions with 25, 50, 100, 200, 400 and 800 total chops. We have used these values to calculate the `chops` variable values in the synthetic benchmark shown in Figure 5.6 which controls the number of iterations to process by the thread before issuing a VSP.

5.5.2 Results

Bandwidth evaluation

Figure 5.7 shows the bandwidth used (in MegaBytes/seconds) by the benchmark when different messages of 4096 bytes are sent concurrently between 2, 4 and 8 nodes.

	Execution time (seconds)		
Nodes	Original	Presend	Chopper
Seq	53.80	53.80	53.80
1	54.09	53.88	54.28
2	50.30	37.89	29.06
4	25.46	18.71	14.61
6	17.15	11.85	9.79
8	12.99	8.69	7.39

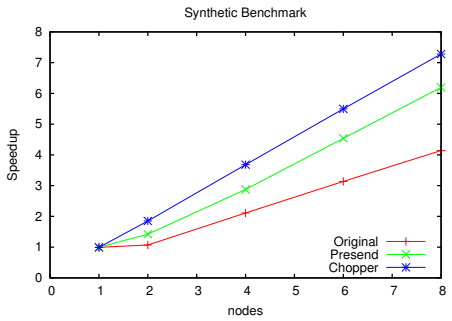

Figure 5.8: Synthetic benchmark performance results.

The figure shows, that the bandwidth between 2 nodes is not affected by the number of messages and it achieves a sustained bandwidth of 180MB/s close to the optimum bandwidth of the targeted network.

In contrast, when we use more than 2 nodes the bandwidth have a peak with 16 or 32 messages and immediately falls down to a minimum bandwidth. Afterwards, the bandwidth curve increases steadily until it arrives to another stable maximum at 100MB/s. This important decrease in bandwidth is due to the TCP congestion avoidance algorithm, and it may affect the performance.

Synthetic benchmark

The performance results for the different synthetic benchmark versions are presented in Figure 5.8. It shows that the original achieves a maximum speedup of 4 with 8 nodes. The pre-send concentrates all the coherence messages at the end of the parallel loops and it achieves a better speedup of 6. Finally the chopper version is able to achieve an speedup of 7, by the simple fact of distributing the coherence messages during the computation phase.

VSP overhead

Figure 5.9 shows the execution times for the synthetic benchmark when using 25, 50, 100, 200, 400 and 800 total chops. The original and presend version

	Execution time (seconds)						
Nodes	Original	25	50	100	200	400	800
1	54.09	54.31	54.25	54.07	53.98	54.09	54.09
2	50.3	49.09	48.05	45.87	40.92	**29.22**	29.78
4	25.46	24.93	24.36	23.16	20.65	**14.64**	14.93
8	12.99	12.77	12.50	11.90	10.57	**7.37**	7.51

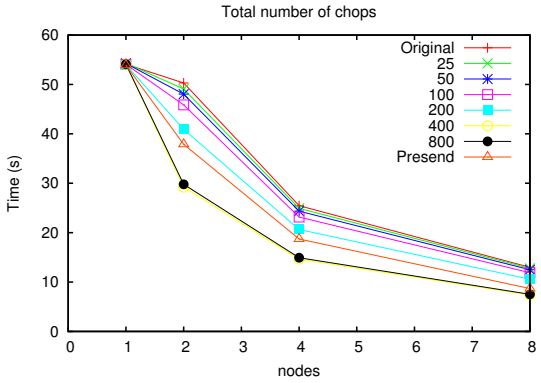

Figure 5.9: Execution time (in seconds) for the Synthetic benchmark with different number of total chops. Best values are highlighted.

	Pages synchronized at each VSP					
Pages	25	50	100	200	400	800
4000	160	80	40	20	10	5

Table 5.1: Number of pages synchronized at each VSP for the Synthetic benchmark with different number of total chops.

	Execution time (seconds)		
Nodes	Original	Presend	Chopper
Seq	46.88	46.88	46.88
1	48.53	47.00	47.20
2	50.44	32.03	28.32
4	36.39	25.06	19.76
8	25.66	17.95	16.33

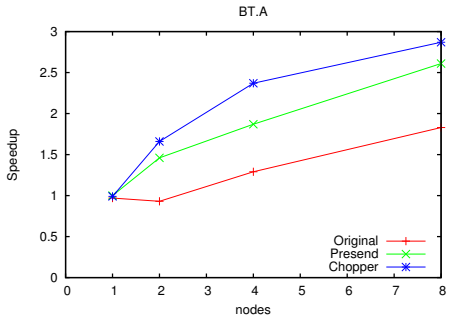

Figure 5.10: BT Class A performance results.

also appear for completeness. Each version is evaluated with 1, 2, 4 and 8 nodes.

This figure shows that the best case corresponds to using 400 chops, which is the value used in the Chopper version presented before.

The figure also shows that the number of chops must be chosen wisely, because it affects the performance. It clearly shows two boundaries: (i) the best execution time achievable with the chopper and (ii) the worst execution time corresponding to the original version. On one hand, adding too few VSPs decrease the execution time marginally and obtains worse behaviour than the presend version. But, on the other hand, adding too much VSPs gets a performance similar to the best execution or slightly worse. Therefore there must be a tradeoff between the execution time and the number of chops, and it will depend on each application.

Due to the fixed number of chops per application, the number of VSPs per thread decrease with the number of threads. This is a desirable situation because the computation time available per thread also diminishes and there are less time to overlap the communication.

Finally, Table 5.1 shows the number of pages synchronized at each VSP for this synthetic benchmark when using different number of VSPs. As the table shows, in this case, the number of pages are directly related to the number of chops.

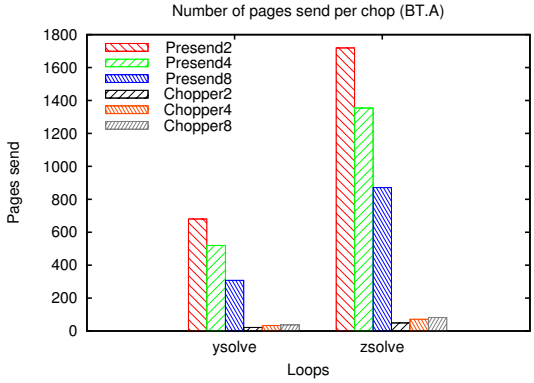

Figure 5.11: Average number of pages send in BT.A benchmark by each thread at each synchronization point in parallel loops `ysolve` and `zsolve` when executing the presend and chopper versions with 2, 4 and 8 nodes.

5.5.3 BT benchmark

In order to execute the 3.3 version of this benchmark in our environment we have modified the code slightly, because our runtime lacks an implementation for the *threadprivate* directive. The solution has been to modify all the variables declared as threadprivate to be globally shared, and privatizing them each time that they are used.

To test our mechanism we used the class A of this benchmark, which defines the size N of the working matrices of NxNxN elements as 64.

We compare the results for the original, presend and chopper versions. The chopper version is the same as the presend, but issuing a chop at each iteration of two of the main loops (`ysolve` and `zsolve`) to reduce the number of pages send at the same time. A more detailed study explaining why these loops needs the chopper will appear in chapter 6.

Figure 5.11 shows the average number of pages send by each thread at each synchronization point in both parallel loops when executing the Presend and Chopper versions with 2, 4 and 8 nodes. As the figure shows, the Presend version needs to send all pages at a single point, and therefore there are a lot of pages send. In contrast, the Chopper version can use the extra synchronization points to distribute these pages and, therefore, reduce the number of pages send.

A serial thread takes 46.88 seconds to execute 20 iterations of the main loop of this class A.

The speedup results for this class are shown in Figure 5.10. Original version has poor scalability. It has some slowdown with 2 nodes and a speedup of 1.9 is obtained when using 8 nodes. Presend technique achieves better results, speedup of 1.5 for 2 nodes and 2.3 for 8 is achieved. Even better speedups for all nodes are obtained when the chopper is used, 1.7 and 2.7 with 2 and 8 nodes respectively, meaning a performance improvement of about 16%. But this speedup trend seems to change slightly compared with the other versions, in fact a superior speedup could be expected with 8 nodes. As we will see in chapter 6, this change is due to the lack of enough computation time to get profit from the chopper.

5.6 Conclusions

In this chapter we have shown that the OpenMP applications, with a high volume of memory accesses, executed on top of a page based DSM produces network congestion.

The congestion appears at OpenMP synchronization points, where the DSM exchanges messages: (1) to maintain the global shared-memory in a consistent state and/or (2) to pre-send data to improve performance.

This congestion may limit or degrade the final application performance due to the limited bandwidth of the network, but it can be avoided using a better temporal distribution of these messages.

Therefore we propose a technique to distribute the data producing the congestion along the computation phase of the application using additional points before those congestion points: the Virtual Synchronization Points.

The technique is based on adding these new virtual synchronization points in the application code and the runtime using them to dynamically decide what data and at which moment can be safely exchanged to minimize network congestion.

A better overlap of computation with communication is achieved, because the network messages are distributed along the computation phase of the

application.

A synthetic benchmark and the BT benchmark are used to demonstrate the potential of the technique on top of our page based DSM implementing sequential consistency. The effects on more applications will be shown in next Chapter 6.

Results show that the network congestion situation is reduced and the application performance is increased by 19% on average over the Presend version. As expected, results also show that the proposed technique is limited by the computation time available to distribute all network data and that the number of VSP to use at each thread should be chosen wisely to achieve the best results. For all these reasons some compiler solution would be desirable.

Part of these results and contributions have been published in [CCM+10, CBMC09, CCM+09].

Chapter 6

Performance Evaluation

The devil is in the details

Classic proverb

Abstract

This chapter presents a detailed analysis of the NAS OpenMP benchmarks performance results when executed on top of our distributed environment. The analysis includes two versions: the original one, to detect the main problems affecting the applications performance; and a final version modified with the techniques explained in previous chapters, to obtain the best performance results.

This study concludes with a list of considerations to take into account when using these techniques in order to maximise the benefits.

6.1 Introduction

In previous chapters we have described three different techniques to overcome some of the problems of executing OpenMP applications on top of a distributed platform using a distributed shared memory. The boundaries alignment, to eliminate the false sharing in linear arrays; the data forwarding with presend and preinvalidation, to reduce the page fault handling latency, overlapping the communication with the computation; and distribution of network coherence messages to avoid the bottlenecks from sending a huge number of messages at the same time like the presend produces.

We have evaluated the benefits of these techniques individually, but in this chapter we analyze their combined effects to get the best performance results for the NAS OpenMP benchmarks. The NAS benchmarks, which are typically used to evaluate the performance of shared memory platforms, consist of seven kernels: EP, CG, BT, SP, LU, MG and FT.

This chapter is organized in five sections. Section 6.2 describes the methodology used to do this evaluation. Section 6.3 describes the testbed where the applications have been executed, and Section 6.4 shows the results for each of the NAS benchmarks. Finally section 6.5 presents the conclusions of this results.

6.2 Methodology

The methodology to evaluate the use of our techniques in an application is as follows. In first place, we show a terse description of the benchmark with an enumeration of its parallel loops. In second place, a preliminary study of these parallel loops is done to detect performance problems and potential uses of our techniques. And, finally, we present the performance results according to the techniques decided before applied and a detailed study of these results.

6.2.1 Benchmark description

This section explains the main structure of each benchmark and describes its parallel loops, because they will guide the following study. A preliminary description of the benchmarks has been already presented in previous chapters, section 2.6.1.

6.2.2 Detailed parallel loops study

The behavior of the benchmarks is directly related to the behavior of its inner parallel loops, thus a detailed study of all parallel loops in the application is shown.

The study analyzes two versions for each benchmark: the *original*, which studies the results for the original source code without any modification; and the *final*, which modifies the original code with a combination of our techniques to obtain the best performance results: The analysis of the parallel loops includes a categorization of the execution time and the detection of the access pattern for the shared variables in the loop.

Benchmark versions

1. **Original version details** limits the study to the original version to detect any problem that degrades the performance of the application when executed in our environment. The benchmark algorithm and the main data structures used by this algorithm are shown, analyzing their access patterns, which will guide the decision of the best techniques to use.

2. **Final version details** presents the set of techniques to apply to each parallel loop to obtain the best final performance result. Accordingly, the effects of this decision are also explained.

Shared variable access pattern

The shared variable access pattern is an study of how the different variables (data structures) used by the benchmark are accessed through the differ-

ent parallel loops. This is necessary to detect data dependences and, more important, detect the producer-consumer relations needed for the presend technique.

The access type of a shared variable inside each parallel loop, can be:

1. Read by some thread (R).
2. Written by some thread (W).
3. Read and written by some thread (RW).
4. Not used by any thread (-).

Categorized parallel loops execution time

The execution time of each parallel loop, measured as the time between the beginning of the parallel loop and the beginning of the next one, quantifies the cost of each loop inside the whole application.

This time has been categorized in three different aspects: **(a)** *compute time*, time doing real computation work; **(b)** *page fault time*, time solving remote page accesses; and **(c)** *idle time*, all the remaining time without computation nor page faults. This *idle time* includes the overhead of the library, barriers and real sequential time.

For each benchmark, we have used a small region that includes 4 or 5 iterations of the benchmark to gather the numbers presented in the results. This results are presented with two different graphs showing the *total time per loop* and *average time per loop*.

On one hand, the **total time per loop graph** shows the time used by all threads to execute each loop inside the considered region. For each loop, it sums up all times (compute, page faults and idle) from all threads instances of the loop.

The time is shown in the Y axis and the different loops of the benchmark are shown in the X axis, with a column for each execution. This statistic shows the global behavior of the application and the most time consuming loops.

On the other, the **average time per loop graph** shows the average time used by each thread to execute a loop in the benchmark. It uses the same axis as the previous graph.

This graph is used to make comparisons between the baseline and the final versions. Due to its finer granularity it is chosen to show the effects of the different techniques on each loop.

6.2.3 Performance results

The performance results section shows a summary of the performance results obtained by the original application without using our techniques, and the results when they are used. The performance of the application without any of our techniques is used as a baseline.

Each version of the benchmark is executed a minimum of 5 times, and the final execution time corresponds to their arithmetical mean. All execution times are measured in seconds. The standard deviation is so small that it is not shown.

Each version has been executed with 1, 2, 4 and 8 nodes and a serial version (Seq) without any call to the OpenMP runtime nor the DSM API. The speedup is calculated using this serial time.

6.3 Testbed

This chapter presents the applications executed in Marenostrum (see Section 2.7.4 for extra details) with a maximum of 8 nodes, using one working thread per node. A single working thread has been used because the study centers on ways of reducing the number of remote page accesses, adding more threads potentially reduces the computation time but the number of remote page accesses per node remains constant.

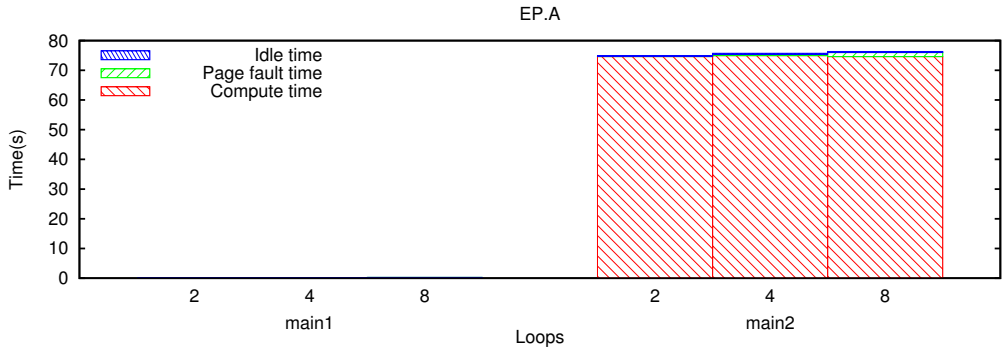

Figure 6.1: Total time per loop for EP class A.

6.4 NAS Benchmarks

6.4.1 EP

The embarrassingly parallel benchmark is a very simple benchmark that makes some data placement at the beginning, does all the computation locally at each node and, finally, the master node recovers the remotely calculated results.

Original version details

The benchmark has two parallel loops: `main1` which does the data placement and `main2` which does the computation locally.

Figure 6.1 shows the total time per loop used by each loop. It shows that the second loop doing the computation is the most time consuming and, as expected, the time solving page faults is extremely low. This is expected because each thread only needs to access the first values of the shared variables, and then all computation can be done without any other remote accesses. These initial page faults, due to the working threads accessing the master node stack for these initial values, are a feature of the current implementation of the OpenMP runtime and so they are actually unavoidable.

The `main1` loop uses very little time and therefore it does not show up in the Figure.

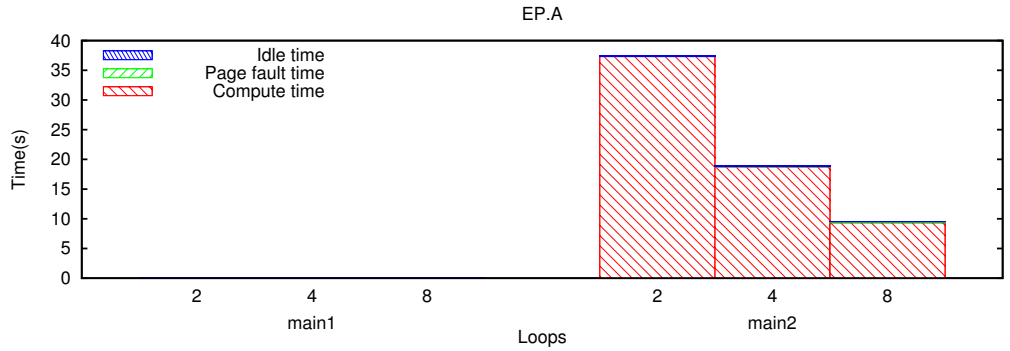

Figure 6.2: Average time per loop EP class A.

	Execution time (seconds)
Nodes	Original
Seq	73.38
1	74.43
2	37.36
4	18.80
8	9.54

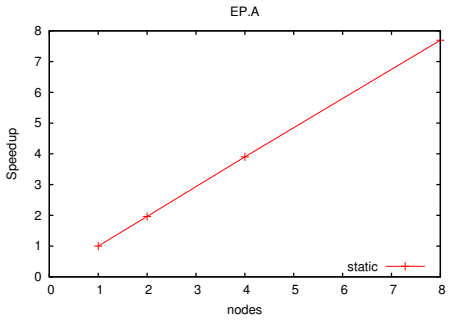

Figure 6.3: EP Class A performance results.

This particular benchmark does not need any additional technique, because, as the Figure 6.2 shows, the scalability of this `main2` loop is linear and there are no page faults. In fact, the use of the presend technique will add unnecessary overhead, because this benchmark is not iterative, and so the cost of registering the memory will just add overhead without obtaining any performance benefit.

Just for completeness, the execution times and speedup curve for this benchmark is shown in Figure 6.3.

6.4.2 CG

The algorithm of this benchmark, shown in Figure 6.4, contains a main loop with a call to the `conj_grad` subroutine and a couple of parallel loops (`main1` and `main2`).

```
1 do it = 1, niter
2   call conj_grad()
3   !$ omp parallel do "main1"
4   !$ omp parallel do "main2"
5 enddo
```

```
1 subroutine conj_grad()
2   !$ omp parallel do "cg1"
3   !$ omp parallel do "cg2"
4   do cgit = 1, cgitmax
5     !$ omp parallel do "cg3"
6     !$ omp parallel do "cg4"
7     !$ omp parallel do "cg5"
8     !$ omp parallel do "cg6"
9   enddo
10  !$ omp parallel do "cg7"
11  !$ omp parallel do "cg8"
12 end subroutine
```

Figure 6.4: Structure of the CG algorithm.

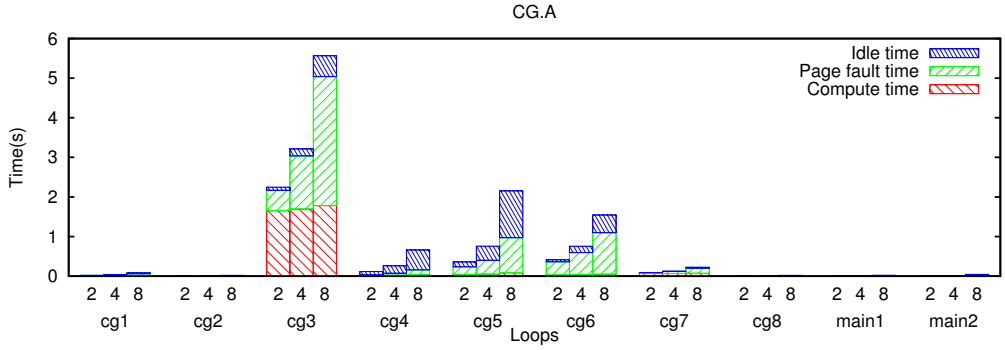

Figure 6.5: Total time per loop for original version of CG class A.

The subroutine has an inner loop that iterates `cgitmax` times over the main four parallel loops (`cg3`, `cg4`, `cg5` and `cg6`), a couple of parallel loops (`cg1` and `cg2`) to initialize the working sets at the beginning and two more (`cg7` and `cg8`) at the end.

Original version details

As Figure 6.5 shows, the most time consuming loops are the ones present in the conj_grad subroutine (`cg3`, `cg4`, `cg5`, `cg6` and `cg7`). So we will center the study on these loops only.

The most time consuming loop (`cg3`) writes an array (`q`) using an indirect access on top of array (`p`) causing a high number of page faults. The following loop (`cg4`) just reads the previously written data, so there are few faults because each node has the pages cached locally. Afterwards, in the third and

Loop	z	q	p	r	Align	Presend	Chopper
cg1	-	W	R	-	-	-	-
cg2	-	W	R	-	-	-	-
cg3	-	W	R	-	√	√	√
cg4	-	R	R	-	√	√	-
cg5	RW	R	R	RW	√	√	-
cg6	-	-	RW	R	√	√	-
cg7	R	-	-	W	-	-	-
cg8	-	-	-	R	-	-	-
main1	R	-	-	-	-	-	-
main2	-	-	-	-	-	-	-

Table 6.1: CG benchmark access pattern and summary of techniques used at each parallel loop.

fourth loops (`cg5` and `cg6`), there are some variables written whose cached copies must be invalidated, and so there are write faults.

Finally, the last loop (`cg7`) reads one of the arrays(`z`) using the indirect access, and therefore some page faults appear. Figure 6.1 summarizes these accesses. It shows all loops in the benchmark, and for each loop the access type to each of the shared variables used in the benchmark.

Final version details

To improve the performance of this benchmark we decide to use the align mechanism in the four main loops, to avoid false sharing; the presend mechanism in the same loops, to avoid the page faults; and the chopper just in the most time consuming loop, to give more time for sending all pages. A summary is shown in Table 6.1, where all parallel loops are shown and there is a column for each of our contributions: Align, Presend and Chopper. A symbol '√' is used to depict that the selected loop uses that contribution.

Figure 6.6 shows the final results for this version and the faults are reduced considerably in almost all loops. The indirect access pattern found in the `cg3` loop may seem difficult to solve, but the truth is that the accesses are repeated along the execution, so the presend is able to detect them. This way, the `cg6` loop is able to presend the right pages on time. The second

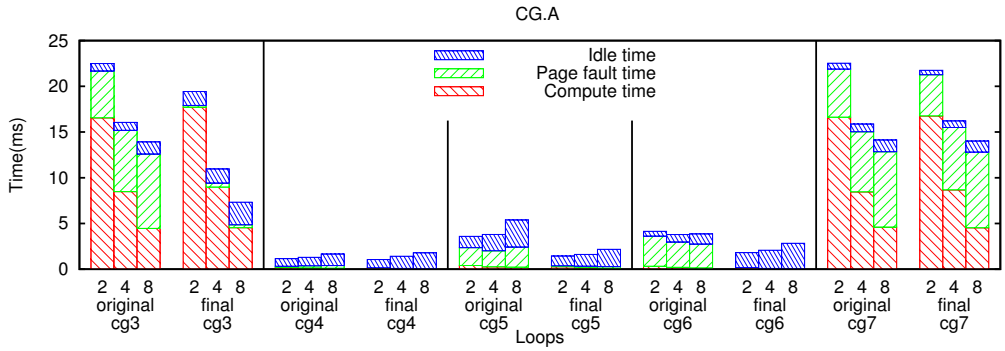

Figure 6.6: Average time per loop for original and final versions of CG class A.

loop is not affected because it has few faults. But the third and fourth loops, are really improved because the preinvalidate is able to invalidate the copies before any write access is done.

Even though the last loop `cg7` takes a non-negligible time, it is not improved. This is explained because the current implementation of the predictor is too simple and it is not able to predict the last iteration of the previous parallel loop (`cg6`) and therefore it does not presend/preinvalidate the `cg7` page faults. Actually, this is not an issue, because the impact on the total execution time is quite small. But it can become a problem with an increasing number of nodes.

It is interesting to note that the use of our techniques, even it reduces the time solving page faults due to a reduction of the number of page faults, increases slightly the idle time due to some network congestion.

The use of the presend technique changes the data movement and concentrates it at single points which we have explained that is a problem, and it is the same behavior that appears in a DSM using a relaxed consistency model.

The presend technique predicts the pages that are needed in the next parallel loop and thus, at the end of a parallel loop, it sends all predicted pages in parallel to the consumer node. These pages produce a huge number of network messages at the same time interfering in the normal behavior of the DSM.

	Execution time (seconds)	
Nodes	Original	Final
Seq	12.82	12.82
1	14.04	14.95
2	11.81	9.26
4	9.70	6.36
8	9.27	5.81

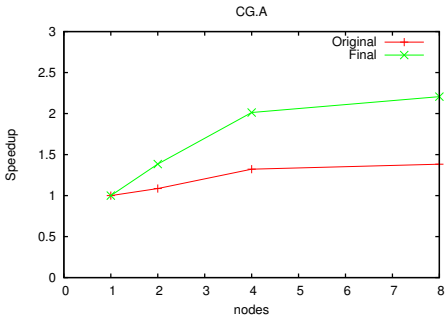

Figure 6.7: CG Class A performance results.

In contrast, a DSM with sequential consistency distributes the remote memory accesses by a node evenly during the computation phase, because the accesses are sequentially ordered. For example, during a parallel loop that traverses a linear array writing each position, a node writes a non-local page, sends a network message and waits for the answer; when the page arrives with the right protections, the node updates their values and access the following non-local page repeating the process.

The chopper technique tries to avoid this problem but, in this case, it is not able to eliminate the problem due to the random access patterns and the small computation time available to overlap this communication.

Figure 6.7 shows the resulting performance results for the original and final versions. The small speedup obtained by this benchmark is basically due to its small computation, insufficient to hide the communication costs.

This is confirmed when a bigger class is used. Figure 6.8 shows the performance results for the class B and it is clear that with a small number of nodes the computation time is enough to overcome the cost of the page faults, but as the number of nodes grows, this computation gets smaller, and the overhead of the page faults affect the performance. In contrast, the use of our techniques allows to overlap this cost and the application maintains the speedup until 8 nodes.

	Execution time (seconds)	
Nodes	Original	Final
Seq	1399.71	1399.71
1	1533.74	1399.71
2	524.96	577.85
4	375.04	317.69
8	329.42	204.05

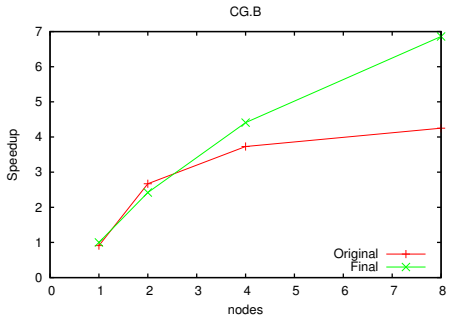

Figure 6.8: CG Class B performance results.

6.4.3 BT

As explained in Section 2.6.1, the BT benchmark repeats the execution of five functions (`rhs`, `xsolve`, `ysolve`, `zsolve` and `add`) a fixed number of iterations.

The source code implementing these five functions has 15 parallel loops. We consider the 11 parallel loops contained in the `rhs` function as a group because their computation is small and the results are best shown.

Original version details

The BT algorithm uses the same dimension to parallelize all loops, maximizing the spatial data locality, but this parallelization changes in the `zsolve` and in one of the parallel loops of `rhs`. This behavior explains why three parallel loops (`rhs`, `zsolve` and `add`) uses a lot of time solving page faults, while the rest (`xsolve` and `ysolve`) do not fail any page. Figure 6.9 shows the total time (in seconds) used by each loop when executing the original version with 2, 4 and 8 nodes.

An example of the effects of a change in the parallelization index is shown in Figure 6.10. It shows the data layout of a matrix when the `xsolve`, `ysolve` and `zsolve` loops are executed by three threads. The first two loops use the outer dimension of the matrix to parallelize the algorithm (see Figure 6.11), while the `zsolve` loop uses an inner dimension (see Figure 6.12).

Due to the memory organization, each thread accesses consecutive pages when the loop is parallelized using the third dimension, because the first two

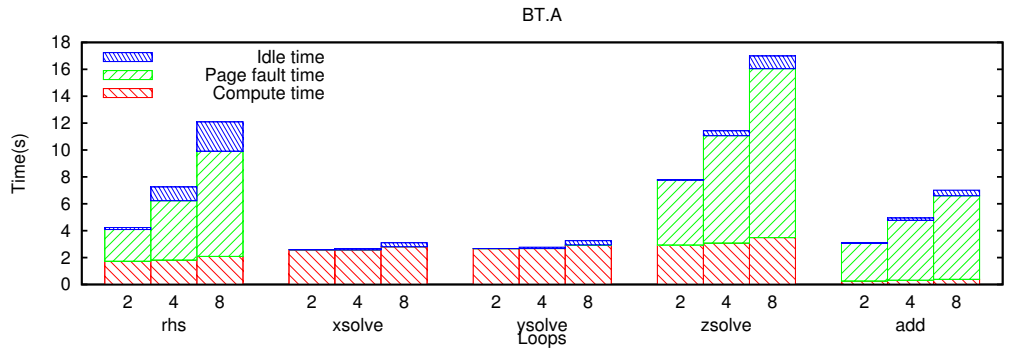

Figure 6.9: Total time per loop for original version of BT class A.

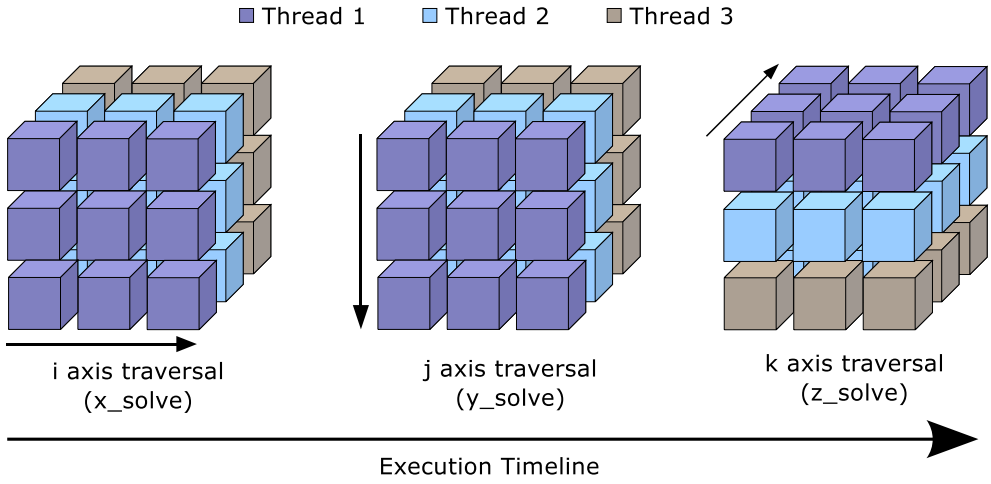

Figure 6.10: Parallelization of BT benchmark.

```
1 !$omp for
2     for k = 1, P
3       for j = 1, N
4         for i = 1, M
5           m[i][j][k] = ...
```

Figure 6.11: Matrix parallelized with the outer dimension.

```
1 !$omp for
2       for j = 1, N
3     for k = 1, P
4       for i = 1, M
5         m[i][j][k] = ...
```

Figure 6.12: Matrix parallelized with an inner dimension.

Loop	u	rho_i	us	vs	ws	square	qs	rhs	Align	Prsnd	Chppr
rhs	R	RW	RW	RW	RW	RW	RW	RW	√	√	-
xsolve	R	R	-	-	-	R	R	RW	√	√	-
ysolve	R	R	-	-	-	R	R	RW	√	√	√
zsolve	R	-	-	-	-	R	R	RW	√	√	√
add	RW	-	-	-	-	-	-	R	√	√	-

Table 6.2: BT benchmark access pattern and summary of techniques used at each parallel loop.

dimensions are kept together.

The change in the parallelization produces a reordering of the memory accesses and thus the locality is lost and it produces a lot of page faults. Even worse, at the next loop, the spatial locality is lost again because the first parallelization is used again, producing more page faults.

Finally, Table 6.2 shows the shared variables used in the benchmark and the access type at each of the parallel loops. The `add` and `rhs` parallel loops write all shared variables that will be accessed in the next loops while `xsolve, ysolve` and `zsolve` read their content updating the final `rhs` variable

Final version details

In this benchmark we use the align technique in all loops because, even it can not be applied to matrices to solve false sharing, it is useful to reuse the same schedule for all loops and avoid slight glitches at the loops limits, when a loop executes one iteration more or one iteration less.

The Presend technique is applied to all parallel loops, while the chopper is used in the `ysolve` and `zsolve` loops only. The chopper is necessary to avoid the congestion that the presend produces at the end of these loops due to the high amount of pages to distribute.

A summary of the application of these techniques at each loop is presented in Table 6.2 and their results are shown in Figure 6.13. It shows the average time (in milliseconds) used by each thread at each parallel loop for the two versions: the original and the final.

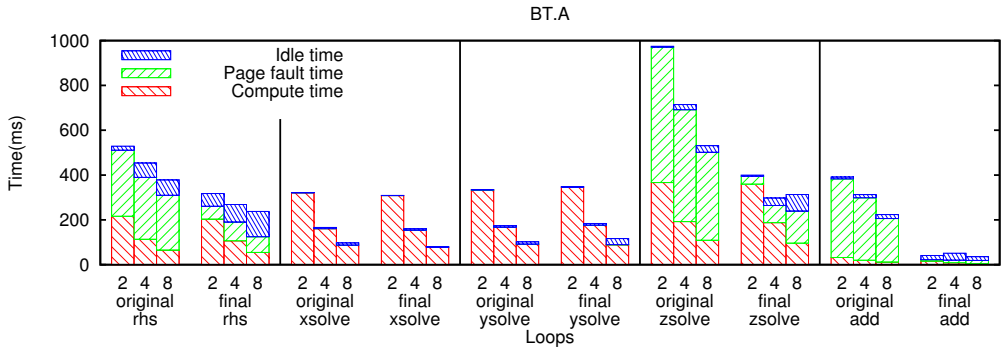

Figure 6.13: Average time per loop for original and final versions of BT class A.

As the figure shows, the time solving page faults have been reduced considerably. These pages are forwarded from the previous parallel loops and thus a local copy exists when the node makes the access. The use of the presend and the chopper alleviates the problem of changes in parallelization index by forwarding data that will be needed before its use.

Even though our techniques reduce the time solving page faults, the computation time per thread decreases when the number of nodes increases, and consequently the time available to hide the communication also decreases. This is specially visible in the final version of `zsolve` when executed with 8 nodes, where the total execution time is smaller than the original, but the page fault and idle times are higher than with fewer nodes in the same version. This explains why the results goes worse with a higher number of nodes for this class size.

Figure 6.14 shows the execution times and the speedups obtained by the original version of the BT class A benchmark and the obtained final version.

The original version of this benchmark has some speedup starting at 4 nodes arriving at a maximum speedup of 1.8 with 8 nodes. The final version improves the performance of the benchmark by a 32% in average, with a maximum performance of 2.6.

In this case, the use of a bigger class does not help because the compu-

	Execution time (seconds)	
Nodes	Original	Final
Seq	46.88	46.88
1	48.53	47.20
2	50.44	28.32
4	36.39	19.76
8	25.66	16.33

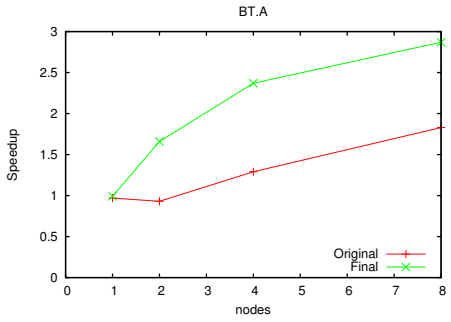

Figure 6.14: BT Class A performance results.

	Execution time (seconds)	
Nodes	Original	Final
Seq	195.00	195.00
1	197.22	202.86
2	205.56	122.50
4	140.08	96.55
8	91.93	55.03

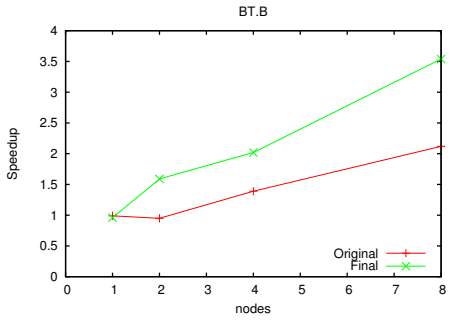

Figure 6.15: BT Class B performance results.

tation time increases, but the number of page faults also increases meaning that the behaviour does not depend on the class size and therefore we obtain similar results as Figure 6.15 shows.

6.4.4 SP

The SP is another benchmark solving a system of partial finite differential equations, similar to the BT, but with a different algorithm. As explained in Section 2.6.1 the SP has nine functions: `rhs`, `txinvr`, `xsolve`, `nivnr`, `ysolve`, `pinvr`, `zsolve`, `tzetar` and `add`. Each function has one parallel loop except `rhs` that has six.

Original version details

Figure 6.16 shows the total time per loop, and it clearly shows that a huge amount of time is invested in the page fault handling. The most time con-

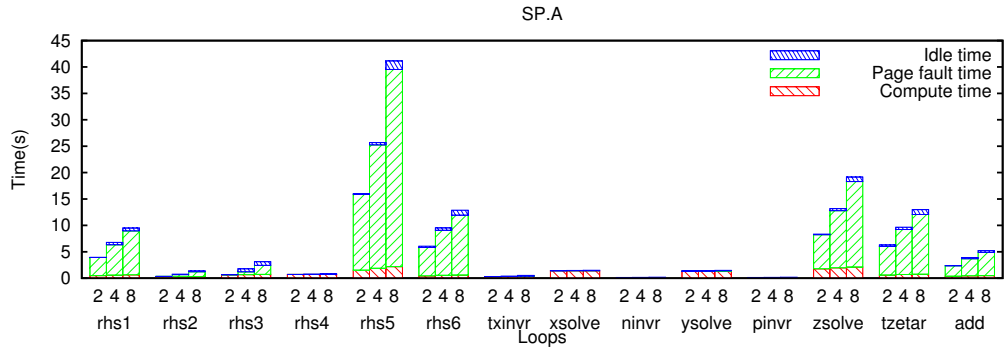

Figure 6.16: Total time per loop for different versions of SP class A.

suming loops are: `rhs5, rhs6, zsolve, tzetar, add` and `rhs1`.

The huge amount of page faults produced are due to changes in the parallelization scheme, in this case it happens in the `rhs5` and `zsolve` loops. It is exactly the same case as explained in Section 6.4.3 where these changes produce the loss of locality.

An important difference of this benchmark compared with the previous BT benchmark is the ratio between the time dedicated to computation and the time solving page faults, which is huge.

As in the BT benchmark, the `rhs` and `add` loops writes all the shared variables that will be needed in the next loops, and the remaining loops just reads these variables and updates the resulting `rhs` variable (see Table 6.3).

Final version details

For the final version, the presend technique is applied to the most notorious loops. In this case just the loops doing the changes in the parallelization index, because the loops writing the main structures have a very small computation time, and it is not worth to use the chopper. As usual, the chopper is used in the loops prior to the change in the parallelization index (`rhs4` and `ysolve`), to prepare the data placement; and the ones with the change (`rhs5` and `zsolve`), to restore the page placement in the next loop. The `pinvr` and `tzetar` also uses the chopper, because they form part of the `ysolve` and `zsolve` parallel loops. Table 6.3 shows a summary of the techniques

Loop	u	rho_i	us	vs	ws	square	qs	speed	rhs	Align	Prsnd	Chppr
rhs1	R	W	W	W	W	W	W	W	-	-	√	-
rhs2	-	-	-	-	-	-	-	-	W	-	√	-
rhs3	R	R	R	R	R	R	R	-	RW	-	-	-
rhs4	R	R	R	R	R	R	R	-	RW	-	√	√
rhs5	R	R	R	R	R	R	R	-	RW	-	√	√
rhs6	-	-	-	-	-	-	-	-	RW	-	-	-
txinvr	-	R	R	R	R	-	R	R	RW	-	-	-
xsolve	-	R	R	-	-	-	-	R	RW	-	-	-
ninvr	-	-	-	-	-	-	-	-	RW	-	-	-
ysolve	-	R	-	R	-	-	-	R	RW	-	√	√
pinvr	-	-	-	-	-	-	-	-	RW	-	√	√
zsolve	-	R	-	-	R	-	-	R	RW	-	√	√
tzetar	R	-	R	R	R	-	R	R	RW	-	√	√
add	RW	-	-	-	-	-	-	-	R	-	√	-

Table 6.3: SP benchmark access pattern for the different structures and summary of techniques used at each parallel loop.

applied at each parallel loop and Figure 6.17 shows the comparison of the two versions for the different loops.

It is interesting to note that even the time solving page faults have been reduced, in the cases where the parallelization index changes (`rhs5` and `zsolve`), the time solving page faults increases with the number of nodes. The problem is that the presend is not able to deliver all pages on time, due to the quantity of pages to send. Even though the chopper has been used to send these pages as early as possible, the computation time available is so small that it is not enough to overlap all the communication.

Another interesting effect is the increase of the 'idle time' in all `rhs`, the `pinvr` and `zsolve` loops. This is an after-effect of the huge amount of presend messages that the DSM infoservers should treat at the same time, interfering on the normal behavior of the DSM, meaning that the messages are queued and therefore the treatment of other messages, like the barrier messages, are delayed.

The performance results of this benchmark, presented at Figure 6.18, shows that even our techniques improves the global performance, the effect

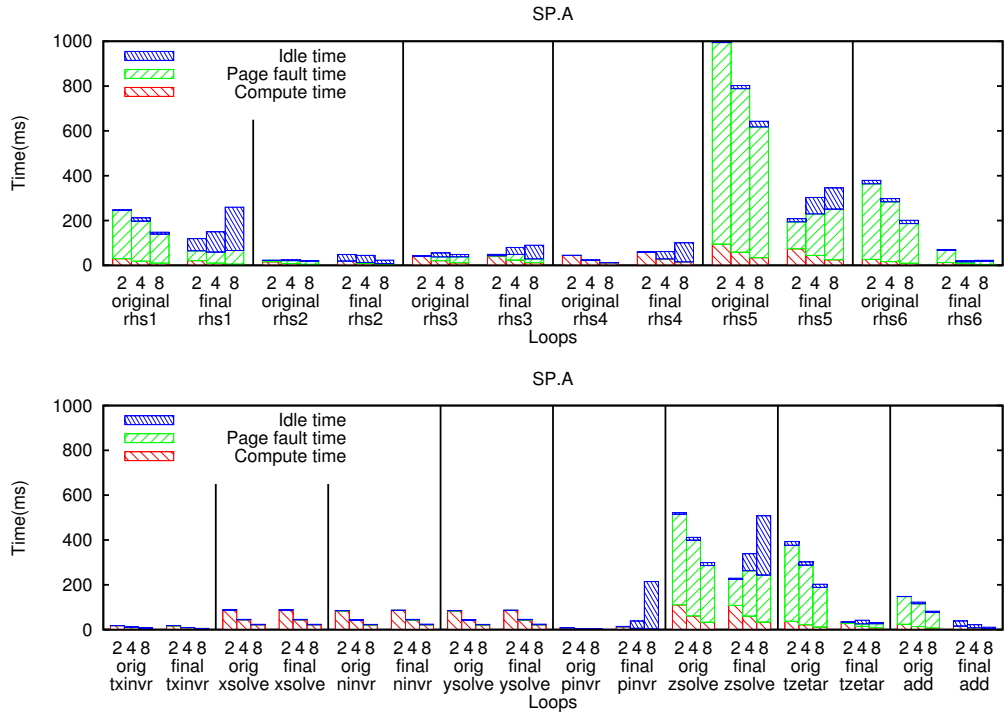

Figure 6.17: Average time per loop for original and final versions of SP class A.

	Execution time (seconds)	
Nodes	Original	Final
Seq	401.88	401.88
1	403.44	416.67
2	1179.91	457.33
4	930.35	526.92
8	683.6	637.42

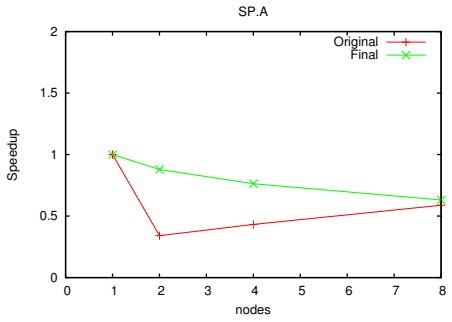

Figure 6.18: SP Class A performance results.

of sending too much data at the same time affects negatively the final result. Again the problem is due to an available computation time that is not enough to overlap the communication.

6.4.5 LU

The LU is yet another kernel to solve a system of partial finite differential equations.

The structure of this benchmark is slightly different to the previous benchmarks presented till this moment. The most interesting part is the fine granularity of the parallel loops. Instead of doing big parallel loops, there is a parallel region that executes a loop with calls to OpenMP orphaned loops. Figure 6.19 shows this structure and a sample from the `jacu` subroutine. There are a total of 12 parallel loops: `ssor1`, `jacld`, `blts1`, `blts2`, `jacu`, `buts1`, `buts2`, `ssor2`, `rhs1`, `rhs2`, `rhs3` and `rhs4`.

Even it is not shown, there is a manual synchronization through a shared variable between the two parallel loops from `buts` and `blts`, to force a pipeline between the working threads.

Original version details

Figure 6.20 presents two graphs with the execution times per loop with some interesting effects.

On one hand, the total time per loop shows that the computation time per loop is small, and that all loops have a high amount of page faults.

```
1  do step=1, niter
2      !$omp parallel do ''ssor1''
3      !$omp parallel
4      do k=2, nz - 1
5          call jacld
6          call blts
7      enddo
8      !$omp end parallel
9      !$omp parallel
10     do k=nz - 1, 2, -1
11         call jacu
12         call buts
13     enddo
14     !$omp end parallel
15     !$omp parallel do ''ssor2''
16     call rhs
17 enddo
```

```
1 subroutine jacu(k)
2 !$omp do
3     do j = jend, jst, -1
4         do i = iend, ist, -1
5             u[i][j][k] = ...
6             ...
7         enddo
8     enddo
```

Figure 6.19: Structure of the LU algorithm and the `jacu` subroutine.

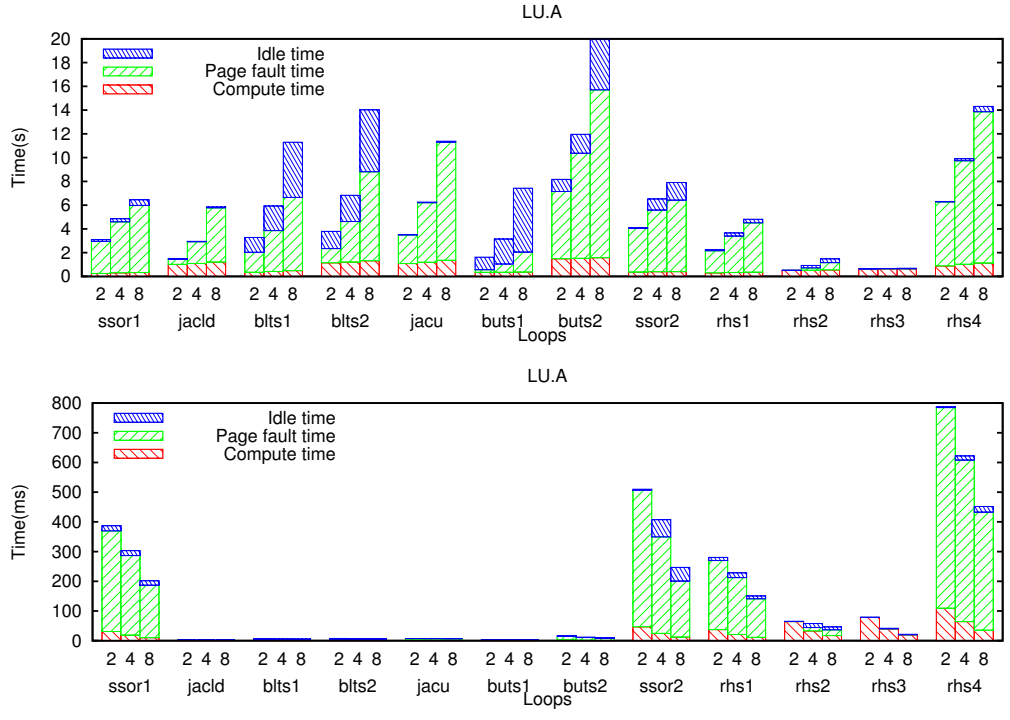

Figure 6.20: Total time per loop (above) and average time per loop (below) for LU class A.

Loop	u	rho_i	a,b,c,d	au,bu,cu,du	rsd	Align	Prsnd	Chppr
ssor1	-	-	-	-	RW	√	√	-
jacld	R	R	W	-	-	-	-	√
blts1	-	-	Ra	-	R	-	-	√
blts2	-	-	Rbcd	-	RW	-	-	√
jacu	R	R	-	W	-	-	-	√
buts1	-	-	-	Rc	R	-	-	√
buts2	-	R	-	Rabd	RW	-	-	√
ssor2	RW	-	R	-	R	√	√	-
rhs1	R	W	-	-	W	√	√	-
rhs2	R	R	-	-	RW	√	√	-
rhs3	R	R	-	-	RW	√	√	-
rhs4	R	R	-	-	RW	-	√	-

Table 6.4: LU benchmark access pattern for the different structures and summary of techniques used at each parallel loop.

On the other hand, the average time per loop shows that the execution time of each orphaned loops (`jacld, blts1, blts2, jacu, buts1` and `buts2`) is incredible small but they are executed a lot of times.

Another effect is that the idle time increases with the number of nodes, basically due to the manual synchronization.

Most of the loops modifies the resulting variable `rsd`. The variables `u` and `rho_i` are written once at `ssor2` and `rhs1` respectively, and read multiple times at `rhs, jacld, jacu` and `buts`. And the remaining variables `a, b, c, d, au, bu, cu` and `du` are used locally at the parallel loops `jacld, blts` and `jacu, buts` (see Table 6.4).

Final version details

The final version uses the align in the main loops (`ssor1, ssor2, rhs1, rhs2` and `rhs3`) to exploit the spatial locality; and the presend in the same loops and in the `rhs4`, because it is the loop that changes the parallelized dimension.

The small granularity of the orphaned loops limits the applicability of the presend technique, because the presend maps a set of accessed pages to

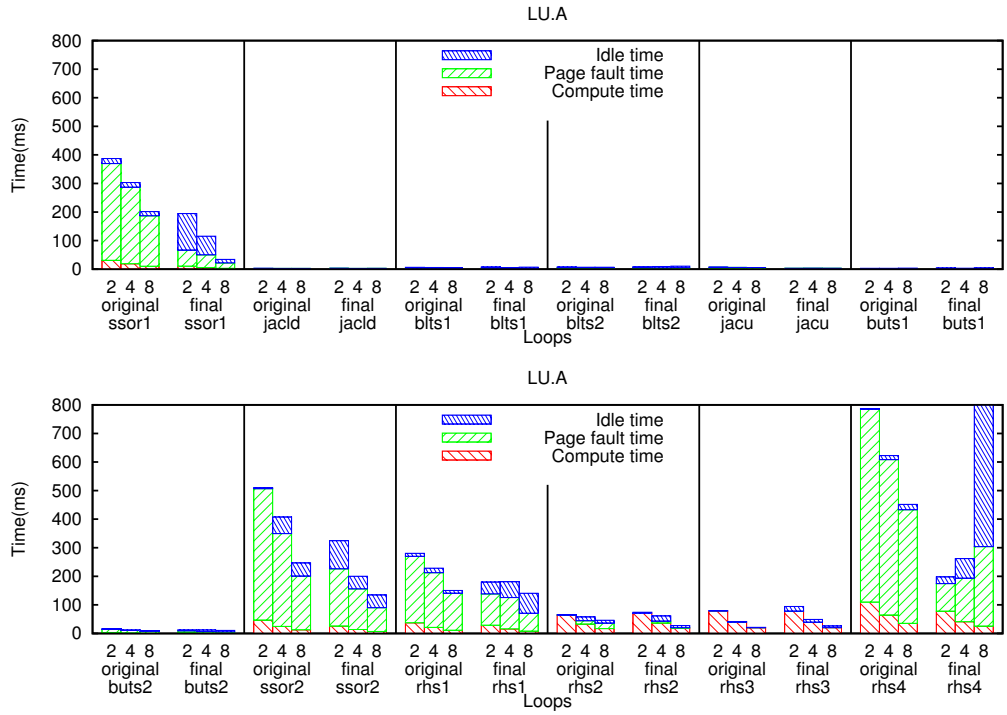

Figure 6.21: Average time per loop for different versions of LU class A.

a region of code, and, in this case, the same region of code (the orphaned loops) accesses different sets of pages at each iteration.

To overcome this situation, the chopper allows us to join different parallel loops in a single region. And therefore, we create two bigger regions: one containing the `jacld, blts1` and `blts2`, and the other containing `jacu, buts1` and `buts2`.

Figure 6.21 shows the results for this final version and compares them with the original version.

The number of page faults is reduced for all loops and the idle time is also slightly increased. In the `rhs4` the number of page faults is reduced, but the page fault time and, specially, the idle time grows with the number of nodes arriving to its maximum with 8 nodes. The problem is that the change in parallelization index needs to move a lot of pages and the computation time is too small as explained in other benchmarks.

	Execution time (seconds)	
Nodes	Original	Final
Seq	423.64	423.64
1	429.20	422.64
2	1190.11	884.46
4	978.80	848.75
8	814.44	863.24

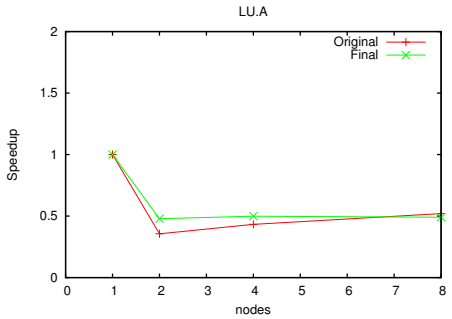

Figure 6.22: LU Class A performance results.

The behavior of this benchmark when executed in parallel in our DSM have worse times than when executed serially, as the idle time shows in the Figure 6.22. The final version improves the performance slightly for small number of nodes but it ends with a worse behavior with 8 nodes.

6.4.6 MG

The structure of this benchmark is shown in Figure 6.23. This benchmark has the peculiarity that the dimension of the shared arrays changes at each iteration, making impossible the use of the presend techniques, because it will learn a wrong access pattern and therefore a wrong portion of the total array would be presend.

In addition, the MG executes in the same iteration different instances of the same parallel loop, this means that even there are 6 parallel loops (`rprj`, `zero`, `interp`, `resid`, `psinv` and `comm`) they are executed multiple times and, again, the predictor will be unable to detect this sequence.

Original version details

Figure 6.24 shows the total time per loop for the MG benchmark, and Figure 6.25 shows the performance results.

```
1  do it=1, niter
2       do k=lt, lb+1, -1
3            call rprj(k)
4       enddo
5       k = lb
6       call zero(k)
7       call psinv(k)
8       do k=lb+1, lt-1
9            call zero(k)
10           call interp(k)
11           call resid(k)
12           call psinv(k)
13      enddo
14      k=lt
15      call interp(k)
16      call resid(k)
17      call psinv(k)
18      call resid(k)
19 enddo
```

```
1 subroutine zero(z,n1,n2,n3)
2 !$omp parallel do
3       do i3=1,n3
4          do i2=1,n2
5             do i1=1,n1
6                z(i1,i2,i3)=0.0D0
7             enddo
8          enddo
9       enddo
```

Figure 6.23: Structure of the MG algorithm and `zero` subroutine.

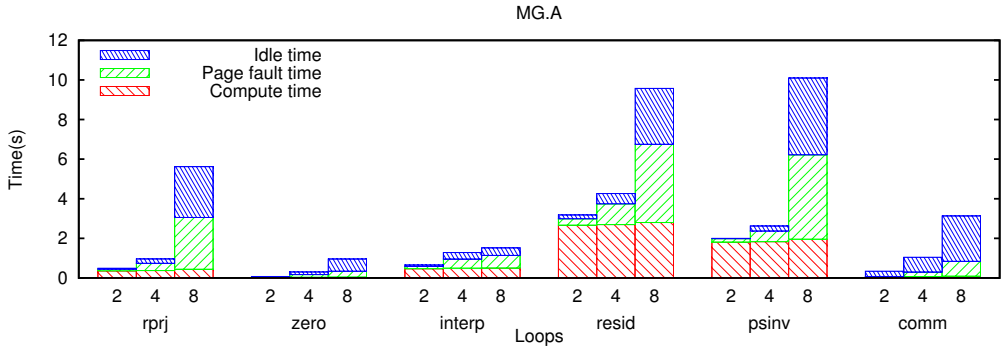

Figure 6.24: Total time per loop for original version of MG class A.

Nodes	Execution time (seconds) Original
Seq	21.94
1	23.43
2	27.28
4	27.26
8	30.78

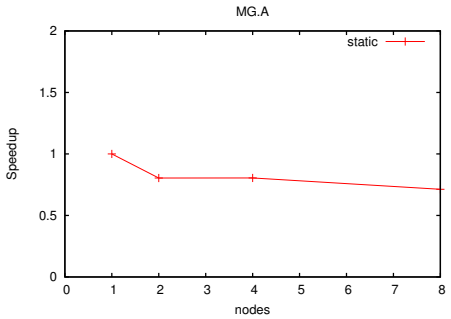

Figure 6.25: MG Class A performance results.

```
1  do it= 1, niter
2    call evolve
3    call fft
4    call checksum
5  enddo
```

```
1  subroutine fft
2    call cffts3
3    call cffts2
4    call cffts1
5  end
```

Figure 6.26: Algorithm structure of the FT benchmark and the `fft` subroutine.

Loop	u1	y1	y2	Align	Prsnd	Chppr
evolve	W	-	-	-	√	-
cffts3	RW	RW	RW	-	√	-
cffts2	RW	RW	RW	-	-	-
cffts1	RW	RW	RW	-	√	-
checksum	R	-	-	-	-	-

Table 6.5: FT benchmark access pattern for the different structures and summary of techniques used at each parallel loop.

6.4.7 FT

The structure of this benchmark is shown in Figure 6.26. It iterates over three functions: `evolve, fft` and `checksum`. The main function `fft` uses three inner functions to calculate a Fourier Transformation on each dimension of the 3 dimensional matrix using blocking. In total we have five parallel loops for these functions: `evolve, cffts3, cffts2, cffts1` and `checksum`.

The benchmark uses three different arrays that are used as linearized matrices. As Table 6.5 shows, the main array (`u1`) is used by all parallel loops, while the other arrays (`y1` and `y2`) are used only in the `fft` function.

Original version details

As usual with these benchmarks parallelizing on a 3D matrix, one of the parallel loops is parallelized using a different dimension than the other two, producing a high number of page faults in that loop and in the adjacent one (`cffts3` and `cffts2`) as the total time per loop graph shows in Figure 6.27.

Another detail of this benchmark is that the `fft` function has two different

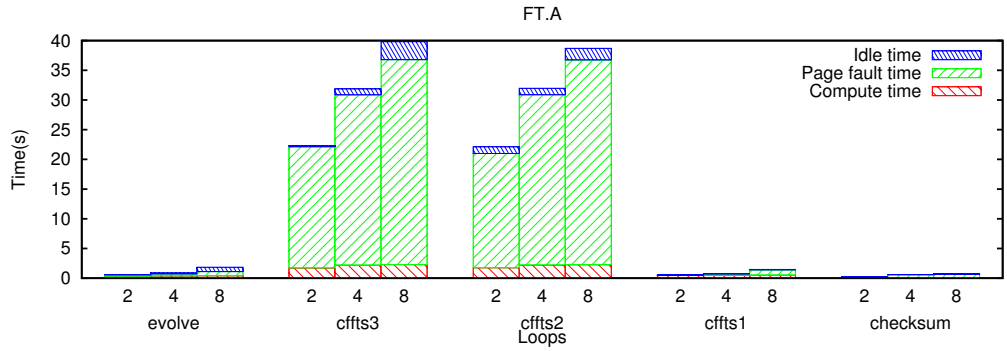

Figure 6.27: Total time per loop for original version of FT class A.

behaviors depending on the value of one of its parameters. This parameter controls the order in which the dimensions are parallelized. In one case it executes the inner functions in the order explained in the text (`cffts3`, `cffts2` and `cffts1`) and, in the other, it reverses the order. This change in the order of the `cffts` functions happens only once at the beginning of the benchmark, so it can be easily ignored. But this kind of algorithms, where the behavior depends on a parameter, may prohibit the use of our prediction techniques.

Final version details

To improve the performance of this benchmark and remove the page faults, we apply the presend technique in the loop changing the dimension (`cffts3`) and in the previous ones (`cffts1` and `evolve`).

The idea is that the presend in the `cffts3` loop reduces the number of page faults in the following loop `cffts2`, because `cffts3` is the producer of the variables `u1, y1`, and `y2`, which are consumed at the `cffts2`.

With the same goal, we use the presend in `evolve` (which updates the `u1` array) and `cffts1` (which updates `y1` and `y2`) to reduce the page faults in `cffts3`. This is summarized in Table 6.5.

The final results appear in Figure 6.28 and they show that the final version reduces the number of page faults in the `cffts3` and `cffts2` but the *idle time* increases. This increase can be explained with the high number of messages

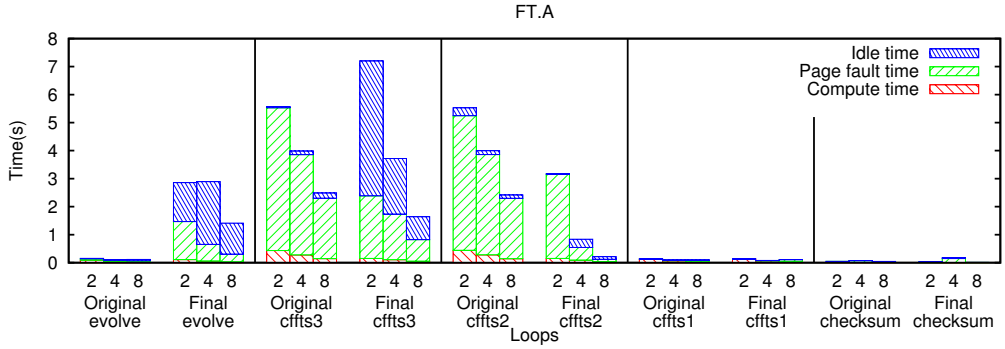

Figure 6.28: Average time per loop for different versions of FT class A.

	Execution time (seconds)	
Nodes	Original	Final
Seq	6.89	6.89
1	7.19	7.23
2	78.94	86.98
4	56.91	57.65
8	35.28	27.73

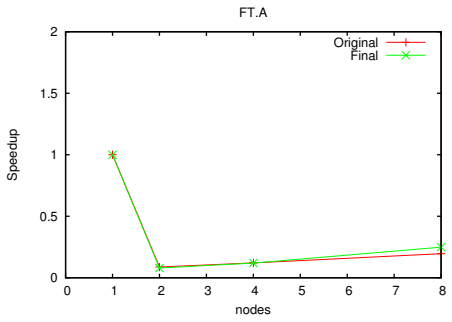

Figure 6.29: FT Class A performance results.

sent at the end of the loops, which interfere in the normal execution of the infoserver. It is interesting to note, that this overhead is reduced when the number of nodes grows, basically due to the reduction of data to be transferred. In fact, it seems that in this benchmark we did the wrong thing using the presend in the `evolve` because the final time solving page faults and the idle time have increased significantly.

The main problem with this benchmark is that the computation time is not enough to overlap all the communication.

The performance results for this benchmark appear in Figure 6.29. The serial version takes just 7 seconds, and when executed in parallel it takes an order of magnitude more to execute. Meaning that the cost of remote memory handling overcomes the computation time.

6.5 Conclusions

In this chapter we evaluate the execution of the OpenMP version of the NAS benchmarks on a distributed environment using a sequential consistency DSM. We use a combination of the techniques explained in previous chapters: boundaries alignment, presend/preinvalidation and chopper; to obtain the best performance results on each benchmark.

For each benchmark we evaluate the original version with the final version, which uses our techniques. The analysis for each benchmark contains the algorithm structure, the shared variables access pattern, the execution time of each parallel loop, a comparison between the average execution time of each parallel loop and the performance results for original and final benchmark versions.

Table 6.6 summarizes the results from all benchmarks. It shows four different columns:

Total The total number of analyzed parallel loops.

Representative The number of loops that consumes most time due to page faults and therefore can be solved.

Better The number of loops that reduces their execution time using our techniques more than 10% of the original.

Worse The number of loops that increase their final execution time by more than 10%.

One loop represents three executions (2, 4 and 8 nodes) of a parallel loop.

A total of 54 parallel loops have been analyzed, from which 29 were responsible for the bad performance of the benchmark, and with our techniques an average of 60% of these parallel loops have been improved.

The quality of this improvement can be seen in Table 6.7, that shows the total number of executions for all parallel loops of each application, counting three executions per parallel loop, corresponding to the execution with 2, 4 and 8 nodes. It counts the number of executions that reduce their execution time by more than 10, 25, 50 and 75 percent and the number of executions

	Total	Representative	Better	Worse
EP	2	-	-	-
CG	10	4	3	-
BT	5	3	3	-
SP	14	6	5.33	0.67
LU	12	10	4.67	3
FT	5	2	1.33	0.33
MG	6	4	-	-
Total	54	29	17.33	4

Table 6.6: Summary of number of loops per benchmark, showing the loops using more time due to page faults (representative) and the effects of our techniques showing the loops that reduce the number of page faults and the loops whose execution time increases.

	Total	Representative	Worse	10-	25-	50-	75-
EP	6	0	0	0	0	0	0
CG	30	12	0	1	4	4	0
BT	15	9	0	0	4	2	3
SP	42	18	2	1	2	4	9
LU	36	30	9	1	8	4	1
FT	15	6	1	0	2	0	2
MG	18	12	0	0	0	0	0
TOTAL	162	87	12	3	20	14	15

Table 6.7: Number of executed parallel loops per benchmark that, using our techniques, reduce their execution time by more than 10, 25, 50 or 75 percent; or decrease more than 10 percent (Worse). It shows the total number and the representative executions, executions of the loops using more time due to page faults.

which increase their execution time by more than 10 percent (Worse). All columns are exclusive, meaning that they do not count the executions from greater percentages.

The main conclusions of the contributions are:

- The boundary alignment is specially useful for removing false sharing in linear arrays, but its importance depends on the importance of the false sharing in the application.
- The schedule reuse is also useful to avoid page faults due to slight changes in the parallel loop limits.
- Presend/Preinvalidation techniques are the most important techniques to avoid page faults, following a model where the producer sends the data before the consumer requests it.
- The chopper technique is also useful to overcome the limitations of the network if, and only if, there is enough computation time to overlap the communication phase.
- Finally, there are cases, where our techniques can not be applied: changes in the size of the matrices, parallel loops with different behaviors during execution time and parallel loops with too few computation.

Chapter 7

Conclusions

- No! Harry! No! Don't look at the light!
- I can't help it. It's so beautiful...
A bug's life (1998)
Disney Pixar

Abstract

In this chapter, we summarize the main contributions of this thesis: a method to avoid false sharing in OpenMP loop worksharings, a mechanism to send/invalidate data at the owner side, and finally the automatic distribution of coherence data messages instead of centralizing them.

7.1 Contributions of this work

In this thesis we have studied the execution of OpenMP applications on top of distributed memory platforms using a shared memory abstraction provided by a DSM layer.

The main idea in this thesis is that there must exist a tight cooperation between both layers to obtain the maximum performance when executing OpenMP applications on top of a DSM.

This idea is enforced with the main contributions of this thesis:

1. **Tolerate false sharing adapting the application at runtime**
2. **Reduce remote memory latencies**
3. **Avoid network congestion**
4. **Proposal of OpenMP extensions**

Even it has not been tested, we think that the contributions explained in this work can benefit other DSMs with different consistency models, like the relaxed ones. Mainly, because the problems we solve here are also present in DSMs using relaxed consistencies but they just avoid them.

7.1.1 Tolerate false sharing adapting the application at runtime

We have shown that an inadequate iteration distribution from the OpenMP runtime produces false sharing, and it can be eliminated by adapting the iteration space at runtime with some cooperation from the DSM. The results also shown that an application can suffer from false sharing without affecting too much the final performance result, because the sharing is small compared with the total computation time.

Our contribution is a new *Align* scheduler that adapts a parallel loop to the memory boundaries that it traverses, so false sharing is avoided. Additionally, it exploits temporal locality by ensuring that the same threads access the same pages that have been used in previous schedule.

	Total	Representative	Using Align
EP	2	-	-
CG	10	4	4
BT	5	3	5
SP	14	6	-
LU	12	10	5
FT	5	2	-
MG	5	4	-
Total	53	29	14

Table 7.1: Number of parallel loops using the *align* technique on NAS Benchmarks.

This technique uses the iteration space of the parallel loop to avoid the false sharing, and therefore only loops that access one position of the array at each iteration can be adapted. Multidimensional arrays using a parallel algorithm with a loop for each dimension can not be adapted directly, because at each iteration of the outer loop, more than one position is modified. But these arrays could be adapted if the code is linearized in such a way that the previous rules is accomplished.

As Table 7.1 shows, the *Align* scheduler have been used in 14 parallel loops, and results shows that, on one hand, it is able to avoid false sharing producing trashing, and it reduces execution time by reusing the same schedule. On the other hand, the scheduler adds small overhead and it obtains similar execution times than an static schedule when the scheduler is not able to find a better schedule.

We have also shown that the false sharing problem depends on the application and thus if the computation time available is small, then the false sharing becomes a problem; but if the computation time is big enough then its effects are not relevant.

7.1.2 Reduce memory latencies

We have shown that the performance of the applications can be improved following a producer-consumer model like MPI. Typical behavior of a sequen-

	Total	Representative	Using Presend
EP	2	-	-
CG	10	4	4
BT	5	3	5
SP	14	6	9
LU	12	10	6
FT	5	2	3
MG	5	4	-
Total	53	29	27

Table 7.2: Number of parallel loops using the *presend* technique on NAS Benchmarks.

tial consistency DSM corresponds to a two-way communication, where one thread requests a page address and the master answers with its page content.

Our contribution propose new mechanisms offering a one-way communication between the data producer and its consumers: the pre-send and the pre-invalidation.

The pre-send sends copies of a page produced at a producer node to all the nodes that will consume it before it is accessed. The pre-invalidation invalidates the local copy of a page before it is written by a different node.

These techniques are used by the OpenMP runtime to send all data that will be used/consumed in the next region at the end of the OpenMP parallel loops

Table 7.2 summarizes how many loops from the NAS benchmarks have been extended with the Presend technique and the experimental results show that the number of page faults is highly reduced compared to the baseline. It also shows that the performance of a sequential consistency DSM using a sender initiated communication is similar to the performance obtained by a relaxed consistency DSM like TreadMarks.

The results also show that the technique forwards data from a parallel region to the next one without any synchronization, meaning that enough time is needed to send all data pages before they are used.

7.1.3 Avoid network congestion

The presend improves the performance of the sequential consistency by changing the way the data is transfered. When using the sequential consistency alone, the data producer is passive, and the consumer node requests pages on an on-demand basis and, therefore, lots of small requests are distributed along the computation. In contrast, when using the presend, the data producer is active and it sends all the pages that will be needed by the consumer after the computation that produces them and, thus, a high number of requests are sent at the same time.

This means that the use of a data forwarding method, like our presend technique, transforms a sequential consistency DSM to behave similarly to a relaxed consistency one, where the coherence messages are also sent at the end of a computation phase.

The difference is that in a relaxed consistency DSM, the data is not forwarded to the consumer that will use it but the home nodes of the data pages. This means that a technique like ours could be also used on this kinds of DSMs.

To overcome the negative effects of concentrating coherence messages at single points, we proposed the chopper mechanism, and distribute the messages during the computation phase.

Figure 7.3 presents 15 parallel loops from the NAS benchmarks that use the chopper mechanism. Results show that the chopper technique is limited by the computation time available to overlap the distribution of all network data and that the number of VSP to use at each thread should be chosen wisely to achieve the best results.

7.1.4 Proposal of OpenMP extensions

As a result of the previous contributions, we propose five new directives to extend the OpenMP specification. The proposals are designed to:

Calculate an schedule to avoid false sharing The new *Align* scheduler calculates new schedules for specific parallel loops taking into account

	Total	Representative	Using Chopper
EP	2	-	-
CG	10	4	1
BT	5	3	2
SP	14	6	6
LU	12	10	6
FT	5	2	-
MG	5	4	-
Total	53	29	15

Table 7.3: Number of parallel loops using the *chopper* technique on the NAS Benchmarks.

the page boundaries to avoid false sharing. It also allows a parallel loop to reuse a previously calculated schedule and so we propose a modification to the original SCHEDULE directive adding this information:

```
1 !$omp SCHEDULE ( ALIGN, <schedule>, <operation> )
```

Enable the presend mechanism The presend mechanism allows to forward all necessary data from a parallel loop to the next parallel loops that need this data. We propose a new directive to enable this mechanism for a specific parallel loop:

```
1 !$omp PRESEND
```

Enable the distribution of coherence messages We finally propose some directives that allow the programmer to mark a specific region of code suffering from the network congestion problem and enable the distribution of network messages through the virtual synchronization points:

```
1 !$omp start_region(id)
2 !$omp stop_region(id)
3 !$omp vsp
```

7.2 Future Work

In the near future, there are still some aspects that should be investigated. In first place there are some implementation details that should be handled, like the coherence protocol to strictly follow a MSI or a better predictor. In second place, there are some topics that should be researched in more detail like: grouping of multiple pre-send and pre-invalidation messages, use of the compiler to give hints for the chopper or aggregation of DSMs. Finally, another research area could be to use the DSM on many-core processors without cache coherence like, for example, the Single-Chip Cloud (SCC) [MRL+10] experimental processor from Intel Labs.

7.2.1 Implementation details

Use a real MSI coherence protocol in the master node as well

The master node when in a SHARED state will always has a copy of the page. This a very strict limitation and is an annoyance for the preinvalidation feature. There are situations when the master node wants to invalidate his own copy but it can not due to this restriction.

For example, when the master node has finished a region of code that reads a page which is needed for writing in the following region by another node, but still there are other nodes reading it. In this case, the state of the page must remain in SHARED, meaning that the invalidation has been ineffective for this node, and so the destination node will never receive an upgrade and it will generate a write page fault.

Improve the predictor

The implemented predictor is quite simple, and a better prediction may detect special cases where the current design is not able to follow, and so may improve the final performance.

7.2.2 Further research

Multipresend and Multipreinvalidation

Usually, when forwarding data from a region to the next one, there are a lot of pages being sent, and the current implementation requests a presend or a preinvalidation in a sequential manner. This means that a message is generated for each page, and each message is processed by its page master node. The latency of this mechanism could be improved if the pages were grouped. For example, instead of sending a hundred messages saying that pages from addresses ranging from 1 to 100 should be invalidated and forwarded to another node, a single message could be sent just saying the same information.

The idea of the multipresend and multipreinvalidation is to group the presend and preinvalidation messages from a node into a single message. Of course, the computation time at the sending and receiving sides will be increased to pack and unpack the pages, but we think that it is more important to reduce the use of the network than the computing time because it is usually cheaper and faster. A potential problem of this mechanism is that these bigger messages could interfere the normal behaviour of the DSM infoservers, delaying the processing of some other important coherence or control messages, therefore it is necessary to evaluate this technique.

Hints for the chopper

Currently the use of the chopper mechanism is quite limited and depends exclusively on the expertise of its user. Some hints from the compiler or even the runtime should make his life better, by giving instructions on when and where to put the right VSPs. It would be nice if the compiler could set this information on its own.

Multiple DSMs instances to aggregate nodes

It has been shown, that when the number of nodes increases too much, the fine granularity of the application may kill the final performance. While the

performance with small number of nodes remain acceptable.

A conclusion of this thesis is that the DSM can be used to aggregate small number of nodes into bigger shared memory machines, meaning that it is useful to add more threads to execute one application. Big clusters with a high number of nodes may use this functionality to build smaller shared memory machines inside with more threads than the single nodes. This idea is similar to the idea presented by Schulthess et al. [SSSW00] agregating nodes in the World-Wide Web, what they call DSM-Communities, but limited to a cluster.

Aggregate cores in a many-core processor

The number of cores per single die is expected to increase in the foreseeable future and so there will be a need to know how to connect these cores and how to program the resulting many-core processor. One of this machines is the Single-Chip Clour processor from Intel Labs [HDH+10]. A many-core processor without any cache coherence between the cores similar to a cluster.

The idea presented before of aggregating nodes can be applied to this processor aggregating cores, taking profit of a fast communication layer [vdWMH11]. A porting of the NanosDSM software to this platform should be performed in first place.

Bibliography

[ACCL00] I. Alexander, LAI Chi, and LEI Chin-Laung. A False-Sharing Free Distributed Shared Memory Management Scheme. *IEICE TRANSACTIONS on Information and Systems*, E83-D(4):777–788, 04 2000.

[ACD+96a] Sarita V. Adve, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *HPCA*, pages 26–37, 1996.

[ACD+96b] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACRA98] Boon Seong Ang, Derek Chiou, Larry Rudolph, and Arvind. The start-voyager parallel system. In *IEEE PACT*, pages 185–, 1998.

[ACRZ97] Cristiana Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99, New York, NY, USA, 1997. ACM Press.

BIBLIOGRAPHY

[AFP02] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, October 2002.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[APS+99] M. Allman, V. Paxson, W. Stevens, et al. TCP congestion control. RFC 2581, April 1999.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BCJ+94] D.F. Bacon, J. Chow, D.R. Ju, K. Muthukumar, and V. Sarkar. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In *Proceedings of CASCON'94*, 1994.

[BCZ89] J.K. Bennet, J.B. Carter, and W. Zwaenepoel. Munin: shared memory for distributed memory multiprocessors. Technical Report TR89-91, Departament of Computer Science, Rice University, April 1989.

[BDG+04] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Sixth European Workshop on OpenMP*, 2004.

[BE05] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. In Arvind and Larry Rudolph, editors, *ICS*, pages 189–198. ACM, 2005.

[BE06] Ayon Basumallik and Rudolf Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems.

In Josep Torrellas and Siddhartha Chatterjee, editors, *PPOPP*, pages 119–128. ACM, 2006.

[BGG+95] Francois Bodin, Elana Granston, Elana D. Granston, Thierry Montaut, and Thierry Montaut. Page-level affinity scheduling for eliminating false sharing. In *In Fifth Workshop on Compilers for Parallel Computers, Malaga*, pages 175–186, 1995.

[BME02] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Towards openmp execution on software distributed shared memory systems. In *Lecture Notes in Computer Science*, volume 2327, 2002.

[Boa04] OpenMP Architecture Review Board. OpenMP Language Specification v2.5, November 2004. www.openmp.org.

[CBMC09] J. Costa, J. Bueno, X. Martorell, and A. Cortes. Measuring tcp bandwidth on top of a gigabit and myrinet network. Technical report, Universitat Politècnica de Catalunya, 2009.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164. ACM New York, NY, USA, 1991.

[CBZ95] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, 1995.

[CCM+04] J.J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP Applications Efficiently on an Everything-Shared SDSM. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS-04)*, Santa Fe, New Mexico, April 2004. IEEE.

[CCM+06] J. Costa, A. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running openmp application efficiently on an everything-shared sdsm. *Journal of Parallel and Distributed Computing*, 66:647–658, May 2006.

[CCM+09] J. Costa, A. Cortes, X. Martorell, E. Ayguade, and J. Bueno. Overlapping communication with computation on nas bt benchmark. In *Fifth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, pages 55–58, Terrassa (Spain), July 2009. Academia Press.

[CCM+10] J.J. Costa, T. Cortes, X. Martorell, J. Bueno, and E. Ayguade. Transient congestion avoidance in software distributed shared memory systems. *Parallel and Distributed Computing Applications and Technologies, International Conference on*, 0:357–364, 2010.

[Cla85] David D. Clark. The structuring of systems using upcalls. *SIGOPS Oper. Syst. Rev.*, 19:171–180, December 1985.

[CS97] Jyh-Herng Chow and Vivek Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *In Proceedings of the 26th International Conference on Parallel Processing*, pages 396–403, 1997.

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *ICPP (1)*, pages 377–381, 1991.

[FA96] Vincent W. Freeh and G.R. Andrews. Dynamically controlling false sharing in distributed shared memory. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*, pages 403–411, Washington, DC, USA, 1996. IEEE Computer Society.

[For94] M.P.I. Forum. Mpi: A message-passing interface standard. Technical report, UT-CS-94-230, 1994.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, volume 18, pages 354–368, 1990.

[GP91] M. Gupta and D.A. Padua. Effects of program parallelization and stripmining transformation on cache performance in a multiprocessor. In *Proceedings of the International Conference on Parallel Processing (ICPP'91)*, pages 301–304, Texas, USA, 1991.

[Gra93] Elana D. Granston. Toward a compile-time methodology for reducing false sharing and communication traffic in shared virtual memory systems. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 1993.

[GW93] Elana D. Granston and Harry A. G. Wijshoff. Managing pages in shared virtual memory systems: getting the compiler into the game. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 11–20, New York, NY, USA, 1993. ACM.

[HCL05] Lei Huang, Barbara Chapman, and Zhenying Liu. Towards a more efficient implementation of openmp for clusters via translation to global arrays. *Parallel Computing*, 31(10-12):1114–1139, October-December 2005.

[HCZ00] Y. Charlie Hu, Alan L. Cox, and Willy Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *SC*, 2000.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain,

T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proceedings of the International Solid-State Circuits Conference*, Feb 2010.

[HDS08] J.P. Hoeflinger and B.R. De Supinski. The openmp memory model. *Lecture Notes in Computer Science*, 4315:167, 2008.

[HIH+00] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 158 –163 vol.1, 2000.

[HJMR02] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster. In *Workshop on OpenMP Applications and Tools (WOMPAT'02)*, August 2002.

[Hoe06] J.P. Hoeflinger. Extending OpenMP to clusters. *White Paper, Intel Corporation*, 2006.

[HST99] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *Proceedings of the High Performance Computing and Networking (HPCN'99)*, volume LNCS 1593, pages 463–472, Amsterdam, Netherlands, April 1999. Springer.

[IS99a] A. Itzkovitz and A. Schuster. Distributed shared memory: Bridging the granularity gap. In *Proceedings of the First ACM Workshop on Software Distributed Shared Memory (WSDSM)*. Citeseer, June 1999.

[IS99b] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage - fine-grain sharing in page-based dsms. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 215–228, February 1999.

[ISL98] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory Comput. Syst.*, 31(4):451–473, 1998.

[ISW96] A. Itzkovitz, A. Schuster, and L. Wolfovich. Millipede: Towards standard interface for virtual parallel machines on top of distributed environments. Technical Report Technical Report 9607, Technion IIT, 1996.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–188, New York, NY, USA, 1995. ACM Press.

[Jég00] Yvon Jégou. Controlling distributed shared memory consistency from high level programming languages. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1800 of *Lecture Notes in Computer Science*, pages 293–300. Springer Berlin / Heidelberg, 2000.

[Jég03] Yvon Jégou. Implementation of page management in Mome, a user-level DSM. In *Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 479–486, Tokyo, Japan, May 2003. Held in conjunction with CCGrid 2003. IEEE TFCC.

[JFY99] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical

Report NAS-99-011, NASA Ames Research Center, October 1999.

[JK88] Van Jacobson and Michael J. Karels. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18:314–329, August 1988.

[KCDZ94] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 10–10. USENIX Association Berkeley, CA, USA, 1994.

[KCRB03] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Reducing false sharing and improving spatial locality in a unified compilation framework. *Parallel and Distributed Systems, IEEE Transactions on*, 14(4):337–354, 2003.

[KCZ92] Peter J. Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA*, pages 13–21, 1992.

[Kel96a] P. Keleher. The CVM manual. Technical Report CS-TR-3545, University of Maryland, 1996.

[Kel96b] P.J. Keleher. The relative importance of concurrent writers and weak consistency models. In *International Conference on Distributed Computing Systems*, volume 16, pages 91–99. IEEE Computer Society Press, 1996.

[Kel99] Peter J. Keleher. Tapeworm: High-level abstractions of shared accesses. In *OSDI*, pages 201–214, 1999.

[KG04] Manjunath Kudlur and R. Govindarajan. Performance analysis of methods that overcome false sharing effects in software dsms. *Journal of Parallel and Distributed Computing*, 64(8):887–907, 2004.

[KKH03] Y.S. Kee, J.S. Kim, and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Supercomputing 2003 (SC'03)*, November 2003.

[KRC99] Mahmut T. Kandemir, J. Ramanujam, and Alok N. Choudhary. Improving cache locality by a combination of loop and data transformations in an integrated framework. *IEEE Transactions on Computers*, 48(2):159–167, 1999.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept. 1979.

[LCBZ97] K. Li, J. Carter, J. Bennett, and W. Zwaenepoel. Ivy: A shared virtual memory system for parallel computing. *Distributed Shared Memory: Concepts and Systems*, page 121, 1997.

[LCD+97] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *PPOPP*, pages 48–56, 1997.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25:63–79, 1992.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS*, pages 63–74, 1991.

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

[Mar94] T.J.; Markatos, E.P.; LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Parallel and Distributed Systems, IEEE Transactions on*, volume 5, pages 379 – 400. IEEE Computer Society, Apr 1994. 1045-9219.

[ME08] Seung-Jai Min and Rudolf Eigenmann. Optimizing irregular shared-memory applications for clusters. In Pin Zhou, editor, *ICS*, pages 256–265. ACM, 2008.

[MLNA96] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A Library Implementation of the Nano-Threads Programming Model. In *Euro-Par, Vol. II*, pages 644–649, August 1996.

[mpi] Message passing interface web page. www-unix.mcs.anl.gov/mpi/.

[MRL+10] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[NAAL01] Dimitrios S. Nikolopoulos, Ernest Artiaga, Eduard Ayguadé, and Jesús Labarta. Exploiting memory affinity in openmp through schedule reuse. *SIGARCH Computer Architecture News*, 29(5):49–55, 2001.

[NS01] Nitzan Niv and Assaf Schuster. Transparent adaptation of sharing granularity in multiview-based dsm systems. In *IPDPS*, page 38. IEEE Computer Society, 2001.

[OSHI03] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa. Performance of cluster-enabled OpenMP for the SCASH software distributed

shared memory system. In *Proc. of the 3rd Intl. Symp. on Cluster Computing and the Grid*, pages 450–456, 2003.

[PLCG95] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *Transputer and occam developments: WoTUG-18: proceedings of the 18Tth world occam and Transputer User Group Technical Meeting, 9th-13th April 1995, Manchester, UK*, volume 44, page 17. Ios Pr Inc, 1995.

[PR01] M. Pizka and C. Rehn. Murks-a POSIX threads based DSM system. *PDCS'01*, pages 642–648, 2001.

[SB97] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proc. of the USENIX Windows NT Workshop*, 1997.

[SBW91] Joel H. Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrency - Practice and Experience*, 3(6):573–592, 1991.

[SHH01] Mitsuhisa Sato, Hiroshi Harada, and Atsushi Hasegawa. Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system. *Scientific Programming*, 9(2-3):123–130, 2001.

[SHI00] M. Sato, H. Harada, and Y. Ishikawa. OpenMP Compiler for a Software Distributed Shared Memory System SCASH. In *WOMPAT2000*, July 2000.

[SSC98] Mark R. Swanson, Leigh Stoller, and John B. Carter. Making distributed shared memory simple, yet efficient. In *HIPS*, pages 2–. IEEE Computer Society, 1998.

[SSKT99] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *EWOMP '99*, pages 32–39, September 1999.

[SSMBL94] Barbara Simons, Vivek Sarkar, Jr. Mauricio Breternitz, and Michael Lai. An optimal asynchronous scheduling algorithm for software cacheconsistency. In *Proceedings of the 27th Hawaii International Conference on Software Technology*, volume II, pages 502–511, January 1994.

[SSS+99] O. Schirpf, M. Schoettner, P. Schulthess, S. Traub, and M. Wende. Dsm-java: Foundation of a lean distributed operating system. In *Proceedings of the International Workshop on Distributed Computing on the Web*, Rostock, Germany, 1999.

[SSSW00] Peter Schulthess, Oliver Schirpf, Michael Schöttner, and Moritz Wende. Dsm-communities in the world-wide web. In Peter G. Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors, *DCW*, volume 1830 of *Lecture Notes in Computer Science*, pages 65–73. Springer, 2000.

[STS98] M. Schoettner, S. Traub, , and P. Schulthess. A transactional dsm operating system in java. In *Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1998.

[SWG92] J. Pal Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

[TGJ93] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *SC*, pages 410–419, 1993.

[TLH90] J. Torrellas, M.S. Lam, and J.L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2, pages 266–270, 1990.

[TMSW08] C. Terboven, D. Mey, D. Schmidl, and M. Wagner. First Experiences with Intel Cluster OpenMP. *LECTURE NOTES IN COMPUTER SCIENCE*, 5004:48, 2008.

[top] Top 500 supercomputer sites. www.top500.org.

[VBB01] R. Veldema, RAF Bhoedjang, and HE Bal. Jackal, a compiler based implementation of java for clusters of workstations. In *Proc. of PPoPP*. Citeseer, 2001.

[vdWMH11] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[VL00] S.P. Vanderwiel and D.J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30–44, 1991.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.