IT Licentiate theses
2004-005

Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures

HENRIK LÖF

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures

BY
HENRIK LÖF

October 2004

DIVISION OF SCIENTIFIC COMPUTING
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Scientific Computing
at Uppsala University 2004

Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures

Henrik Löf
`henrik.lof@it.uu.se`

Division of Scientific Computing
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

`http://www.it.uu.se/`

© Henrik Löf 2004
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Solving Partial Differential Equations (PDEs) is an important problem in many fields of science and engineering. For most real-world problems modeled by PDEs, we can only approximate the solution using numerical methods. Many of these numerical methods result in very large systems of linear equations. A common way of solving these systems is to use an iterative solver such as the method of conjugate gradients. Furthermore, due to the size of these systems we often need parallel computers to be able to solve them in a reasonable amount of time.

Shared memory architectures represent a class of parallel computer systems commonly used both in commercial applications and in scientific computing. To be able to provide cost-efficient computing solutions, shared memory architectures come in a large variety of configurations and sizes. From a programming point of view, we do not want to spend a lot of effort optimizing an application for a specific computer architecture. We want to find methods and principles of optimizing our programs that are generally applicable to a large class of architectures.

In this thesis, we investigate how to implement the method of conjugate gradients efficiently on shared memory architectures. We seek algorithmic optimizations that result in efficient programs for a variety of architectures. To study this problem, we have implemented the method of conjugate gradients using OpenMP and we have measured the runtime performance of this solver on a variety of both uniform and non-uniform shared memory architectures. The input data used in the experiments come from a Finite-Element discretization of the Maxwell equations in three dimensions of a fighter-jet geometry.

Our results show that, for all architectures studied, optimizations targeting the memory hierarchy exhibited the largest performance increase. Improving the load balance, by balancing the arithmetical work and minimizing the number of global barriers showed to be of lesser importance. Overall, *bandwidth minimization* of the iteration matrix showed to be the most efficient optimization.

On non-uniform architectures, proper data distribution showed to be very important. In our experiments we used page migration to improve the data distribution during runtime. Our results indicate that page migration can be very efficient if we can keep the migration cost low. Furthermore, we believe that page migration can be introduced in a portable way into OpenMP in the form of a directive with a *affinity-on-next-touch* semantic.

Acknowledgments

First of all, I would like to thank all the people involved in writing the papers presented in the thesis. Without your efforts and hard work the papers would not have been completed. Thank you Jarmo, Sverker, Erik, Markus and Zoran.

I also would like to thank all the people at the Department of Information Technology for their support and many fruitful and interesting discussions. Special thanks goes to my supervisors: Jarmo, Sverker and Erik, the Ph.D students of the UART group: Erik, Martin, Dan, Håkan and Zoran and the real heroes down in the sysadmin dungeon: Jukka and Janne.

This work is in part supported by the Parallel and Scientific Computing Institute (PSCI) of Sweden and Sun Microsystems.

List of Appended Papers

This thesis is a summary of the following four papers. References to the papers are made using the capital letter associated with each paper.

Paper A H. Löf, M. Nordén and S. Holmgren. *Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers.* In Proceedings of the International Conference on Computational Science (ICCS) 2004, part II, LNCS 3037, pages 9-16, 2004. `http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3037&spage=9`
Also available as a technical report[43].

Paper B H. Löf and S. Holmgren. *affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System* In Proceedings of the Sixth European Workshop on OpenMP (EWOMP) 2004.

Paper C H. Löf, J. Rantakokko. *Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures*. Technical Report. Department of Information Technology, Uppsala University, oct-2004, nr 2004-48.

Paper D H. Löf, Z. Radovic and E. Hagersten. *THROOM - Supporting POSIX Multithreaded Binaries on a Cluster*. In Proceedings of the Euro-Par conference 2003, LNCS 2790, pages 760-769, 2003. `http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2790&spage=760`

Comments on my participation

Paper A I was responsible for the results regarding the conjugate gradient solver

Paper B I am the principal author of this paper

Paper C I am the principal author of this paper

Paper D I am the principal author of this paper

Contents

1. Introduction **1**
1.1. Contributions 2

2. Solving a Large Sparse System of Equations using Conjugate Gradients **3**
2.1. GEMS - An Industrial CEM Solver 3
2.2. The Mathematical Problem 5
2.2.1. The Method of Conjugate Gradients 5
2.3. Implementing Conjugate Gradients 5
2.3.1. Exploiting Sparsity 6
2.3.2. Implementing Conjugate Gradients for Sparse Systems of Equations 7
2.4. Optimizing Conjugate Gradients for Sparse Systems of Equations . . 8
2.4.1. Optimizing the Sparse Matrix-Vector Product 8

3. An Introduction to Shared Memory Architectures **10**
3.1. The Cache Coherency Problem 10
3.2. Software Distributed Shared Memory Systems 11
3.3. Examples of Shared Memory Architectures 11
3.3.1. DSZOOM, a Fine-grain SW-DSM System 12
3.4. Future Shared Memory Architectures 13

4. Parallelizing Conjugate Gradients for Shared Memory Architectures **14**
4.1. An Introduction to Shared Memory Programming 14
4.1.1. OpenMP 15
4.1.2. Programming SW-DSMs 15
4.2. Identifying Parallelism in the CG algorithm 16
4.2.1. An Analysis of the Parallel Overhead 17
4.2.2. Implementing Conjugate Gradients using OpenMP 17
4.3. Using Algorithmic Optimization to Improve Performance 18

5. Optimizing for Non-Uniform Shared Memory Architectures **20**
5.1. Architectural Optimizations on DSM Systems 20
5.1.1. Thread Scheduling on DSM Systems 20
5.1.2. Page Placement Policies on DSM Systems 21

5.1.3. Page Migration on DSM Systems 21
5.2. Improving the Geographical Locality of the CG Implementation . . . 22
5.3. Data Distribution in OpenMP 23

6. Conclusions 24

7. Future Work 25

A. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers 27
A.1. Introduction 27
A.2. Applications 28
A.3. Results 29
A.4. Conclusions 33

B. *affinity-on-next-touch*: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System 36
B.1. Introduction 36
B.2. Thread-data affinity on cc-NUMA systems 38
B.3. The application 39
B.4. The cc-NUMA system 40
B.5. Results 40
B.6. Conclusions 43

C. Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures 46
C.1. Introduction 46
C.2. Parallelizing the CG-algorithm using OpenMP 47
C.3. Algorithmic optimizations 49
C.3.1. Cache-aware optimizations to the SpMxV 49
C.3.2. Reducing the number of global barriers 50
C.3.3. Improving load-balance 51
C.3.4. Related Work 52
C.4. Methodology 52
C.4.1. Execution environment 52
C.4.2. The FEM solver 53
C.4.3. Experimental setups 55
C.4.4. Evaluating the effect of the algorithmic optimizations 55
C.5. Experimental Results 56
C.6. Conclusions 57
C.7. Implementations in OpenMP 58

D. THROOM - Supporting POSIX Multithreaded Binaries on a Cluster **68**
D.1. Introduction 68
D.2. The THROOM concept 70
D.3. DSZOOM - a Fine-Grained SW-DSM 70
D.4. Implementing THROOM 71
D.4.1. Achieving transparency 71
D.4.2. Distributing Threads 71
D.4.3. Creating a Global Shared Address Space 71
D.4.4. Cluster-Enabled Library Calls 72
D.5. Implementation Details 72
D.6. Performance Study 73
D.7. Related work 75
D.8. Conclusions 76

Bibliography **77**

1. Introduction

Partial differential equations (PDE) appear in many fields of science and engineering. They are used to model elements of our world from matter, on the quantum level, to the motion of planets. In most real-life applications, the PDEs cannot be solved using analytical methods, instead, we have to construct a numerical approximation (solution) to the PDE. There are many ways of forming these approximations, the most common methods are the *Finite Difference* (FD) method and the *Finite Element Method* (FEM). One fundamental property of many numerical methods is that they result in sparse linear systems of equations. If we want use the methods for real applications the linear system quickly becomes very large. Also, if the PDE involves time, the systems have to be solved several thousand times to advance the approximation forward to a given time. As a consequence, we are faced with the problem of solving a very large and sparse system of equations several thousand times. To do this in a reasonable period of time, we need very powerful computing resources.

Throughout the history of computing, parallelism has been a primary tool for accelerating computations. The classical image of a parallel computer system is a system where several processors are connected together to collaborate on the same problem. Ideally, if a program takes T seconds to complete, we can solve it in $\frac{T}{p}$ seconds using p processors (the holy grail of parallel computing). We can formulate this in another way: if we just keep on adding processors the problem can be solved in almost no time at all (by letting p go to infinity). Such perfect scalability, can be achieved only if the algorithm itself is scalable (can be completely parallelized), if the programmer has managed to write an efficient program and if we can build (and afford) a computer system with a large enough number of processors with the right scalability properties. Hence, high parallel performance is an intricate inter-play between, the computer system, the parallel algorithm and the programming model used.

In this thesis we will discuss how to parallelize an iterative solver for large and sparse systems of equations. We will do this for an important class of parallel architectures, so called *shared memory architectures*. To evaluate the performance of our implementations we have used data from a real industrial application, a Computational Electro-Magnetics (CEM) solver. This solver uses the finite element method (FEM) to approximate to the Maxwell equations on an unstructured mesh. The resulting linear system is large, sparse and unstructured and the conjugate gradient (CG) method, is used for solving it.

The primary focus of this thesis is to investigate how to implement the method of conjugate gradients in way that enables us to execute the resulting parallel program efficiently on a large variety of shared memory architectures. We want to investigate which algorithmic optimizations to use and how to introduce them in a portable manner. We study load balance, data distribution and locality to find the most efficient combination of optimizations.

The rest of this thesis is structured as follows. First, we give a short introduction to the method of conjugate gradients and how to implement it efficiently on uni-processor systems, followed by a description of the input data from the CEM solver. Then we discuss how to parallelize the algorithm for shared memory architectures. To support this discussion we give a brief overview of the available shared memory architectures and programming languages and we discuss the impact of overheads introduced by the parallelization for these architectures.

1.1. Contributions

This thesis makes the following contributions:

- We show how to implement the method of conjugate gradients in an efficient manner using OpenMP.
- We show how to increase the performance of our implementation using various algorithmic optimizations for both uniform and non-uniform shared memory architectures.
- We show the importance of data distribution for non-uniform architectures for this type of application.
- We show how to achieve this data distribution in an easy to use and portable manner in OpenMP.
- To extend our results to software distributed shared memory systems (SW-DSMs), we show that it is possible to implement a POSIX programming interface in a transparent way for a fine-grained SW-DSM.

2. Solving a Large Sparse System of Equations using Conjugate Gradients

From a mathematical point of view, we are interested in solving a system of linear equations. Such systems can be solved using a standard *direct* method such as Gaussian elimination or some other method based on factorization. However, for large and sparse matrices, these methods tend to be very inefficient both in terms of the number of arithmetical operations required to compute the solution and on the amount of memory required to store the coefficient matrix. Instead *iterative* methods have been constructed that use successive approximations of the solution to form a solution from an initial guess. The choice of method and the speed of convergence of iterative methods depend greatly on the eigenvalue spectrum of the coefficient matrix. For a good introduction to the subject, see Barett et al. [3]

Many important applications such as circuit analysis, numerical boundary value problems and solutions of partial differential equations, result in large and sparse systems of equations where iterative methods can successfully be employed. The reason for this is that the sparsity of the matrices coming from the applications mentioned earlier can be exploited to save memory which enables us to solve larger problems or solving problem to a higher degree of accuracy. Direct methods usually result in *fill-in,* which basically means that we need to store temporaries to support the factorization process. Also, the process of factorization is often poorly load balanced.

In this chapter we provide the basic definitions and notation to define a standard iterative method, the method of Conjugate Gradients (CG). This method is applied in a large class of real applications, such as systems coming from finite element (FEM) discretizations, and it also resembles more general and complex methods such as GMRES. We also discuss how to implement the method of conjugate gradients on a uniprocessor system efficiently.

2.1. GEMS - An Industrial CEM Solver

GEMS is an abbreviation for General Electro-Magnetic Solver and was a three year Swedish development project funded by an extensive research program. The main objective of the GEMS project was to develop a state-of-the-art software suite for solving the Maxwell equations. For time-dependent problems, GEMS uses a hybrid

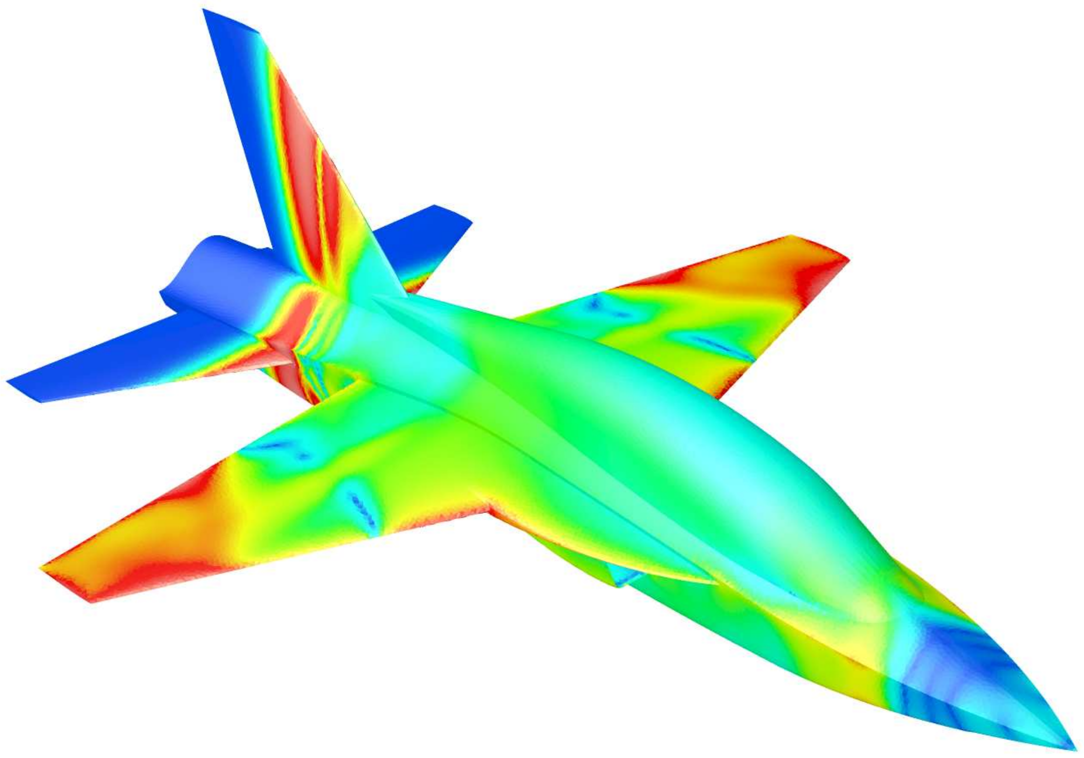


Figure 2.1.: Surface plot of the computational mesh used for the FEM discretization in GEMS

of Finite-Differences and Finite-Volumes or Finite-Elements methods for solving the equations. In a typical problem an object, such as an antenna, is placed into a domain. The program then solves the equations for an incoming wave to study the radiation or scattering of the wave. In a hybrid method, the object is discretized using a Finite-Element or Finite-Volume method and the surrounding space is discretisized using a Finite-Difference method. This approach has been shown to be very efficient since we can exploit the good characteristics of the different methods of discretization without loosing accuracy, see [24] and references therein. We are concerned with the Finite-Element part of GEMS. This solver consists of two parts. First, the Maxwell equations are discretized using the Finite-Element Method (FEM), see [24]. Second, the result of the FEM discretization, a linear system of equations is solved in each time step using an iterative solver. Throughout this thesis we use data from a discretization of a fighter-jet geometry, see Figure 2.1. This geometry results in a linear system with 1794058 unknowns.

In GEMS, the linear system is solved in each time step, which means that we can increase the performance of the overall GEMS solver by parallelizing the iterative solver part. An efficient implementation will enable users of the GEMS solver to solve more challenging and demanding problems in the area of CEM.

Algorithm 1 Method of Conjugate Gradients

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$ and set $p_0 = r_0$.
For $k = 0, 1, \dots$
(1)     Compute and Store $Ap_k$
(2)     Compute $< p_k, Ap_k >$
(3)     $$\alpha_k = \frac{< r_k, r_k >}{< p_k, Ap_k >}$$
(4)     $x_{k+1} = x_k + \alpha_k p_k$
(5)     Compute $r_{k+1} = r_k - \alpha_k Ap_k$
(6)     Compute $< r_{k+1}, r_{k+1} >$
(7)     $$\beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$$
(8)     Compute $p_{k+1} = r_{k+1} + \beta_k p_k$
```

2.2. The Mathematical Problem

We want to solve the linear system of equations $Ax = b$, where A is called the *coefficient matrix*, b is the *right-hand side vector* and x the solution of the problem. Formally, the solution x is given by $x = A^{-1}b$. The goal of any iterative method is to produce an approximation to the solution within a given tolerance in as few iterations or operations as possible. In most cases, lowering the amount of iterations is by far the most efficient way of achieving high performance. The most common technique is to use a *pre-conditioner* to transform the system to a system with better convergence properties, where the solution to the original problem can be calculated with little or no effort from the preconditioned system. For the actual problems studied here, the convergence rate is already high. Hence, the use of a pre-conditioner is not very well motivated in our case as we will see later.

2.2.1. The Method of Conjugate Gradients

Algorithm 1, shows a standard formulation of the CG method. The mathematical properties and the actual derivation of the algorithm can be found in many textbooks on numerical linear algebra. It can be shown, see [3] that the CG method converges to the best possible approximation if the coefficient matrix is *positive definite* and symmetric.

2.3. Implementing Conjugate Gradients

Implementing an algorithm such as the method of conjugate gradients involves the process of stating the mathematical operations needed in some programming language.

We also need to identify the core data structures needed to represent the matrix and the vectors. If the matrix is dense the choice of data structure becomes trivial as a matrix of floating point numbers is very well represented in most programming languages. If the matrix is sparse, it does not exist a natural representation and the choice of data structure becomes more difficult.

To achieve maximal performance of the target computer platform, the implementations of the operations in the algorithm need to be very efficient. To ensure this, we can in the dense case, use standardized, precompiled libraries built by the system vendor to maximize performance[7]. However, if the matrix is sparse, there is no general method of attaining high performance, compared to the dense case. Hence, we discuss some methods of improving the performance of the sparse matrix-vector product, which is the only operation influenced by the data structure needed to represent the sparse matrix.

In later chapters, we will discuss how to parallelize the algorithm on shared memory systems, which is the primary focus of this thesis. However, the work put into optimizing a serial implementation can be reused to provide an efficient parallel implementation as we still need to use the target microprocessor in an efficient way.

To be able to use the CG method in practise, we need to add ways of detecting convergence and divergence. Divergence can be found by simply counting the number of iterations and aborting when the iteration count has exceeded a certain divergence threshold. Accurate predictions on the convergence behavior and error are difficult to make and usually we want to iterate until the solution $x^{(k)}$is close to the real solution x in some norm. This means that we want to stop iterating when the error $\parallel e^{(k)} \parallel = \parallel x^{(k)} - x \parallel < \varepsilon$, where ε is a user prescribed tolerance. Instead of approximating $e^{(k)}$ directly, which is very costly to compute, we can use the *residual* $r^{(k)} = Ax^{(k)} - b$ instead. In terms of programming, the stopping criteria adds conditional statements in the main loop if the CG iteration.

2.3.1. Exploiting Sparsity

Many applications, especially PDE solvers, result in sparse matrices. For PDE solvers, it is not unusual that the dimension of the matrix is in the order of millions or even billion. In these cases it is not realistic to store the entire matrix since it would require memories of petabyte capacities. It is not necessary either, since the numerical methods used for discretizing the PDE have a strong local behavior. Only a few neighboring points in the mesh influence, which give rise to very sparse matrices. We can exploit this sparsity and store only the non-zero elements. This makes it possible to solve very large problems using iterative methods since we save both memory and unnecessary operations on zeros.

There are many formats for storing sparse matrices, [57]. They differ in the sense that some other structural property, such as banded or blocked matrices, of the matrix can be exploited[30]. In this study we have used the Sparse Compressed Row (CSR) format. This format is a well-known and established format for unstructured sparse matrices. The CSR format consist of three arrays, `VAL`, `COL_IND` and `ROW_PTR`, where `COL_IND` stores the column indeces of each non-zero, and `ROW_PTR` points to the beginning of each row in the `VAL` and `COL_IND` arrays.

An example: the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 0 \\ 8 & 0 & 9 & 10 & 11 & 0 \\ 0 & 13 & 0 & 0 & 14 & 15 \\ 0 & 0 & 16 & 0 & 17 & 18 \end{pmatrix}$$

is stored in the following way using the CSR format:

`VAL`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`COL_IND`	0	3	5	0	1	1	2	5	0	3	4	5	1	4	5

and

`ROW_PTR`	0	3	5	8	12	14

Sparse matrices can also be interpreted as the *adjacency matrix* of an associated graph. An adjacency matrix is a matrix where rows and columns are labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether v_i or v_j are adjacent of not. In our case we modify this definition slightly by saying that any number different from zero states that two nodes are adjacent in the graph. This property enables us to apply graph theoretical results for a number of useful things such as minimizing the bandwidth of matrix or partitioning (load balancing) the matrix, see Section 2.4.1 and Chapter 4.

2.3.2. Implementing Conjugate Gradients for Sparse Systems of Equations

We first need to decide on a format for storing the sparse matrix. This decision depend on many things, such as the structure of the matrix and the target computing platform. As described earlier we choose the CSR format since it is general and has been proven to give relatively good performance for a large set of matrix structures. Also, many software libraries use this format.

In the sparse matrix case, the only difference from a data structure point of view is how we represent the matrix. The vector operations are still dense and we can use standard dense operations. For the matrix vector product we need to replace the dense operation with its sparse counter-part.

The sparse BLAS standard is relatively new and there exist very few implementations by vendors compared to the dense case[22]. How to optimize sparse BLAS is still an open research issue. Although many techniques developed for dense matrices can be used in the sparse case, we investigate additional optimizations techniques to improve the performance of a CG implementation.

Algorithm 2 OpenMP implementation in C of a sparse matrix vector product for the compressed sparse row (CSR) format.

```
void spmxv(const double *val, const int *col,
            const int *row, const double *x,
            double *v, int nrows)
{
   register int i,j;
   register double d0;

#pragma omp for private(i,j,d0) nowait
   for( i = 0; i < nrows; i++ )
     {
       d0 = 0.0;
       for( j = row[i]; j < row[i+1]; j++ )
         d0 += val[j] * x[ col[j] ];
       v[i] = d0;
     }
}
```

2.4. Optimizing Conjugate Gradients for Sparse Systems of Equations

The performance of the CG algorithm in the sparse case is dominated mainly by two things: the convergence rate, i.e how many iterations are performed and the efficiency of implementation of the sparse matrix-vector product. Accelerating the convergence is very problem dependent it is a subject out of scope of this thesis. More on this subject can be found in [3].

2.4.1. Optimizing the Sparse Matrix-Vector Product

The SpMxV have several performance problems on modern microprocessors. First of all, this operation is memory-bound, since each element in the result vector `v` require more memory operations than floating-point operations, see Algorithm 2. The inner loop can be very short and varies in size between the rows. Also, the spatial locality in the right-hand side (RHS) vector `x` is very bad due to the indirect addressing through the column index vector `col`. The spatial locality for the row and column vectors are however good since they exhibit perfect stride-1 access patterns. The distributions of elements of each row or column of the matrix determines the amount of reuse present in the cache blocks of the RHS `x`. In the case of FEM, this pattern is determined by the type and numbering of elements and also on the geometry of the problem. Furthermore, the number of non-zero elements in each row or column is bound by how many elements that geometrically couple in physical space. For 3-dimensional problems, a common number of non-zeros per row or column is about 20. This small number of elements and the irregularity in loop length makes it hard for an optimizing compiler to use standard optimizations such as blocking, unrolling and tiling on the inner-loop efficiently.

Using the fact that any sparse matrix can be interpreted as an adjacency matrix of a corresponding graph, we can apply graph theoretical methods to improve the locality of the SpMxV. Remember that the spatial locality of the RHS accesses in the SpMxV will probably increase if the matrix contain large dense blocks since we can reuse more elements of the cached block. Unfortunally, many applications does not exhibit a large dense blocks, but the matrix can be re-ordered to maximize the number of such blocks. It has been shown by Pinar et al [54] that this problem can be formulated as a traveling salesman problem (TSP). Hence, heuristics developed for TSP can be employed to find dense blocks in polynomial time.

Another way of increasing the amount of dense blocks is to minimize the bandwidth of the matrix [19, 28]. This has been successfully employed in several previous studies see, [52, 11, 69]. More lower level optimization methods resembling the methods used to implement dense BLAS have also been studied by several authors: register blocking to lower the amount of load instructions [54, 71, 69], address precomputation [69] and cache blocking [71].

3. An Introduction to Shared Memory Architectures

Shared memory architectures are usually described as the logical continuation of a single-processor architecture. In a shared memory architecture all processing elements have direct access to a single shared memory. These architectures have been well studied in the literature[32, 18, 30, 21].

3.1. The Cache Coherency Problem

Most modern computer systems, use *cache memories* to reduce the latency and increase the bandwidth to the memory system. On a shared memory multi-processor system, shared data will be replicated in cache memories of several processors. If a processor modifies cached data, this will lead to in-consistencies as the cached copy is different from the contents of the main memory and other cached copies of this data. This problem is referred to as the *cache coherency problem*. Some shared memory architectures, such as the Cray T3E [62], does not enforce coherency among its cache memories. In these so called non-coherent shared memory systems, communication between processors must go through the shared memory only. Locality for shared data can in this case not be fully exploited. In cache-coherent shared memory architectures, which is the most common type, this locality is exploited and the cache coherency problem is solved in a transparent way in hardware or software, see [32, 18]. To construct a well balanced and scalable cache-coherent architecture several difficult design tradeoffs have to be made. This fact has led the industry to construct two basic types of cache-coherent architectures targeted at different segments of the commercial market:

Uniform shared memory architectures (UMA) In these architectures, sometimes referred to as Symmetrical Multi-Processors (SMP), the access time to the shared memory is uniform. This is the most common form of shared memory architectures on the market today. Due to its centralized design these architectures are constrained by the memory demands of the processors. Hence, they usually exhibit processor counts of a few dozen.

Non-uniform shared memory architectures (NUMA) In these architectures, the memory is physically distributed over a set of *nodes*. As a consequence, the access time the the logically shared memory is non-uniform. These architectures are constructed to support the bandwidth demands of a larger amount of processors without incurring excessively long access times. Sometimes non-uniform systems are referred to as distributed shared memory systems (DSM)

In a non-uniform architecture, the access time for memory within a node is smaller than the time required for a remote access. The difference is often measured by the *NUMA-ratio*, see [56].

$$\text{NUMA-ratio} = \frac{\text{Remote access time}}{\text{Local access time}} \tag{3.1}$$

Also, in modern microprocessors the locality of many programs is exploited by introducing an hierarchy of cache memories. On non-uniform architectures an additional form of locality has also to be taken into account. The *geographical locality* of data governs the amount of remote accesses issued to the memory system. An application where most accesses are served by local memory is said to exhibit good geographical locality.

3.2. Software Distributed Shared Memory Systems

Most shared memory architectures solve the cache coherency problem by using specialized hardware. The complexity of this solution often result in very expensive systems. To reduce the cost of shared memory systems, there has been a significant amount of research aimed at solving the coherency problem in software. The idea is to use high volume commodity products and assemble a *computing cluster*. A computing cluster is a system of loosely connected computers called nodes, using standard networking technology (cables and switches). The nodes of a cluster could be any type of computer, parallel or sequential. Coherency is then managed by a software layer between the application and the hardware. Implementing the coherency in software allows for the use of commodity components while still providing a shared memory abstraction. In this way we can dramatically decrease the cost of cache-coherent shared memory architectures. These systems are categorized as non-uniform architectures and they are often referred to as *software distributed shared memory systems* (SW-DSM) or *shared virtual memory systems* (SVM). Whether this can be done with any performance comparable to an all hardware solution is still an open research issue, see [58, 23, 39, 65, 55].

3.3. Examples of Shared Memory Architectures

Here the shared memory architectures used in the thesis are described and compared. The machines represent typical shared memory machines available from most vendors although in our case all machines were delivered by Sun Microsystems. The architectural parameters of these systems are summarized in Table 3.2 and Table 3.1.

The Sun Enterprise 6000 (E6000), introduced in the mid nineties, represent a typical UMA or bus-based shared memory architecture, see [18]. This system represents the vast majority of shared memory architectures available today. Systems of this kind normally exhibit processor counts from two to sixteen.

To investigate possible ways of increasing the scalability of E6000-like systems, Sun constructed a system called the Sun WildFire [32, 31]. The main idea of this system is similar to that of a SW-DSM although in this case coherency was maintained in hardware. The WildFire system is constructed by connecting upto four E6000s using a special network hardware. In this way, a cluster of SMP:s can be assembled to increase the scalability of a single E6000 almost four times. To keep the same cache-coherent view of memory as a single E6000 node, the interconnect hardware supports a global and coherent shared memory. However, the interconnect hardware exhibits lower bandwidth and higher latency compared to the shared memory bus of a single node. As a consequence, the access time to memory in a remote node increased making the WildFire a non-uniform architecture.

The Sun Enterprise 10000 (E10K) [14] is one of the most scalable UMA architectures ever built. Using components from the Sun Enterprise models, this architecture can have 64 UltraSPARC-II CPUs all with a uniform access time to memory. To achieve this the E10K have four multiplexed address buses to support the CPUs with enough bandwidth to cope with the traffic generated by the protocol used to enforce cache coherency.

To follow up the very popular E10K, Sun introduced the Sun Fire 15000 (SF15K) in 2001 [15]. This system was one of the first commercial non-uniform systems Sun built and it can contain 18 nodes. Every node, or CPU board, consists of four UltraSPARC-III CPUs, DRAM memory and a local bus. Locally each node behaves like a small SMP. The nodes are connected to a large crossbar switch supporting a global, coherent, shared but physically distributed memory. Other architectures of this type include the SGI Origin series [41, 63] and the AlphaServer GS series [27]

3.3.1. DSZOOM, a Fine-grain SW-DSM System

In a *page-based* SW-DSM the virtual memory subsystem of the operative system is used as a mechanism to detect when the coherence protocol needs to be invoked. This design has the disadvantage that the granularity of sharing is of the size of a page which is very large compared to the size of a cache block. Another mechanism is to modify the binary code of an application and insert code fragments to detect coherence violations. These SW-DSMs are called fine-grained SW-DSMs [59, 61, 60, 55]. This mechanism has the advantage that we can keep the coherence unit at the size of a cache block. In this thesis, we have used the DSZOOM fine-grained SW-DSM system [55] running on the Sun WildFire.

	System	Frequency	L1 cache	L2 cache
US-II	E6000, WildFire	250Mhz	16KB	4MB
US-II	E10K	400Mhz	16KB	8MB
US-III	SF15K	900Mhz	64KB	8MB

Table 3.1.: Comparison of the micro-processors used in this thesis

System	CPU	Nodes	CPUs/node	Max CPUs	Local	Remote	NUMA-ratio
E6000	US-II	1	14	30	330ns	330ns	1.0
E10000	US-II+	1	32	64	600ns	600ns	1.0
WildFire	US-II	2	14	112	330ns	1700ns	6.0
SF15K	US-III	8	4	106	200ns	400ns	2.0
DSZOOM	any	any	any	any	low	high	high

Table 3.2.: Comparison of architectural characteristics of the shared memory systems used in this thesis

3.4. Future Shared Memory Architectures

Recent trends in microprocessor architecture indicate that future shared memory systems will be hosted in a single chip [2]. Chip Multiprocessors (CMP) [4, 32] are constructed by placing several microprocessors (cores) onto a single chip and connecting them using an on-chip network. The cores can also be capable of executing code from several programs simultaneously, a technique called Simultaneous Multi-Threading (SMT) [25, 32]. Combining these techniques will produce shared memory architectures with drastically different characteristics of the memory systems since the cores can communicate on-chip. Whether these systems will be uniform or non-uniform is not clear today. However, several CMP/SMT chips can be connected together using interconnect hardware to form a large shared memory architecture. Coherence can in this case be managed in software or hardware depending on the price/performance ratio of the final system.

CMP/SMT architectures will probably increase the general availability of shared memory systems. In just a few years, every desktop computer will probably be able to execute at least two threads in parallel. Hence, we belive that there will be an increased demand for parallel programs. In the light of these observations, the shared memory programming model seems like an attractive abstraction for producing future parallel programs.

4. Parallelizing Conjugate Gradients for Shared Memory Architectures

Parallelizing a sequential program and producing an efficient implementation is a non-trivial task. The parallelization process can, following Culler et al. [18], be divided into four different steps

1. *Decomposition* of the computation into tasks
2. *Assignment* of tasks to threads
3. *Orchestration* of the necessary synchronization among the threads
4. *Mapping* or binding of threads to processors

In this chapter we will discuss and describe how to perform these four steps in the shared memory paradigm. The primary concern of this thesis is to investigate if it is possible to implement the CG algorithm in way so that the program can be executed efficiently on a wide range of shared memory architectures. To investigate this we continue the overview of the shared memory architectures with an introduction on how to program them. Then we decompose the algorithm into parallel tasks (decomposition) and discuss the various parallel overheads associated with load balance (assignment), false and true sharing (orchestration), and synchronization (orchestration). This discussion will serve as a background to chapter 5 and to the work presented in papers A, B and C.

We choose to use OpenMP, which is a standard for programming in the shared memory model, for our implementation. In papers A, B and C we found that we needed to introduce algorithmic optimizations in order to attain high performance on both uniform and non-uniform shared memory architectures. In these papers, we also show the importance of proper data distribution and how to achieve this in a way that is easy to use and portable to many current and future non-uniform shared memory architectures, see Chapter 5.

4.1. An Introduction to Shared Memory Programming

In the shared memory programming model, all processors can access a global shared memory. The fundamental abstraction in this model is the concept of a *thread*. When a program is loaded, the OS sets up an address space of the application containing its program code and data. To run the program, the OS loads the start address of the program into the program counter (PC) of a processor. Now, since the address space is shared and we have several processors available, we can initiate another *thread of control* by loading the program counter of another processor. These two threads can now execute in parallel and communicate by reading and writing into the shared memory.

From a programmers point of view, a new thread is created by calling a system library function and giving the address of a function where the new threads shall begin its execution as a parameter. A parallel computation is obtained if several threads are created in this manner each operating on different data. Although the address space is kept coherent we often need to synchronize the threads of an application to ensure the integrity of the data. This is a characteristic feature of the shared memory programming model, see [30, 21].

4.1.1. OpenMP

There are several standardized ways of programming in the shared memory model. The POSIX [34] standard provides a standardized interface for programming in the shared memory model. In most cases, the POSIX standard often become too difficult or cumbersome to use. To make it easier for the programmer to handle synchronization and creation of threads, a consortium of most vendors designed the OpenMP language [53, 20, 47].

In this language a master thread executes until it encounters a parallel region. At this point several threads are created and the work contained in the parallel region is divided onto the threads. This style of programming is often referred to as *fork/join* programming. The most simple form of work-sharing is a loop. If the iterations of the loop are independent, the work can be divided by assigning disjunct sets of the loop indeces to different threads.

In OpenMP, the parallel regions are identified by inserting a directive or compiler pragma into the serial code. This method has the advantage that the code can be incrementally parallelized by adding directives to the code. Also, the directives can be ignored by the use of a compiler flag resulting in a serial program.

4.1.2. Programming SW-DSMs

In contrast to a HW-DSM system (cc-NUMA), where the whole address space of all processes are kept coherent by hardware, most SW-DSMs only keep coherence for specified segments in the user-level part of the virtual address space. This segment, which we call G_MEM, is mapped shared across the DSM nodes using the interconnect hardware. Furthermore, the text (program code), data and stack segments of the UNIX process abstraction are private to the parent process and its children on each node of the cluster. This creates a SW-DSM programming model where special constructs are needed to separate shared data, which must be allocated in G_MEM, from private data, which is allocated in the data and stack segments of the UNIX process at program loading. This is often done by creating a separate heap space in G_MEM with an associated primitive for doing allocation. In a standard multi-threaded world, there exist only one process and one address space which is shared among all threads. There is no distinction between shared and private data. Consider the following example: An application allocates a shared global array for its threads to operate on. This is often done by a single thread in an initialization phase. In a typical SW-DSM system such as TreadMarks [39], a special `malloc()`-type call has to be implemented to allocate the memory for the shared array inside the G_MEM. Also, the pointer variable holding the address, which is allocated in the static data segment of the process, has to be propagated to all remote nodes. This is often done by introducing a special propagation primitive.

In paper D, we present THROOM which is a runtime system concept that creates the illusion of a single process shared memory abstraction on a cluster. In essence, we want to make the static data and heap segments globally accessible by threads executing in remote nodes without introducing special DSM constructs in the application code. In the light of the example above, the application should use a standard `malloc()` call and the pointer variable should be replicated automatically.

The results from paper D, show that it is possible to share the address space in a transparent way using binary instrumentation. However, due to the high instrumentation overhead incurred by the DSZOOM SW-DSM the performance of this approach was not very encouraging, even if we excluded the stack from the G_MEM.

4.2. Identifying Parallelism in the CG algorithm

We have already identified the fundamental operations needed to implement the CG algorithm in chapter 2. To identify parallelism we can decompose each of the operations into tasks as follows

Vector operations (Lines (4),(5) and (8) of Algorithm 1) Here, each element of the resulting vector is an independent task.

Inner products (Lines (2) and (6) of Algorithm 1) This operation corresponds to a shared memory reduction. In this case the computing the local contribution is an independent task.

Sparse matrix-vector product (SpMxV) (Line (1) of Algorithm 1) Here, a task T_i corresponds to an element of the result vector which is computed by forming the inner product of row i and the right-hand side vector.

We can also identify phases of computation as the threads needs to be synchronized at the two inner products to compute the scalars α and β needed to compute the vectors.

To parallelize the CG algorithm in a POSIX like environment you need to distribute the tasks over the threads. Each set of tasks is called a partition of the data and corresponds to a set of indices of the vectors or a collection of rows in the matrix. If the matrix and vector are of dimension n, a static partition would assign chunks of tasks of size $\frac{n}{p}$ to each thread. In terms of programming, each partition correspond to a set of indices in the loops needed to implement the required operations. The partitions need not to be the static during the iterations. We can use different partitions for the operations as long as we keep the threads synchronized between the different stages of the algorithm. This can be useful to achieve better load balance in the SpMxV.

4.2.1. An Analysis of the Parallel Overhead

The main sources of parallel overhead in any shared memory implementation are due to synchronization, true/false sharing effects and arithmetical load balance. If the assignment of tasks is unbalanced most threads will wait for the slowest thread in the barriers implied by the reductions.

In the case of the CG algorithm, false sharing can occur if two threads are working on neighboring tasks, where it is possible to have multiple threads writing at the same cache block. If the partitions collect tasks in an ordered fashion neighboring tasks will only exist at the partition boundaries. Also, when performing the reduction the data structure used to collect the local reductions must be padded to avoid false sharing. Synchronization overheads will be present in the barriers needed to ensure that the vectors are computed in the correct order given by the algorithm. The number of barriers should be kept as low as possible while maintaining a correct program flow.

Finally, load balancing is good for the vector operations and the reductions if they are statically partitioned. The load balance in the SpMxV is however dependent on the structure of the sparse matrix, i.e how many non-zeros elements there are associated with each task. This is determined by the non-zeros structure of the matrix. Hence, we might want to use some kind of data dependent partitioning in the SpMxV to produce partitions with an equal number of non-zeros entries (not just an equal number of rows).

4.2.2. Implementing Conjugate Gradients using OpenMP

Implementing the CG algorithm is fairly straight-forward from a sequential implementation. We simply add work-sharing directives to the loops that implement the operations of the algorithm. We decided to include the entire main iteration into a single parallel region to reduce the overheads of entering and leaving parallel regions. In OpenMP the task of producing partitions is built into the language in the form of scheduling clauses. We can, by using different scheduling clauses of the work-sharing directives produce different partitions. In our case this was not needed since we used more sophisticated methods of reordering the matrix and partitioning the tasks, see Paper C.

4.3. Using Algorithmic Optimization to Improve Performance

In Paper C we evaluate different algorithmic optimizations to increase the performance of the solver. Most of the optimizations used are well known but they have, in most cases, only been evaluated in a message passing environment. The algorithmic optimizations studied where: *graph partitioning*, *S-Step formulation of the algorithm* and *bandwidth minimization*. Apart from the algorithmic optimizations we also worked extensively with the implementation in OpenMP and managed to reduce the number of global barrier from seven to three per iteration. To evaluate the efficiency of our implementation and the effect of the algorithmic optimizations, we executed the code on both a uniform (an E10K) and a non-uniform shared memory architecture (a SF15K) using upto 28 threads.

Results show that the matrix resulting from the FEM discretization was poorly load balanced. Both graph partitioning and bandwidth minimization improved the load balance. In fact, bandwidth minimization together with a static partitioner proved to give the best load balance. This was also quantified by measuring the time spent executing in barriers. The improved load balance also improved to overall runtime of the solver for the uniform architecture. On the non-uniform architecture, the reordering produced by the graph partitioner showed to increase the number of remote accesses compared to the standard formulation. Hence, graph partitioning showed to give little or no performance improvement on the non-uniform architecture.

Implementing the CG algorithm in an S-step formulation removes yet another barrier to a total of two barriers per iteration. This formulation introduces an extra vector which increased the amount of work compared to the standard formulation. Reducing the number of barriers did not show to increase the performance of the solver substantially.

Bandwidth minimization exhibited the best performance on both the uniform and non-uniform architectures. Apart from exhibiting very good load balance the lower bandwidth of the matrix gave rise to fewer cache misses in the second level cache. Also, the number of remote accesses was reduced compared to the original algorithm. Combining bandwidth minimization with graph partitioning and the S-step formulation did not increase the performance compared to using bandwidth minimization only.

Previous studies has shown that bandwidth minimization is an efficient technique for accelerating iterative solvers on shared memory systems, see Oliker et al. [52]. In this paper the authors studied a shared memory implementation of the CG algorithm on a SGI Origin system. Our work differs in the sense that we use data from a real application and we concentrate on investigating what mechanisms limit the performance on a larger set of shared memory architectures. In [52] the authors focus on the performance aspects of using different programming paradigms and parallel architectures for solving a large system of linear equation.

On the non-uniform architecture we used techniques to increase the geographical locality of the solver developed in papers A and B, see Chapter 5. Without these techniques the performance was very poor on the non-uniform architecture. In fact, the parallel performance on 28 threads was slower than the uniform architecture although the code exhibited a factor of 5 performance improvement running on a single thread. Using the techniques developed in papers A, B and C, the performance on the non-uniform architecture increased and we managed to achieve a speedup of a factor of 3 when comparing the uniform and non-uniform system on 28 threads.

5. Optimizing for Non-Uniform Shared Memory Architectures

On non-uniform architectures we have to take the concept of geographical locality into account. The amount of geographical locality present in an application, i.e how well threads are co-located with data on the DSM nodes is dependent on many things. The application need to interact with the mechanisms available in the DSM architecture or OS to improve geographical locality. To achieve this we must know how data gets distributed over the nodes and how threads are scheduled. Using this knowledge we might rewrite code in the application or use system calls to increase the geographical locality. However, such optimizations tend to be non-portable across a large set of DSM architectures.

Another possibility of increasing geographical locality is to let the system adapt to the application and migrate or replicate data to increase the amount of local accesses. In this case, the application code could be left unmodified which will increase the level of portability. The performance will on the other hand depend on how efficient the system can adapt to the access pattern of the application.

5.1. Architectural Optimizations on DSM Systems

There has been many attempts to introduce various features in the systems to improve the geographical locality of applications. These features are presented and discussed below.

5.1.1. Thread Scheduling on DSM Systems

When a new thread is created it gets scheduled onto some of the nodes of the DSM architecture. On a time-sharing OS threads might also be moved to some other node to balance the workload since each node has a finite capacity in terms of memory and processors. On a DSM system it is normally beneficial to include the notion of affinity in the scheduling algorithm. Affinity scheduling means that the thread should be scheduled to the same processor or node as it has been executing on previously. This increases the possibility of reuse of old data stored in caches and node memory, see [70, 12] and references therein.

5.1.2. Page Placement Policies on DSM Systems

As with thread scheduling, the OS also needs to decide on how to distribute memory over the nodes. This introduces the notion of *page placement policy*. The most common page placement policy is the *first-touch* policy, where pages are placed on the node where a thread first references the page (generates a page fault). Other policies include *random placement* where pages are distributed randomly in a uniform fashion and *round-robin*, where pages are placed in a modulo-*n* fashion. Previous studies have shown that the first-touch strategy is an efficient strategy for typical HPC workloads [49]. Also, first-touch placement has the potential to work very well when single-threaded programs are executed compared to the random placement policy.

If the first-touch strategy is used, affinity scheduling becomes important since not only cached data could be reused. If threads are moved from one node to another, the amount of remote accesses will increase since data is physically allocated to memory in the node where the thread first touches it. The optimal schedule would be to keep the thread in the node where it was first scheduled. Some systems also include ways of controlling the placement of pages. In these systems, a programmer can explicitly distribute pages to better match the access pattern of the application. For these placements to be efficient, the scheduler also needs to match the placement of pages.

5.1.3. Page Migration on DSM Systems

In many cases, an application exhibits phases where the memory access pattern changes dramatically. As an example consider the case of initialization. In many applications, a single thread initializes the data before more threads are created. On a system with first-touch page placement strategy this will result in all memory being allocated to the node where the initialization thread executes. In the parallel phase, threads not scheduled to this node will generate a large amount of remote accesses. This problem can sometimes be solved by initializing the data in parallel phase and ensure that threads are not moved. However, for more complex codes, it can be very hard or even impossible to initialize data in parallel.

A way of adapting to changes in the access pattern is to try to automatically migrate pages to minimize the amount of remote accesses, i.e move the data to the threads. Such mechanisms can be constructed in user-level, using explicit page placement calls [49] or by augmenting the operative system with dynamic data distribution features [12, 41, 31]. Page migration has shown to be very efficient for some scientific applications [33, 49, 10]. One problem with automatic page migration is that the system could easily introduce overheads when deciding which pages to migrate and when. Interrupting the application to often might lower the efficiency, also migrating pages consumes bandwidth of the interconnect network [6].

Page migration can also be initiated from the application. This mechanism has the advantage that the programmer can control when to migrate to adapt to changes in the application. A recent method of implementing this behavior is to include a call with a *migrate-on-next-touch* semantic, see [51, 6]. Using this system call a programmer can redo a placement using the first-touch policy again.

Many of the features described above have been implemented in commercial systems. The SGI Origin systems [41] was one of the first systems of this type on the market. It supported a first-touch placement policy as well as a dynamic page migration mechanism. This system also featured system calls for performing explicit page placement. The efficiency of the migration feature was not very high due to the schedulers inability to achieve high affinity to the data, see [17]. The HP AlphaServer GS Series [27, 6] support mechanisms for explicit page placement as well as a migrate-on-next-touch call. The Sun WildFire [31, 32] prototype supports a dynamic page migration and page replication engine which has been proven to work very well for many applications, [31, 51, 33, 10]. The Sun Fire series of servers [15] support a choice of random or first-touch placement policy and a migrate-on-next-touch call [66].

5.2. Improving the Geographical Locality of the CG Implementation

In paper A we investigate the importance of geographical locality for a small set of scientific applications. We evaluated the use of parallel initialization, migrate-on-next-touch and dynamic page migration to overcome the problem of serial initialization on system using a first-touch page placement policy. In the case of the CG algorithm, which was my contribution to this study, parallel initialization is not possible due to the indirect addressing of the CSR data structures. In this case, migrate-on-next-touch and a the dynamic page migration engine of the Sun WildFire managed to increase the amount of geographical locality significantly. In this study we also found that the effect of page migration differ somewhat when comparing codes from benchmarks and real applications.

In paper B we investigated the scalability of the migrate-on-next-touch primitive. We found that the Solaris implementation was not very scalable due to overheads associated with keeping TLBs coherent. By using large pages and a migrate-on-next-touch directive, we managed to improve the performance of the CG solver with up to 160% compared to the original solver where all data was allocated onto a single node due to the first-touch page placement policy.

In paper C we compare the scalability of the CG solver on a uniform architecture to the scalability achieved on a non-uniform architecture using the optimizations developed in paper B. On the uniform architecture we achieved a speedup of 30 using 28 threads and bandwidth minimization. On the non-uniform architecture we got a speedup of about 15 using 28 threads. Possible reasons for the difference in scalability could be that in the non-uniform case, communication is more expensive since some of these accesses are served by remote nodes. Also since a first-touch strategy is used, sequential execution will exploit the low latencies of node local accesses. Another possible reason for this difference could be attributed to NUMA-effects in the implementations of the synchronization primitives in the OpenMP runtime system, see [26].

5.3. Data Distribution in OpenMP

In the OpenMP community there has been a lot of discussion if it is necessary to expose the non-uniformity of the architecture in the language [47]. The primary concern from the OpenMP steering group has been portability. In paper B we propose to add a directive with an *affinity-on-next-touch* semantics. The functionality is that of a migrate-on-next-touch directive but to give it a more general name we choose to include the concept of affinity. We believe that such a directive is portable and usable on many current and future shared memory systems.

6. Conclusions

We have implemented and parallelized the method of conjugate gradients in the shared memory programming paradigm in OpenMP. Using data from an industrial CEM solver we found that, in order to obtain high performance and efficiency for both uniform and non-uniform architectures, we need to apply various algorithmic optimizations. Of the optimizations studied, bandwidth minimization of the sparse matrix showed to give the highest performance. The primary reason for this is that the reordering produced by the bandwidth minimization algorithm increased the spatial and temporal locality in implementation of the sparse-matrix vector product. This increased locality also lowered the amount of remote accesses on the non-uniform architectures studied. Furthermore, using a simple linear partitioner in combination with bandwidth minimization produced a better arithmetical load balance than a more sophisticated method such as graph partitioning.

We also found that on non-uniform architectures proper data distribution is needed in addition to the algorithmic optimizations studied. Data distributions can either be produced in a transparent way by the computer system using page migration or by explicit page placement directives in the programming language. In our studies both methods showed to work well for this application. We also showed that the overhead introduced by page migration in the operative system studied was dominated by mechanisms used for keeping translation look-aside buffers (TLBs) coherent and not from the copying of data. As a consequence, we could lower this overhead using larger pages.

Finally, we found that a primitive with an affinity-on-next-touch semantic, was as efficient as more complex methods of creating data distributions such as dynamic page migration. We also believe that such a primitive could be added to a language like OpenMP without affecting portability and ease of use. In our experiments we show that, adding a single affinity-on-next-touch primitive could increase the performance of our implementation with upto 160% on a non-uniform architecture where the ratio of remote to local latency was as low as a factor of two.

7. Future Work

We believe that SW-DSM systems will become more important in the future. Hence, we would like to study the effect of the algorithmic optimizations studied on such systems. However, to be able to do this we need either to port our implementation to a programming model supported by the SW-DSM system, or we can try to add features to the SW-DSM system to enable it to use a more standard model such as POSIX or OpenMP. The results from Paper D showed that supporting the POSIX programming model was very expensive due to the overheads associated with adding software coherence to the entire address space. However, in a language like OpenMP, shared data is explicitly stated in the language. Hence, we could lower these overheads by just adding coherence to the shared part of the address space. Recently, there has been some activity to produce an open compiler infrastructure for OpenMP. These initiatives could lower the amount of work needed to experiment with SW-DSMs and OpenMP. It would be interesting to study if we can use these initiatives to produce an OpenMP interface to the DSZOOM SW-DSM system. Having produced such an interface, we could study the performance of our implementation as well as other important OpenMP applications on SW-DSM systems.

We would also like to study the applicability of the affinity-on-next-touch primitive for a larger set of applications, such as the SpecOMP benchmark suite. It would also be interesting to study the effect of this primitive for applications where the access pattern changes during the execution such as adaptive mesh refinement and molecular dynamics simulations.

Paper A

A. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers

Henrik Löf, Markus Nordén and Sverker Holmgren
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
henrik.lof,markus.norden,sverker.holmgren)@it.uu.se

Abstract

On cc-NUMA multi-processors, the non-uniformity of main memory latencies motivates the need for co-location of threads and data. We call this special form of data locality, geographical locality, *as one aspect of the non-uniformity is the physical distance between the cc-NUMA nodes. We compare the well established first-touch strategy to an application-initiated page migration strategy as means of increasing the geographical locality for a set of important scientific applications.*

The main conclusions of the study are: (1) that geographical locality is important for the performance of the applications, (2) that application-initiated migration outperforms the first-touch scheme in almost all cases, and in some cases even results in performance which is close to what is obtained if all threads and data are allocated on a single node.

A.1. Introduction

In modern computer systems, *temporal* and *spatial locality* of data accesses is exploited by introducing a memory hierarchy with several levels of cache memories. For large multiprocessor servers, an additional form of locality also has to be taken into account. Such systems are often built as cache-coherent, non-uniform memory access (cc-NUMA) architectures, where the main memory is physically, or *geographically* distributed over several multi-processor nodes. The access time for local memory is smaller than the time required to access remote memory, and the *geographical locality* of the data influences the performance of applications.

The NUMA-ratio is defined as the ratio of the latencies for remote to local memory. Currently, the NUMA-ratio for the commonly used large cc-NUMA servers ranges from 2 to 6. If the NUMA-ratio is large, improving the geographical locality may lead to large performance improvements. This has been recognized by many researchers, and the study of geographical placement of data in cc-NUMA systems is an active research area, see e.g. [51, 6, 49, 10].

In this paper we examine how different data placement schemes affect the performance of two important classes of parallel codes from large-scale scientific computing. The main issues considered are:

- What impact does geographical locality have on the performance for the type of algorithms studied?
- How does the performance of an application-initiated data migration strategy based on a migrate-on-next-touch feature compare to that of standard data placement schemes?

Most experiments presented in this paper are performed using a Sun Fire 15000 (SF15k) system, which is a commercial cc-NUMA computer. Some experiments are also performed using a Sun WildFire prototype system [31].

Algorithms with static data access patterns can achieve good geographical locality by carefully allocating the data at the nodes where it is accessed. The standard technique for creating geographical locality is based on static first-touch page allocation implemented in the operating system. In a first-touch scheme, a memory page is placed at the node where its first page fault is generated. However, the first-touch scheme also has some well known problems. In most cases, the introduction of pre-iteration loops in the application code is necessary to avoid serial initialization of the data structures, which would lead to data allocation on a single node. For complex application codes, the programming effort required to introduce these loops may be significant. For other important algorithm classes, the access pattern for the main data structures is computed in the program. In such situations it may be difficult, or even impossible, to introduce pre-iteration loops in an efficient way. Instead, some kind of dynamic page placement strategy is required, where misplacement of pages is corrected during the execution by migrating and/or replicating pages to the nodes that perform remote accesses. Dynamic strategies might be explicitly initiated by the application [6], implicitly invoked by software [50], or they may be implicitly invoked by the computer system [70, 13, 17].

A.2. Applications

To evaluate different methods for improving geographical locality we study the performance of four solvers for large-scale partial differential equation (PDE) problems. In the discretization of a PDE, a grid of computational cells is introduced. The grid may be structured or unstructured, resulting in different implementations of the algorithms and different types of data access patterns. Most algorithms for solving PDEs could be viewed as an iterative process, where the loop body consists of a (generalized) multiplication of a very large and sparse matrix by a vector containing one or a few entries per cell in the grid. When a structured grid is used, the sparsity pattern of the matrix is pre-determined and highly structured. The memory access pattern of the codes exhibit large spatial and temporal locality, and the codes are normally very efficient. For an unstructured grid, the sparsity pattern of the matrix is unstructured and determined at runtime. Here, the spatial locality is normally reduced compared to a structured grid discretization because of the more irregular access pattern.

We have noted that benchmark codes often solve simplified PDE problems using standardized algorithms, which may lead to different performance results than for kernels from advanced application codes. We therefore perform experiments using kernels from industrial applications as well as standard benchmark codes from the NAS NPB3.0-OMP suite [37]. More details on the applications are given in [43]. All codes are written in Fortran 90, and parallelized using OpenMP. The following PDE solvers are studied:

NAS-MG The NAS MG benchmark, size B. Solves the Poisson equation on a $256 \times 256 \times 256$ grid using a multi-grid method.

I-MG An industrial CFD solver kernel. Solves the time-independent Euler equations describing compressible flow using an advanced discretization on a grid with $128 \times 128 \times 128$ cells. Also here a multi-grid method is used.

NAS-CG The NAS CG benchmark, size B. Solves a sparse system of equations with an unstructured coefficient matrix using the conjugate gradient method. The system of equations has 75000 unknowns, and the sparse matrix has 13708072 non-zero elements, resulting in a non-zero density of 0.24%.

I-CG An industrial CEM solver. Solves a system of equations with an unstructured coefficient matrix arising in the solution of the Maxwell equations around an aircraft. Again, the conjugate gradient method is used. This system of equations has 1794058 unknowns, and the non-zero density is only 0.0009%.

A.3. Results

On the a SF15k system, a dedicated domain consisting of four nodes was used, and the scheduling of threads to the nodes was controlled by binding the threads to Solaris processor sets. Each node contains four 900 MHz UltraSPARC-IIICu CPUs and 4 GByte of local memory. The data sets used are all approximately 500 MByte, and are easily stored in a single node. Within a node, the access time to local main memory is uniform. The nodes are connected via a crossbar interconnect, forming a cc-NUMA system. The NUMA-ratio is only approximately 2, which is small compared to other commercial cc-NUMA systems available today.

All application codes were compiled with the Sun ONE Studio 8 compilers using the flags `-fast -openmp -xtarget=ultra3cu -xarch=v9b`, and the experiments were performed using the 12/03-beta release of Solaris 9. Here, a static first-touch page placement strategy is used and support for dynamic, application-initiated migration of data is available in the form of a migrate-on-next-touch feature [66]. Migration is activated using a call to the `madvise(3C)` routine, where the operating system is advised to reset the mapping of virtual to physical addresses for a given range of memory pages, and to redo the first-touch data placement. The effect is that a page will be migrated if a thread in another node performs the next access to it.

We have also used a Sun WildFire system with two nodes for some of our experiments. Here, each node has 16 UltraSPARC-II processors running at 250 MHz. This experimental cc-NUMA computer has CPUs which are of an earlier generation, but includes an interesting dynamic and transparent page placement optimization capability. The system runs a special version of Solaris 2.6, where pages are initially allocated using the first-touch strategy. During program execution a software daemon detects pages which have been placed in the wrong node and migrates them without any involvement from the application code. Furthermore, the system also detects pages which are used by threads in both nodes and replicates them in both nodes. A per-cache-line coherence protocol keeps coherence between the replicated cache lines.

We begin by studying the impact of geographical locality for our codes using the SF15k system. We focus on isolating the effects of the placement of data, and do not attempt to assess the more complex issue of the scalability of the codes. First, we measure the execution time for our codes using four threads on a single node. In this case, the first touch policy results in that all application data is allocated locally, and the memory access time is uniform. These timings are denoted UMA in the tables and figures. We then compare the UMA timings to the corresponding execution times when executing the codes in cc-NUMA mode, running a single thread on each of the four nodes. Here, three different data placement schemes are used:

Serial initialization (SI) The main data arrays are initialized in a serial section of the code, resulting in that the pages containing the arrays are allocated on a single node. This is a common situation when application codes are naively parallelized using OpenMP.

Parallel initialization (PI) The main data arrays are initialized in pre-iteration loops within the main parallel region. The first-touch allocation results in that the pages containing the arrays are distributed over the four nodes.

Serial initialization + Migration (SI+MIG) The main arrays are initialized using serial initialization. A migrate-on-next-touch directive is inserted at the first iteration in the algorithm. This results in that the pages containing the arrays will be migrated according to the scheduling of threads used for the main iteration loop.

Application	Time				Remote accesses			
	UMA	SI	PI	SI+MIG	UMA	SI	PI	SI+MIG
NAS-CG	1.00 (233.9s)	1.12	1.08	1.04	0.0%	75.1%	35.9%	6.2%
NAS-MG	1.00 (20.8s)	1.58	1.43	1.15	0.0%	72.4%	48.5%	11.0%
I-CG	1.00 (39.9s)	1.58	N/A	1.15	0.0%	67.9%	N/A	31.4%
I-MG	1.00 (219.1s)	1.18	1.00	1.01	0.1%	77.1%	4.3%	3.6%

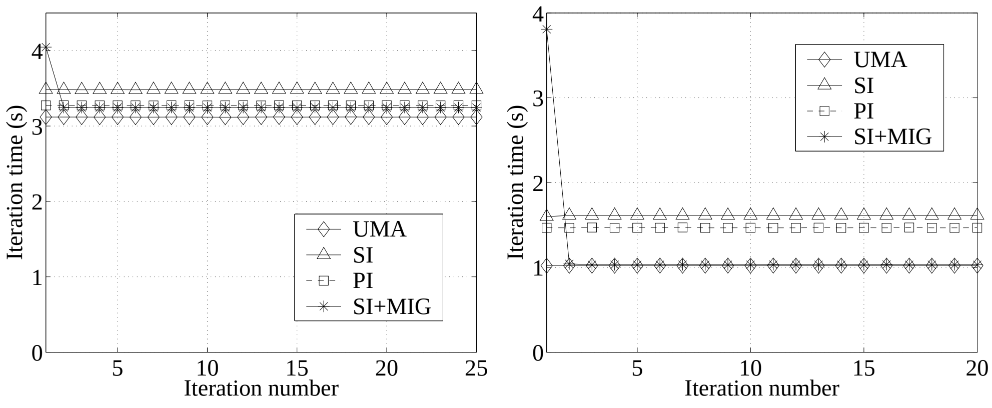
Table A.1.: Timings and fraction of remote memory accesses for the different codes and data placement settings. The timings for the cc-NUMA settings are normalized to the UMA case

In the original NAS-CG and NAS-MG benchmarks, parallel pre-iteration loops have been included [37]. The results for PI are thus obtained using the standard codes, while the results for SI are obtained by modifying the codes so that the initialization loops are performed by only one thread. In the I-CG code, the sparse matrix data is read from a file, and it is not possible to include a pre-iteration loop to successfully distribute the data over the nodes using first touch allocation. Hence, no PI results are presented for this code.

In Table A.1, the timings for the different codes and data placement settings are shown. The timings are normalized to the UMA case, where the times are given also in seconds. From the results, it is clear that the geographical locality of data does affect the performance for all four codes. For the I-MG code, both the PI and the SI+MIG strategy are very successful and the performance is effectively the same as for the UMA case. This code has a very good cache hit rate, and the remote accesses produced for the SI strategy do not reduce the performance very much either. For the NAS-MG code the smaller cache hit ratio results in that this code is more sensitive to geographical misplacement of data. Also, NAS-MG contains more synchronization primitives than I-MG, which possibly affects the performance when executing in cc-NUMA mode. Note that even for the NAS-MG code, the SI+MIG scheme is more efficient than PI. This shows that sometimes it is difficult to introduce efficient pre-iteration loops also for structured problems.

For the NAS-CG code, the relatively dense matrix results in reasonable cache hit ratio and the effect of geographical misplacement is not very large. Again SI+MIG is more efficient than than PI, even though it is possible to introduce a pre-iteration loop for this unstructured problem. For I-CG, the matrix is much sparser, and the caches are not so well utilized as for NAS-CG. As remarked earlier, it is not possible to include pre-iteration loops in this code. There is a significant difference in performance between the unmodified code (SI) and the version where a migrate-on-next-touch directive is added (SI+MIG).

In the experiments, we have also used the UltraSPARC-III hardware counters to measure the number of L2 cache misses which are served by local and remote memory respectively. In Table A.1, the fractions of remote accesses for the different codes and data placement settings are shown. Comparing the different columns of Table A.1, it



(a) Execution time per iteration of NAS-CG. Only the first 25 iterations are shown in the graph

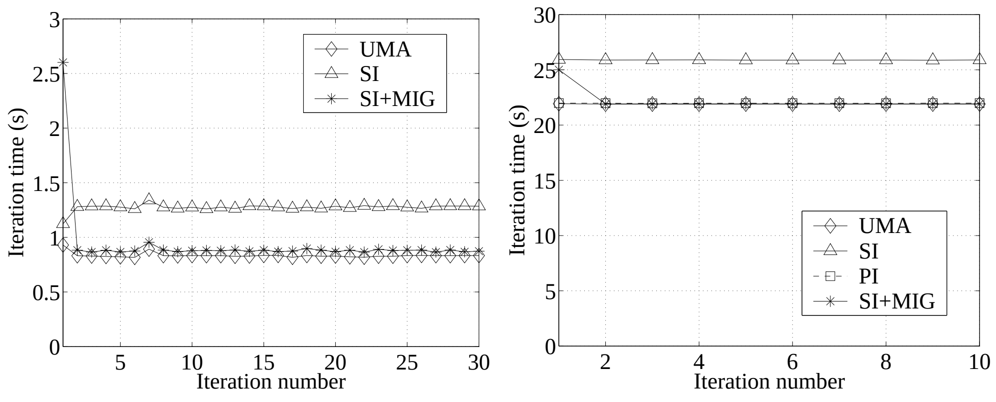
(b) Execution time per iteration of NAS-MG

Figure A.1.: Execution time per iteration for NAS-CG and NAS-MG on the SF15K using 4 threads

is verified that that the differences in overhead between the cc-NUMA cases compared to the UMA timings is related to the fraction of remote memory accesses performed.

We now study the overhead for the dynamic migration in the SI+MIG scheme. In Figures A.1(a), A.1(b), A.2(a), and A.2(b), the execution time per iteration for the different codes and data placement settings is shown. As expected, the figures show that the overhead introduced by migration is completely attributed to the first iteration. The time required for migration varies from 0.80 s for the NAS-CG code to 3.09 s for the I-MG code. Unfortunately, we can not measure the number of pages actually migrated, and we do not attempt to explain the differences between the migration times. For the NAS-MG and I-CG codes, the migration overhead is significant compared to the time required for one iteration. If the SI+MIG scheme is used for these codes, approximately five iterations must be performed before there is any gain from migrating the data. For the NAS-CG code the relative overhead is smaller, and migration is beneficial if two iterations are performed. For the I-MG code, the relative overhead from migration is small, and using the SI+MIG scheme even the first iteration is faster than if the data is kept on a single node. A study of the scalability of the SI+MIG scheme is performed in [43].

Finally, we do a qualitative comparison of the SI+MIG strategy to the transparent, dynamic migration implemented in the Sun WildFire system. In Figures A.3(a) and A.3(b), we show the results for the I-CG and I-MG codes obtained using 4 threads on each of the two nodes in the WildFire system. Here, the SI+TMIG-curves represent timings obtained when migration and replication is enabled, while the SI-curves are obtained by disabling these optimizations and allocating the data at one of the



(a) Execution time per iteration of I-CG (b) Execution time per iteration of I-MG

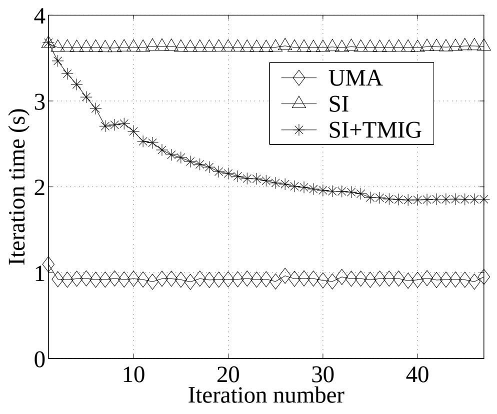
Figure A.2.: Execution time per iteration for I-CG and I-MG on the SF15K using 4 threads

nodes. Comparing the UMA- and SI-curves in Figures A.3(a) and A.3(b) to the corresponding curves for SF15k in Figures A.2(a) and A.2(b), we see that the effect of geographical locality is much larger on WildFire than on SF15k. This is reasonable, since the NUMA-ratio for WildFire is approximately three times larger than for SF15k. From the figures, it is also clear that the transparent migration is active during several iterations. The reason is that, first the software daemon must detect which pages are candidates for migration, and secondly the number of pages migrated per time unit is limited by a parameter in the operating system. One important effect of this is that on the WildFire system, it is beneficial to activate migration even if very few iterations are performed.

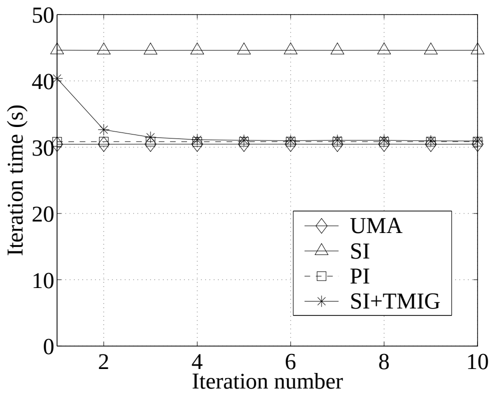
A.4. Conclusions

Our results show that geographical locality is important for the performance of our applications on a modern cc-NUMA system. We also conclude that application-initiated migration leads to better performance than parallel initialization in almost all cases examined, and in some cases the performance is close to that obtained if all threads and their data reside on the same node. The main possible limitations of the validity of these results are that the applications involve only sparse, static numerical operators and that the number of nodes and threads used in our experiments are rather small.

Finally, we have also performed a qualitative comparison of the results for the commercial cc-NUMA to results obtained on a prototype cc-NUMA system, a Sun WildFire server. This system supports fully transparent adaptive memory placement optimization in the hardware, and our results show that this is also a viable alternative on cc-NUMA systems. In fact, for applications where the access pattern changes dynami-



(a) Execution time per iteration of I-CG



(b) Execution time per iteration of I-MG

Figure A.3.: Execution time per iteration for I-CG and I-MG on the Sun WildFire using 8 threads

cally but slowly during execution, a self-optimizing system is probably the only viable solution for improving geographical locality.

Paper B

B. *affinity-on-next-touch*: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System

Henrik Löf and Sverker Holmgren
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
henrik.lof,sverker.holmgren@it.uu.se

Abstract

To achieve close to optimal performance on cc-NUMA systems for shared memory parallel applications with complex data access patterns, a mechanism for co-locating threads and the data during the execution of the program is needed. The affinity-on-next-touch *procedure studied in this paper is based on re-doing the standard first-touch allocation at explicitly given locations in the code. We study the performance of a parallelized scientific computing application for which thread-data affinity can not be created by standard methods. We observe a performance improvement of 64% if the affinity-on-next-touch procedure is invoked once at a critical location in the code. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed the handling of page entries in the TLBs. The cost for actually migrating data is negligible. Using larger but fewer pages we measure a performance improvement of 127% compared to the original code.*

B.1. Introduction

The memory access patterns of scientific computing codes are becoming more and more complex. Unstructured grids and adaptive schemes are introduced in the computational models to increase the accuracy and resolution, resulting in that the data access patterns are more irregular and possibly also changes during execution. When parallelizing such codes, a shared memory programming model is preferred. One reason is that the programmer does not have to explicitly manage the communication between processors. Another important reason is that programming tools like OpenMP use a uniform shared memory model, implying that the programmer does not have to specify how the data should be partitioned and where it should be stored in the memory. However, modern scalable shared memory computers are normally of cc-NUMA type. In such systems, memory and CPUs are distributed over several nodes, where each node normally consists of a physical CPU-memory board. The access time for memory within a node is smaller than the time required for a remote memory access, and the difference is measured by the NUMA-ratio,

$$\text{NUMA-ratio} = \frac{\text{Remote access time}}{\text{Local access time}}.$$

On a NUMA system, the distribution of data in the shared memory has an effect on the performance of programs. Misplacement of data may cause that the performance and/or scalability is reduced even if the code is highly optimized in other aspects.

Possible ways of reducing negative performance effects arising from the mismatch between the programming model and the underlying computer architecture has been discussed by many authors, see e.g. [8, 49, 17, 33, 47]. The proposals fall into two main categories:

Extend the programming model: Here, the programmer may explicitly specify the data distribution using directives that are added to the programming model, see e.g [6]

Extend the computer system: Here, the OpenMP runtime system, the operating system and/or special hardware provide mechanisms for detecting and correcting misplacement of data in a transparent way while the programming model is left unmodified [70, 51, 49, 10].

Both of these approaches have advantages and disadvantages. If the programmer knows how data is accessed, it may be easy to explicitly specify a good data distribution. Then, a programming model where this distribution can be described directly will probably result in good performance. On the other hand, if the data access pattern is complex, a substantial effort may be needed by the programmer to determine a good data distribution, or it may not even be possible. In such situations, a solution where the computer system adaptively determines the data distribution in a transparent way may be more beneficial. However, there is a risk that the overhead introduced by the modifications of the system will result in a loss of performance and/or scalability, counteracting positive effects of improving the data distribution.

In this paper, we study a method for dealing with the data distribution problem based on a run-time mechanism for co-locating data and the threads that use it on the same node of the computer system. The mechanism can be made available to the programmer by a portable and simple addition to the programming model. We evaluate the performance effect of applying this method to an industrial scientific application, parallelized using OpenMP and executed on a cc-NUMA computer. The results show that it is possible to decrease the execution time significantly compared to if the original code is used. The results also show that, on the computer system studied, the overhead introduced by the directive is dominated by handling TLB coherence and not by the actual movement of data. This implies that the overhead can be reduced by using large page sizes.

B.2. Thread-data affinity on cc-NUMA systems

When executing a parallel program on a cc-NUMA system, it must be determined how the data should be partitioned over the nodes and at which nodes the threads should be run. To get high performance, the number of accesses to remote memory should be small. This can be achieved if the *thread-data affinity* is large, i.e. if each data item resides in the memory of the node where the thread that uses it the most is running.

The standard method for creating thread-data affinity is to use a first-touch strategy. When a memory page is accessed for the first time during execution, the operating system has to reserve a physical page for it in one of the nodes. Using first-touch data placement, the page is allocated in the node where the thread responsible for the first access is running. The hope is that this thread will also be the one that accesses the page the most later, and that the thread will continue to run in the node where it was started.

If the data access pattern changes after the initialization or if the threads move between nodes during execution, the thread-data affinity is normally reduced. The first situation occurs for example if the data is initialized by a single thread before the parallel region in the program begins. In such cases the first-touch strategy normally results in that all data is allocated on a single node. This problem might sometimes be tackled by rewriting the code so that it performs a parallel initialization of the data, see e.g. [37, 33]. However, in other cases a substantial programming effort is needed to achieve this. Furthermore, some form of indirect addressing is used in many codes, for example for handling unstructured computational grids. Then, the data access pattern is determined by the data itself, and it is not possible to exploit the standard first-touch strategy in an optimal way. In such cases, a viable approach for creating thread-data affinity is to introduce a mechanism for re-doing the first-touch placement of data at some locations in the code during the execution. This would allow a re-creation of thread-data affinity by effectively migrating data to the correct nodes. We use the term *affinity-on-next-touch* for such an operation.

If the access pattern of the main part of the program is static, it might be sufficient to invoke the affinity-on-next-touch procedure once after the initialization. For example, for an application using an unstructured grid, the procedure could be invoked for migrating data to the correct nodes once the access pattern is determined. In general scientific computing codes, a few distinct phases with different access patterns can often be identified. It is normally easy for the programmer to determine where affinity should be created, which implies that it is reasonable to add an affinity-on-next-touch directive to the programming model. A directive of this type is also mentioned in [51], and a first implementation is found in the Compaq OpenMP compilers [6]. An affinity-on-next-touch directive does not specify how and where to migrate data, which makes is portable and easy to use. Moreover, the time-consuming phases in computational codes normally consist of iterations where the data structures are repeatedly accessed using the same access pattern. In such cases, the overhead introduced by invoking the directive before the loop is amortized over the iterations.

If the data access pattern changes more often during the execution, the affinity-on-next-touch mechanism could be invoked repeatedly, or a fully transparent scheme for adaptive data distribution could be used. In a transparent scheme, ways of detecting candidate pages for migration and/or replication must be built into the system, and algorithms for deciding where and when to move data must be frequently executed. When implementing schemes of this type, great care must be taken not to introduce significant amounts of overhead.

B.3. The application

To evaluate the applicability of an affinity-on-next-touch directive, we study the performance of an industrial solver for the Maxwell equations in 3 dimensions, describing the scattering of electromagnetic waves from e.g. an airplane [24]. The solver uses an unstructured grid and a finite element discretization, and solves the resulting sparse system of equations using a conjugate gradient (CG) solver. In the iterative solver, a bandwidth minimization algorithm [28] has been introduced [46]. The code is written in Fortran 90 and C, and parallelized using OpenMP.

The data access pattern in the computations is static and unstructured, and it is possible identified two main phases in the code:

Assembly of iteration matrix This part of the code is not very time-consuming. Also, it is rather complex and hence not parallelized

Iterative solver This computationally heavy part of the code is parallelized using an algorithm with a small number of synchronization points [46]. For normal application problems, about 50 iterations are required to compute the solution

In the experiments presented below we study a problem where the iteration matrix has 1794058 rows, and each row has on average 15 non-zeros entries. This leads to that the size of the data set is approximately 500 MByte.

Since the assembly of the iteration matrix is performed by a single thread, all of the data will be allocated to a single node after the assembly phase. By invoking the affinity-on-next-touch procedure just before the first iteration in the iterative solver, the data will be migrated to several nodes matching the location of the multiple threads used in the parallel region. In our experiments, we run the iterative solver to convergence once, and investigate the performance gain from creating affinity for this computation. In a real-life computation, the system of equations is solved over and over again for hundreds of time steps, implying that the overhead associated with the creation of thread-data affinity will be amortized over many more iterations.

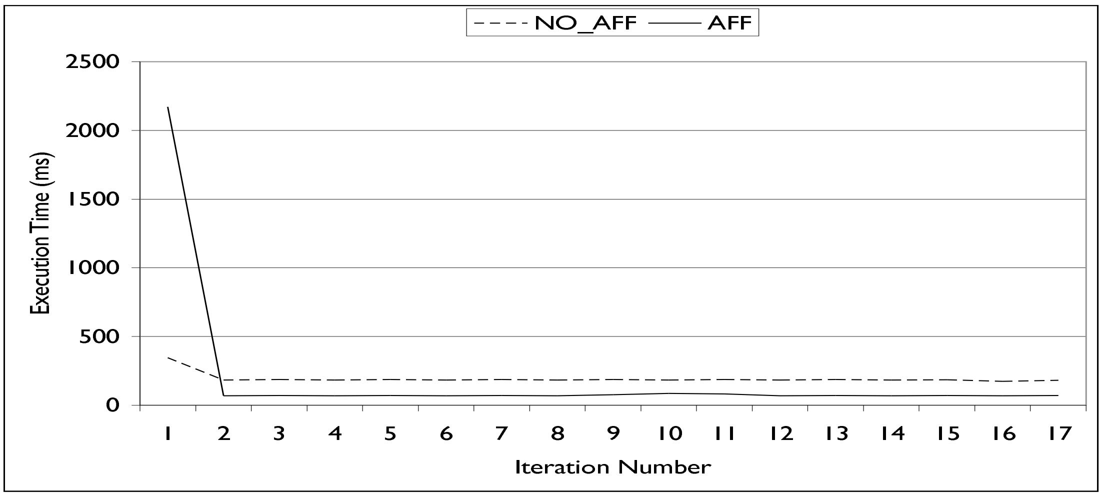


Figure B.1.: The effect of the affinity-on-next-touch procedure when 16 threads are used. The directive was executed just before the first iteration. Only the first 17 out of 47 iterations are shown.

B.4. The cc-NUMA system

The experiments are performed on a dedicated 32 CPU domain on a Sun Fire 15000 system (SF15K) [15]. Here, each node consists of four 900MHz UltraSPARC-IIICu CPUs, each having 8 MByte of L2 cache and 4 GByte of main memory. The system runs Solaris 9, where the affinity-on-next-touch mechanism can be implemented by a call to the `madvise(3C)` library routine.

In the experiments, the setting of a kernel parameter for the Solaris scheduler is modified to increase the probability that the threads stay on the same nodes during the execution of the program [66]. We did not use any binding to CPUs or processor sets.

It should be noted that NUMA-ratio of the SF15K system is only approximately two. This means that we can not expect that the effects of modifying the distribution of data to be dramatic.

B.5. Results

In Figure B.1, the effect of invoking the affinity-on-next-touch mechanism is illustrated. In this experiment, 16 threads are used. The graph shows the execution time for each iteration if the original code is used (`NO_AFF`) and if the affinity-on-next-touch mechanism is introduced before the first iteration in the solver (`AFF`).

In the NO_AFF case, the data remains allocated on a single node. The number of remote accesses is large, resulting in a rather large execution time per iteration and a total execution time of 9.35 s. The slightly larger execution time for the first iteration can be attributed to cold-start effects, where some of the data is cached on multiple processors when the parallel region is entered.

For the AFF case, it is clear that an affinity-on-next-touch directive affects the performance in two ways. In the first iteration, there is a cost for creating the affinity and the execution time for this iteration is increased. After this, the execution time per iteration is significantly reduced. Using the UltraSparc-III hardware counters, a measurement of the number of L2 cache misses that are served from remote memory verifies that the data is actually migrated to the nodes involved in the parallel computations. Therefore, we conclude that the thread-data affinity actually has a positive effect on performance. The average execution time for iteration $2-47$ is reduced from 185 to 71 ms, and the total execution time is reduced to 5.91 s.

In a more general situation, it is clear that the initial cost of creating affinity must be amortized over a number of iterations (or similar) if the total performance should be increased. A simple calculation shows that for our experiment, 17 iterations are needed before the break-even point is reached.

It is interesting to study of the overhead for the affinity-on-next-touch procedure in more detail. In the experiment presented in Figure B.1, we know that data is actually migrated from one node to several others at the first iteration. The overhead for this can be divided into two parts: The cost of copying data and the cost of keeping the page tables coherent. The copy operations are parallelizable, and the performance is only limited by the memory system characteristics. However, the modifications of the page tables are potentially more more problematic.

If a page translation is cached in several translation look-aside buffers (TLBs), we need to enforce coherency among these buffers. This procedure is serial, and involves an expensive *TLB shoot-down* process where dirty translations in the TLBs of processors sharing the page are replaced. This process often involves global kernel locks, polling and/or interrupts [18]. To investigate the relative effect of the two types of overhead, another experiment was performed where the affinity-on-next-touch procedure is invoked twice, once at iteration 15 and once at iteration 40. Since the access pattern is static during the iterations, no data is actually moved on the second invocation. The result show that there was no measurable time-difference iterations 15 and 40, and it is clear that the overhead is dominated by handling the TLBs. This assumption is further supported by results from measuring the number of minor page faults and cross-processor interrupts using kernel statistics.

As described above, we have found that the overhead connected to creating thread-data affinity is dominated by handling entries in the TLBs. It is reasonable to assume that using fewer but larger pages could reduce this overhead. In Figure B.2, the effect of using 64kB pages instead of the standard 8kB pages is show, again for an experiment using 16 threads. In the figure, the two curves in Figure B.1 are repeated (NO_AFF8kB and AFF8kB), but now together with the corresponding curves for 64kB pages (NO_AFF64kB and AFF64kB). It is clear that the overhead in the first iteration is reduced by a factor of 3 when using 64kB pages compared to the 8kB case. Also, the performance for the subsequent iterations is almost unaffected by changing the page size. The total execution time is now reduced to 3.9 s, and the break-even point where the overhead has been amortized is reached already after four iterations. The primary reason for the reduction of the overhead when using a larger page size is that fewer TLB entries must be handled. In the TLB shoot-down process, the initiating thread sends one interrupt per page to invalidate possible shared TLB entries for other threads. The UltraSparc-III processor does not have hardware support for 64kB pages, implying that when 64 kB pages are used, in practice one interrupt invalidates eight 8kB pages. The results for the original code in Figure B.2 indicate that there is another, smaller positive effect provided by a general reduction of the cold-start effects for entering the parallel region when the number of TLB entries needed to cache the working set is reduced.

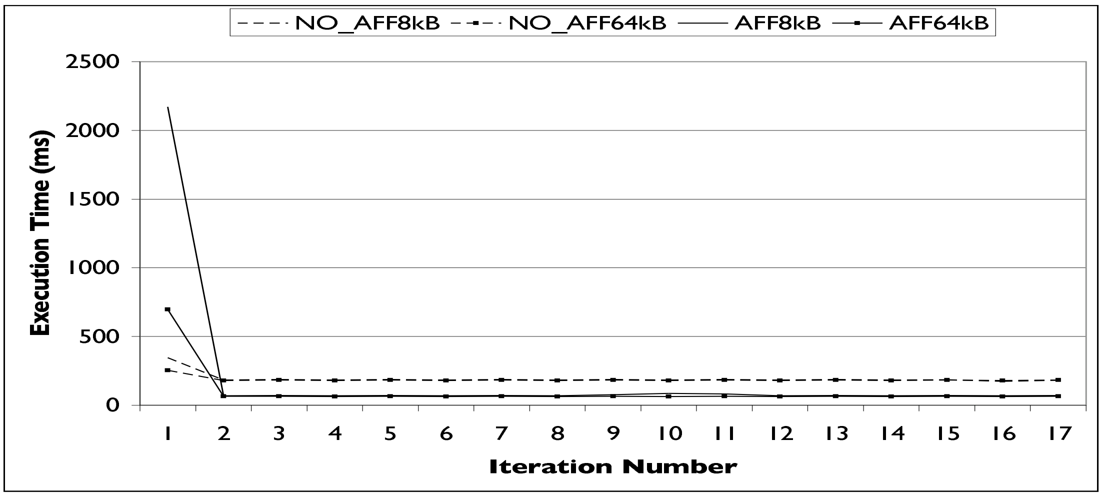


Figure B.2.: A comparison of the effect of the affinity-on-next-touch procedure when 8kB and 64kB page are used.

In Figure B.3, the effect of invoking the affinity-on-next-touch procedure for different numbers of threads is shown. The bars show the improvement of the total execution time when using 8kB and 64kB pages compared to the cases where the original code is executed using 8kB pages using the same number of threads. From the figure, it is clear that enforcing a proper distribution of data by invoking an affinity-on-next-touch procedure in general improves the performance for the application studied here. Also, it is clear that in all cases the performance improvement is larger when 64kB pages are used instead of 8kB pages. The performance decrease for 2 threads and 8kB pages is explained by that in this case the two threads are scheduled to the same node, and there will be no possible benefit of migrating data. Then, the overhead in the first iteration is not followed by any performance improvement in subsequent iterations.

A more detailed study of the experiments described by Figure B.3 reveals that, using

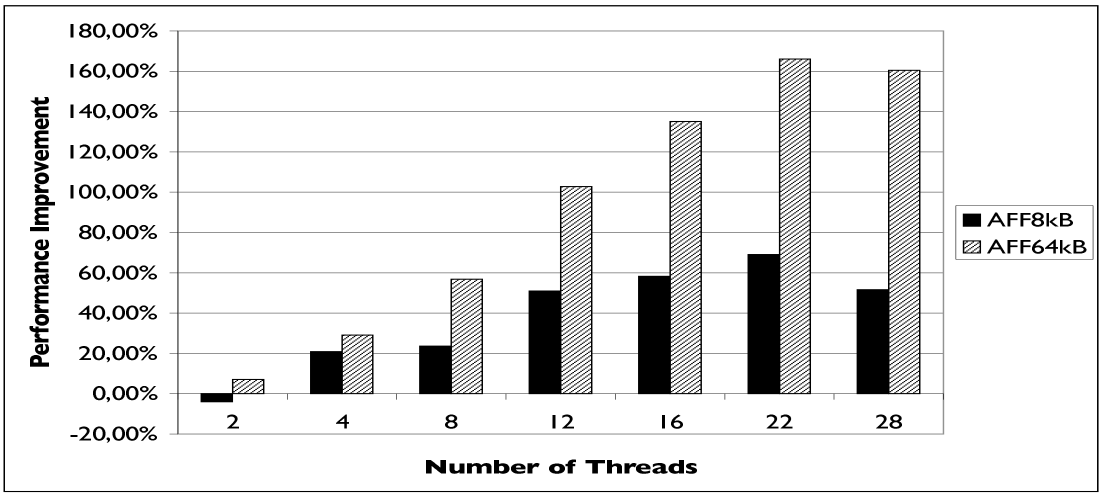


Figure B.3.: Performance improvement of using the affinity-on-next-touch procedure compared to if the original code is used. Results for both 8kB and 64kB pages are shown.

the implementation studied in this paper, the affinity-on-next-touch mechanism is not scalable. The initial iteration overhead is almost constant as the number of threads is increased. As the iterations become faster when more threads are used, more and more iterations must be performed before the break-even point is reached and the initial overhead is amortized. However, this effect is reduced by using 64kB pages.

Finally, we present a traditional speedup graph where the code is executed using different numbers of threads and the affinity-on-next-touch procedure is invoked as described earlier. Results for both 8kB and 64kB pages are shown. B.4. From the figure, it is obvious that reducing the initial iteration overhead has a positive effect on the scalability of the parallel implementation.

B.6. Conclusions

To achieve close to optimal performance on cc-NUMA systems for shared memory applications with complex and possibly changing data access patterns, a mechanism for creating or re-creating thread-data affinity during the execution of the program is needed. In this paper, we have evaluated such a procedure, denoted *affinity-on-next-touch*, based on re-doing the standard first-touch allocation at given locations in the code. This normally results in that data is migrated between the nodes in the system. The experiments are performed using a parallelized scientific computing application where thread-data affinity can not be created by standard methods. We find that the execution time can be significantly reduced by invoking the affinity-on-next-touch procedure. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed the handling of page entries in the TLBs.

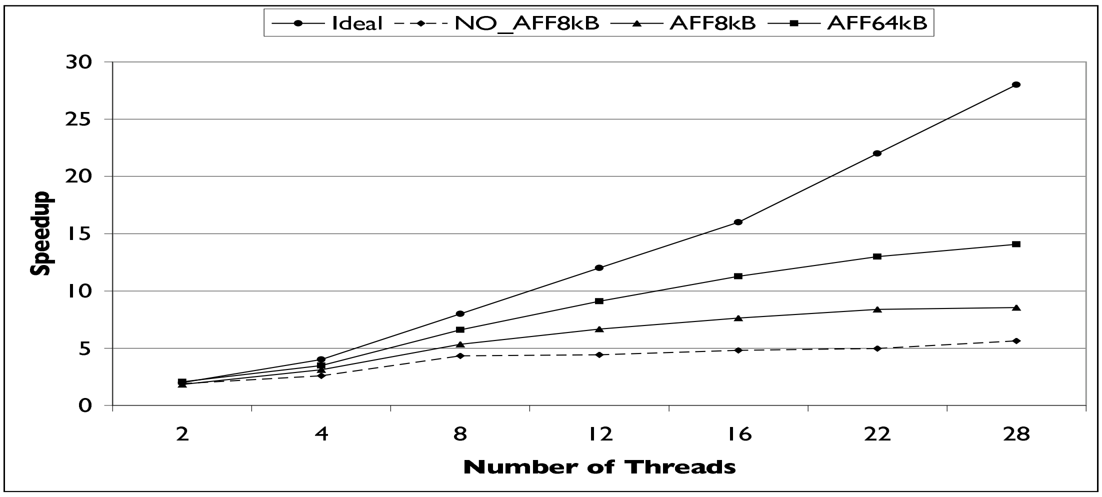


Figure B.4.: The speedup of the solver using the original code (`NO_AFF8kB`), affinity-on-next-touch and 8k pages (`AFF8kB`), and affinity-on-next-touch and 64k pages (`AFF64kB`).

The cost for actually migrating data is negligible. Led by this observation, we manage to reduce the overhead by increasing the page size, which results in that fewer TLB entries must be handled.

Since the affinity-on-next-touch procedure is invoked at distinct locations in the algorithm, it is natural to suggest that it should be accessible by a directive in programming tools like OpenMP. The programmer usually knows where thread-data affinity is critical for performance. In OpenMP, other options are to add an affinity-creation attribute to the parallel region directive, or to automatically invoke the procedure for creating affinity at the beginning of each parallel region. The latter solution has the advantage that the programming model is left unmodified.

Paper C

C. Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures

Henrik Löf and Jarmo Rantakokko
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
henrik.lof,jarmo.rantakokko@it.uu.se

Abstract

OpenMP is an architecture-independent language for programming in the shared memory model. OpenMP is designed to be simple and powerful in terms of programming abstractions. Unfortunately, the architecture-independent abstractions sometimes come with the price of low parallel performance. This is especially true for applications with unstructured data access pattern running on distributed shared memory systems (DSM). Here proper data distribution and algorithmic optimizations play a vital role for performance. In this article we have investigated ways of improving the performance of an industrial class conjugate gradient (CG) solver, implemented in OpenMP running on two types of shared memory systems.

We have evaluated bandwidth minimization, graph partitioning and reformulations of the original algorithm reducing global barriers. By a detailed analysis of barrier time and memory system performance we found that bandwidth minimization is the most important optimization reducing both L2 misses and remote memory accesses. On an uniform memory system we get perfect scaling. On a NUMA system the performance is significantly improved with the algorithmic optimizations leaving the system dependent global reduction operations as a bottleneck.

C.1. Introduction

Many parallel applications exhibit unstructured and/or highly irregular patterns of communication. Implementing such applications often require substantial software engineering efforts regardless of the parallel programming model used. These efforts can however be lowered using a shared memory model, like OpenMP, since in this model the orchestration of the communication is handled in an implicit manner hidden from the programmer. Unfortunately, this very attractive feature of OpenMP often comes with the price of lower parallel performance compared to other, more explicit programming models such as message passing. OpenMP is designed to be a simple and architecture independent tool, i.e., OpenMP uses a *uniform* shared memory model. However, most large-scale shared memory systems are of cc-NUMA type. The OpenMP model still works well for applications with structured and local data dependencies [44] but for applications with unstructured data dependencies special care have to be taken to the algorithm.

An example of an important scientific application exhibiting an unstructured pattern of communication is an iterative solver for large sparse systems of equations. Many algorithmic optimizations have been proposed to increase the performance of iterative solvers. Most of them have been evaluated in a message passing programming model. In this study, we want to evaluate these algorithmic optimizations on a conjugate gradient (CG) solver implemented in OpenMP. We want to investigate whether these optimizations can be used to increase the performance on both uniform (SMP) and non-uniform (cc-NUMA) shared memory architectures. Can we get high performance using algorithmic optimizations and still keep the software engineering efforts low using the shared memory model?

We use a real industrial data set from a finite element discretization of the Maxwell equations in three dimensions on a fighter jet geometry. The code was run on a Sun Fire 15000 (SF15K) cc-NUMA server [15] and on a Sun Enterprise 10000 (E10K) SMP, a uniform memory access system. The algorithmic optimizations we have studied are, reformulation of the algorithm for reducing global barriers [16], bandwidth minimization of the iteration matrix [28] and graph partitioning using the MeTiS tool [38]. We have also used a page migration feature of Solaris 9 [66] to improve the data distribution of the application.

The effects of the different optimizations were quantified. The parallel overhead is dominated by the poor locality properties this application exhibits. Hence, bandwidth minimization showed to give the highest performance improvements for both uniform and non-uniform architectures. Load balance showed to be of less importance and a simple linear partitioner in combination with bandwidth minimization gave the best results. We saw no need for more advanced partitioning like graph partitioning. Reducing global barriers had no significant effect on these small size systems (28 threads) but gave some improvements in combination with bandwidth minimization (when the load imbalance was minimized).

C.2. Parallelizing the CG-algorithm using OpenMP

When parallelizing iterative solvers like the CG-algorithm [3] there are three basic types of operations to consider:

Algorithm 3 Method of Conjugate Gradients

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$
and set $p_0 = r_0$.
For $k = 0, 1, \dots$
(1)     Compute and Store $Ap_k$
(2)     Compute $< p_k, Ap_k >$
(3)     $$\alpha_k = \frac{< r_k, r_k >}{< p_k, Ap_k >}$$
(4)     $x_{k+1} = x_k + \alpha_k p_k$
(5)     Compute $r_{k+1} = r_k - \alpha_k Ap_k$
(6)     Compute $< r_{k+1}, r_{k+1} >$
(7)     $$\beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$$
(8)     Compute $p_{k+1} = r_{k+1} + \beta_k p_k$
```

Vector operations (Lines (4),(5) and (8) of Algorithm 3) These operations are trivially parallelized and they require very large vectors to amortize the parallel overhead. In OpenMP this operation translates into a simple parallelized loop.

Inner products (Lines (2) and (6) of Algorithm 3) These operations map well onto OpenMP reductions and they are inherently serial in the sense that they introduce global barriers.

Sparse matrix-vector product (SpMxV) (Line (1) of Algorithm 3) This operation stands for the significant part of the execution time. A straight forward parallelization consists of assigning rows or columns of the matrix to different threads using a loop directive on the outer-most loop.

The main sources of parallel overhead in any shared memory implementation are: global barriers, true/false sharing effects and load balance. On a NUMA system additional overheads come from non-optimal data placement which will trigger remote memory accesses. In the case of the CG algorithm and OpenMP, there are no major false sharing effects[1] as most stores are primarily stride-1 accesses which map very well to the static partitions generated by OpenMP loop directives.

When it comes to global barriers, we can see from Algorithm 6 (Appendix A), that we have several flow dependencies that must be protected by global barriers. In OpenMP we have implicit barriers at the end of the workshare directives. In some

[1]There might still be false-sharing effects in the OpenMP runtime system. As an example the final stage of a reduction, which must be performed serially, should use padded arrays to minimize false sharing.

Algorithm 4 OpenMP implementation in C of a sparse matrix vector product for the compressed sparse row (CSR) format.

```
void spmxv(const double *val, const int *col,
            const int *row, const double *x,
            double *v, int nrows)
{
   register int i,j;
   register double d0;

#pragma omp for private(i,j,d0) nowait
   for( i = 0; i < nrows; i++ )
     {
       d0 = 0.0;
       for( j = row[i]; j < row[i+1]; j++ )
         d0 += val[j] * x[ col[j] ];
       v[i] = d0;
     }
}
```

cases these can be removed using the `NOWAIT` clause. Also, in OpenMP an inner product requires two barriers, one to clear the reduction variable and one at the end of the reduction. In total, we found that a correct OpenMP implementation of the basic CG-algorithm needs 7 barriers per iteration, see Algorithm 6.

Finally, load balancing is good for the vector operations and the reductions if they are statically partitioned. The load balance in the SpMxV is however dependent on the structure of the sparse matrix, and the partitioning of the data, ie how the OpenMP threads are scheduled in the outer-loop of the SpMxV, see Algorithm 4. Even though we assign an equal amount of rows to each thread, load balance may still be poor since the number of non-zero elements per row varies.

C.3. Algorithmic optimizations

We can improve the performance of a standard implementation using some well known optimizations. To address the parallel overheads we have studied bandwidth minimization (true sharing), reformulation of algorithm (global barriers), and graph partitioning (load balance).

C.3.1. Cache-aware optimizations to the SpMxV

The SpMxV have several performance problems on modern microprocessors. First of all, this operation is memory-bound, since each element in the result vector `v` require more memory operations than floating-point operations, see Algorithm 4. The inner loop can be very short and varies in size between the rows. Also, the spatial locality in the right-hand side (RHS) vector `x` is very bad due to the indirect addressing through the column index vector `col`. The spatial locality for the row and column vectors are however good since they exhibit perfect stride-1 access patterns. The distributions of elements of each row or column of the matrix determines the amount of reuse present in the cache blocks of the RHS `x`. In the case of FEM, this pattern is determined by the type and numbering of elements and also on the geometry of the problem. Furthermore, the number of non-zero elements in each row or column is bound by how many elements that geometrically couple in physical space. For 3-dimensional problems, a common number of non-zeros per row or column is about 20. This small number of elements and the irregularity in loop length makes it hard for an optimizing compiler to use standard optimizations such as blocking, unrolling and tiling on the inner-loop.

Using the fact that any sparse matrix can be interpreted as an adjacency matrix of a corresponding graph, we can apply graph theoretical methods to improve the locality of the SpMxV. Remember that the spatial locality of the RHS accesses in the SpMxV will probably increase if the matrix contain large dense blocks since we can reuse more elements of the cached block. Unfortunately, many applications does not exhibit a large dense blocks, but the matrix can be re-ordered to maximize the number of such blocks.

One way of finding dense blocks is to minimize the bandwidth of the matrix using graph theoretical methods. This has been successfully employed in several previous studies for uniform memory systems. In a DSM or NUMA setting, the prize of a cache miss can be much higher due to the possibility of the miss to be served by remote memory. Hence, we would expect bandwidth minimization to be efficient on non-uniform architectures as well. In this study we use an optimization of the standard method Reverse Cuthill-McKee (RCM) [19] developed by Gibbs, Pool and Stockmeyer (GPS). The GPS method give equal quality minimizations in less time as shown in [28].

C.3.2. Reducing the number of global barriers

The original algorithm requires 7 barriers in total to be correct. Two barriers can immediately be removed by making private copies of the global variables α and β. Moreover, we can remove two barriers by interleaving the zeroing of one reduction variable with the reduction of the other variable and vice versa, exploiting the reduction barriers. This gives an optimized algorithm with only three barriers per iteration, see Algorithm 7 (Appendix A). In this algorithm we assume that the runtime system will produce identical partitions for all `STATIC` schedules.

Furthermore, we can use the s-step formulation of the CG-algorithm to remove one more barrier. Originally developed in a message passing environment, this optimization tries to reduce the number of inner products by computing several CG iterations in parallel. Using the identity

$$< Ap_k, p_k > = < Ar_k, r_k > - \beta_{k-1}/\alpha_{k-1} < r_k, r_k >$$

Chronopoulos et al. formed the *s-step* ($s = 1$) version of the CG algorithm, see Algorithm 5. Here, an extra vector operation is introduced to remove one inner product,

Algorithm 5 S-step formulation of CG

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$
and set $p_0 = r_0$.
Compute $Ar_0$, set $b_{-1} = 0$ and compute
        $\alpha_0 = \frac{< r_0, r_0 >}{< r_0, Ar_0 >}$
For $k = 1, 2, \ldots$
(1)    $p_k = r_k + \beta_{k-1} p_{k-1}$
(2)    $Ap_k = Ar_k + \beta_{k-1} Ap_{k-1}$
(3)    $x_{k+1} = x_k + \alpha_k p_k$
(4)    $r_{k+1} = r_k - \alpha_k Ap_k$
(5)    Compute and Store $Ar_{k+1}$
(6)    Compute
            $< Ar_{k+1}, r_{k+1} >$ and $< r_{k+1}, r_{k+1} >$
(7)
            $\beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$
(8)
            $\alpha_{k+1} = \frac{< r_{k+1}, r_{k+1} >}{< Ar_{k+1}, r_{k+1} > - (\beta_k / \alpha_k) < r_k, r_k >}$
```

hence, removing also one barrier and giving a total of two barriers per iteration (see Algorithm 8 Appendix A). However, the s-step formulation contain more arithmetical work than the standard formulation.

C.3.3. Improving load-balance

Graph partitioning is a standard way of minimizing interprocessor communication and producing load balanced computations for mesh based applications[2], see [21]. They are typically used in implementations on systems with a distributed memory where the computation needs to be explicitly partitioned in some way. In a shared memory setting we can permute the matrix using a partition vector from a graph partitioner to create load balance partitions. An outer loop is added to the SpMxV indexed by the OpenMP threadID which fetches the partition boundaries each thread will use. A graph partitioner can potentially also effect the locality of the implementation, something we want to study further here. As the partitioner minimizes the *edge-cut*, which is an approximation of communication volume, the partitions could also decrease the number of remote accesses although some communications will be local. We used the k-way multilevel recursive bisection routine `METIS_PartGraphKway` of the MeTiS

[2]To only achieve load balance one can use a sequence partitioning method for the rows.

graph partitioner using Heavy-Edge matching and Random matching with maximized connectivity throughout this study.

C.3.4. Related Work

The CG algorithm has been extensively used and studied in the numerical community [29, 3]. Several attempts have been made to increase the performance of the SpMxV [54, 11]. Apart from reordering, which we also use, other optimizations are: register blocking to lower the amount of load instructions [54, 71, 69], address precomputation [69] and cache blocking [71].

Most of the work, when it comes to optimizing the parallel performance of the algorithm has been conducted in the distributed memory setting. Early work in the shared memory setting include Brehm et al [9], where they studied the performance of the CG algorithm on earlier type of shared memory machines from vendors like Alliant, Sequent and Encore. In the DSM setting, Oliker et al. [52] comes closest to our work. Here, the authors showed that for a similar implementation of the CG algorithm, running on an SGI Origin 2000 [41] and using SGI's native shared memory pragmas, the performance was very much increased when proper data distribution and bandwidth minimization was used. In fact, the performance of the shared memory implementation was as good as a pure message passing implementation. In this article we use data from a real industrial application and we also perform a more detailed analysis of the different overheads associated with the parallelization and the computer system. Our goal is to understand how to produce the most efficient implementation of a particular solver rather than comparing different programming models for a given problem which was the case in Oliker et. al.

In the OpenMP/DSM setting there has been a lot of activity on deciding how to achieve data distribution on non-uniform architectures [49, 6, 10, 44]. These articles all discuss the CG benchmark of the NAS NPB3.0 OpenMP suite. Unfortunately, the data in this benchmark is not very realistic as it is randomly generated and hence differs from real world data which often has some kind of structure.

C.4. Methodology

In this section we describe additional details about the input data and the actual source code. We also explain the experimental setups and our experimental methodology to quantify the effects of the different optimizations.

C.4.1. Execution environment

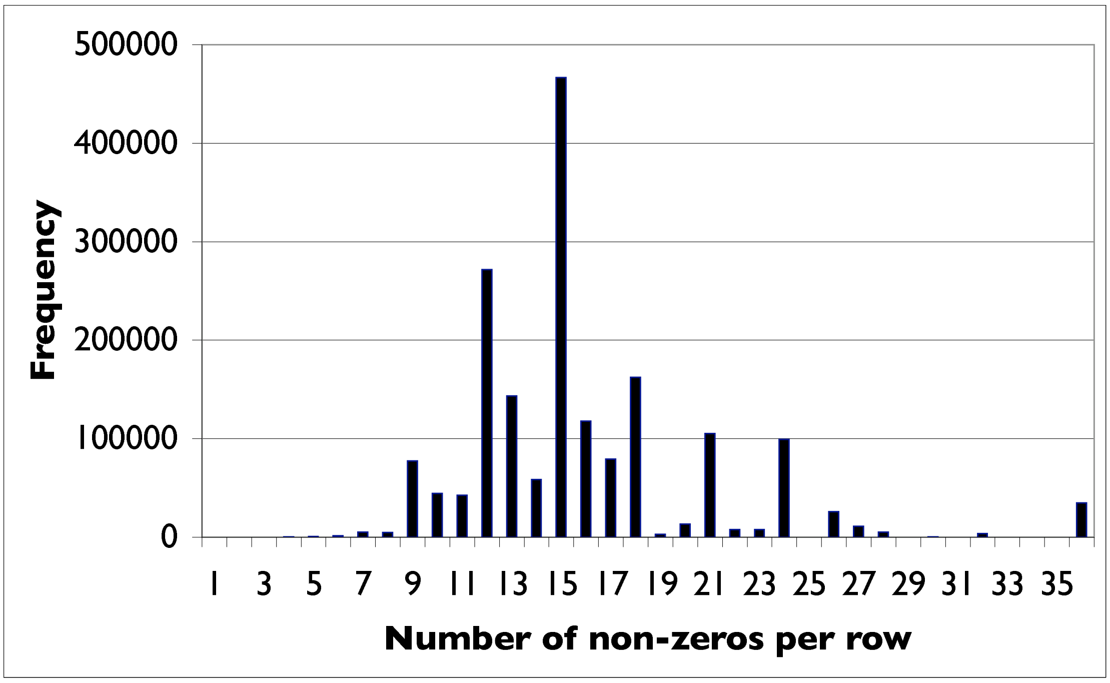
The results where measured on a dedicated 32 CPU domain of a SF15K [15] and on a 32 processor SUN E10K server [14]. The SF15K is a cc-NUMA system consisting of 4 CPU nodes (cpu boards) connected together using a crossbar switch. On the system used, each UltraSPARC-IIICu [67] is clocked at 900MHz and had a 64Kb L1 data cache, a 8Mb L2 cache and a single CPU is capable of addressing up to 16Gb of memory at 2.4 Gb/s. The domain had a total DRAM size of 32 Gb, where each 4 CPU node had 4 Gb of local memory on the board. The SUN E10K server is a uniform memory access machine of SMP type consisting of 32 UltraSPARC-II CPUs clocked at 400MHz. The UltraSPARC-II has a 8Mb L2 cache and a 16Kb L1 cache. The total DRAM size is 32Gb. All source codes where compiled using the Sun ONE Studio 8 compiler.

C.4.2. The FEM solver

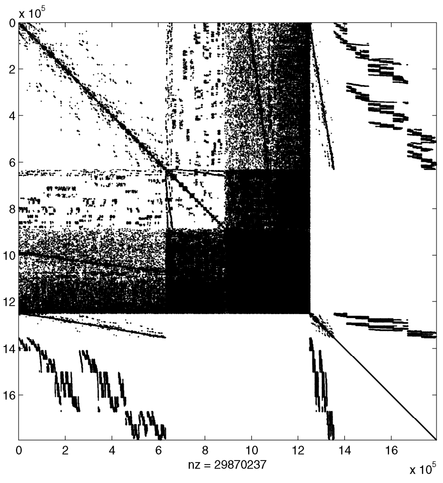
The studied solver solves the Maxwell equations in electro-magnetics around a fighter jet configuration [24] in 3D using the finite element method (FEM). The system of equations has 1794058 unknowns, the non-zero density is only 0.0009% (see Figure C.1) and the solver converge in 48 iterations without any preconditioning. Figure C.1(a) shows a histogram of the number of non-zeros per row. The maximum number of non-zeros for all rows was 36 and on average each row has about 15 non-zero elements. After initialization the resident working set of the applications was about 500Mb.

The main part of the code was implemented in Fortran90. The SpMxV however was implemented in C, to support more aggressive optimizations by the compiler. We do not run the entire FEM solver framework. Instead we have saved the iteration matrix to file and isolated the iterative solver. Hence, the matrix is read from file by the master thread which, by the first-touch principle, allocates all memory in the node that the master thread is running in. A similar situation might occur if we had used the entire FEM framework as it is hard to do the FEM assembly process in a way that distributes data according to the first touch principle. A more elegant way is to use what ever code you have for the assembly process and let the iterative solver worry about data distribution. As the sparse matrix is stored in the compressed sparse row (CSR) format, we cannot use some kind of pre-iteration to do data distribution. This is because the access pattern is determined indirectly from the CSR arrays that need to be loaded into memory. Hence, page migration seems like a solution given that the overhead of migration can be amortized. This is discussed further in Löf et al. [44, 42]. In the following experiments, the page migration directives (calls to `madvise(3C)`) was inserted at the beginning of the first iteration just before the SpMxV when running on the SF15K machine. By using migration data is migrated, by redoing the first touch, according to the access pattern of the solver.

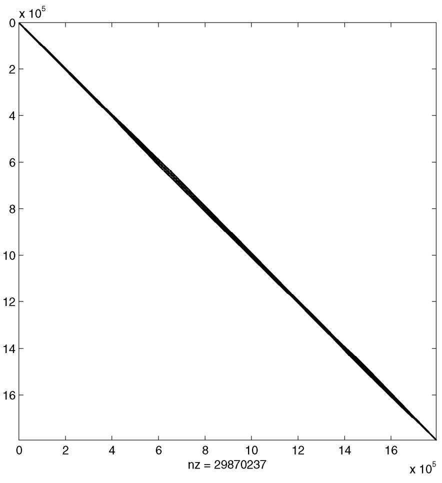
To control thread affinity, the Solaris scheduler was told to try and keep threads in their “home” node. This is especially important when we do data placement (migrate-on-next-touch mode). In practice, the Solaris 9 scheduler assign the home node to spawned threads in a way so that the node is not overloaded. As an example, if the node has 4 CPUs, the first three threads created will be assigned to this node (this will be their home node) and the fourth thread will be assigned to another un-loaded node



(a) Histogram of the non-zero structure of the input matrix



(b) Base



(c) Bandwidth minimization

Figure C.1.: Structure of the iteration matrix used for the four different cases of reorderings. The system has 1794058 unknowns

if there is one available. In our experiments we did not use any explicit CPU bindings. Hence, the scheduler can move threads away from their home node to create a better load balance from a system perspective.

C.4.3. Experimental setups

We constructed five different combinations of reorderings and algorithms

Base This is the standard OpenMP implementation using the original input matrix, see Figure C.1(b) and Algorithm 6.

S-Step This is an implementation of the S-Step algorithm, see Section C.3.2 and Algorithm 8 using the original data, see Figure C.1(b)

GPS Here, we use the standard CG algorithm but we have reordered the orginal matrix using the Gibbs-Pool-Stockmeyer (GPS) algorithm, see Section C.3.1. The structure of the resulting reordered matrix can be found in Figure C.1(c)

OPT Here, we use the optimized CG algorithm, see Algorithm 7

MeTiS Here, we use the optimized algorithm, see Algorithm 7 and the matrix is reordered according to the partitions produced by MeTiS as described in Section C.3.3

GPS+OPT Here, we use the optimized CG algorithm, see Algorithm 7 on the matrix reordered by the GPS method

GPS+MeTiS Here, the matrix reordered using the GPS method is reordered again according to the partitions produced by MeTiS as described in Section C.3.3

GPS+S-Step In this setting we use the S-Step algorithm on the matrix reordered by the GPS algorithm

C.4.4. Evaluating the effect of the algorithmic optimizations

We have performed the experiments using hardware counters to quantify the effects of the optimizations. To quantify the load balance we measured the total CPU time spent in the `__mt_EndOfTask_Barrier_` and `__mt_Explicit_ Barrier_` functions of the Sun OpenMP runtime system called from within the main parallel region. This corresponds to the CPU time spent in global barriers in the implemented algorithm. The `__mt_Explicit_Barrier_` function is not due to any `BARRIER` directives. This function is called by the Sun runtime system at the end of the `SINGLE` directive. We also measured the executed cycles and instructions to calculate the number of Instructions Per Cycle (IPC) and we measured the impact on the memory subsystem by counting L2-cache references, misses and remote misses using the UltraSPARC-IIICu hardware counters [67]. This allows us to calculate L2 miss ratios as well as the distribution of the L2 misses into the remote or local categories. We also counted the number of completed floating-point instructions from the CPU. Using these counters we could calculate FLOP/s rates.

All detailed measurements were made using the Sun collect/analyzer profiling tool set using 16 threads. In the case of the SF15K we also used page migration (migrate-on-next-touch mode) and 64Kb pages, see [42]. For the performance graphs, the execution times were measured using the Solaris high resolution timer `gethrtime(3C)`. Each binary was run ten times and we choose the minimum to represent the execution time of the binary. The minimum time is used to reduce the variations caused by thread scheduling and intrusion of system deamons. We measure the arithmetical load imbalance γ defined as the maximum number of non-zeros per partition divided by the mean number of non-zeros per partition. As the number of operations needed to produce an entry in the solution is dependent on the number of non-zeros in the partition, the *arithmetical load balance* can be described using the total number of non-zeros per partition. True load balance is a combination of the arithmetical load balance and high throughput in the computing hardware. Even though the partitions have balanced number of non-zeros the actual time to compute is very dependent on how much the CPU stalls on cache misses. This is especially true for non-uniform architectures. Also, in the case of OpenMP, non-uniform memory systems might affect the time spent in global barriers.

C.5. Experimental Results

From Figures C.2 and C.3, we can see the effect of the different algorithmic optimizations. The bars are normalized to the corresponding `Base` case (the left-most bar in each group) which corresponds to the original code with no optimizations used. We can clearly see that bandwidth minimization gives the highest performance improvement both on NUMA and the uniform SMP machine. Reordering with MeTiS has some effect on the SMP machine but this declines with increasing number of threads. Reducing the number of barriers has no significant effect alone but in combination with bandwidth minimization the peformance is further improved on both machines.

Looking at Figures C.4 and C.5, we see that we can achieve perfect scalability on the SMP machine using the different algorithmic optimization techniques while on the NUMA machine we only reach about 50% parallel efficency on 28 threads. To understand the effects of the different optimization techniques we have made a detailed analysis of a 16 thread run (Figure C.6, Table C.1, Table C.2). Figure C.6 visualizes the arithmetical load balance. Here the “Reference” ring corresponds to a static equal size partitioning of the non-zeros, ie perfect arithmetical load balance. This can however not be achieved since it does not correspond to any valid reordering. Comparing the sectors at the innermost ring, the Base ring, at 12 and 6 o‘clock of Figure C.6 we can see that the static partitioning does not produce very balanced partitions for this realistic data set. This is also quantified by the γ parameter. A simple linear partitioning of rows in the bandwidth minimized matrix gives a very good load balance, better than using graph partitioning. A primary goal of MeTiS is to minimize the edge-cut. Experimenting with different algorithms for refinement and edge-matching in MeTiS did not have any significant effect on runtime performance. Looking at the results from the

	BASE	S-Step	GPS	MeTiS	OPT
Load balance (γ)	1.24	1.24	1.01	1.15	1.24
User CPU time	336.66s	328.44s	234.57s	299.71	323.71s
System time	0.4s	0.45s	0.5s	0.07s	0.33s
SpMxV	213.79s	211.19s	190.29s	224.74s	212.43s
Barrier	85.02	79.72s	10.84s	46.66s	83.7s

Table C.1.: Detailed analysis using the Sun analyzer profiling tool. All results are for 16 threads on the E10K. On this system, the analyzer could not use hardware counters.

more detailed measurements in Table C.1 and Table C.2 we can confirm that the algorithm is memory-bound with a low instruction per cycle rate (IPC). Hence, bandwidth minimization that reduces both the L2 misses and the remote memory accesses most is expected to give best performance improvements. MeTiS reduces the total edge-cut and the load imbalance but gives a high number of remote memory accesses increasing the time spent in SpMxV. MeTiS does not consider that the processors are clustered four by four in the nodes. The different optimizations to reduce the number of barriers do not give any significant improvements. For this small number of threads the time spent in barriers is mostly due to load imbalance in the SpMxV operation and not in the algorithm to synchronize the threads. Thus, reducing the load imbalance gives the best effect also on barrier time. For larger number of threads we would expect to gain more in reducing the number of barriers. On the NUMA system the time spent in barriers (20-40%) is still very high for all optimizations. This is also causing the poor scaling. We would expect that the GPS+OPT or GPS+S-step with good load balance and few barriers would give low barrier times but this is not the case on the NUMA system. This suggest that the reductions may not be optimal here. For the same optimizations we get a barrier time below 5% of total time on the SMP system.

C.6. Conclusions

We have investigated ways of improving the performance of an industrial class conjugate gradient (CG) solver, implemented in OpenMP running on two types of shared memory systems, a uniform SMP and a cc-NUMA machine. We have evaluated bandwidth minimization, graph partitioning and reformulations of the original algorithm reducing global barriers. Bandwidth minimization has the greatest impact on performance, reducing both the L2 misses and remote memory accesses. Graph partitioning

	BASE	S-Step	GPS	MeTiS	OPT
Load balance (γ)	1.24	1.24	1.01	1.15	1.24
User CPU time	131.29s	127.7s	74.85s	130.11s	123.01s
System time	4.77s	6.51s	6.78s	6.23s	4.3s
SpMxV	59.45s	57.4s	36.23s	70.02s	55.67s
Barrier	45.15s	48.29s	21.67s	33.25s	47.5s
IPC	0.63	0.57	0.65	0.47	0.55
L2 misses 10^6	427	421	376	438	418
Remote misses 10^6	125	115	70	144	104
MFLOPS/s	450.2	463.4	710.7	451.9	450.8

Table C.2.: Detailed analysis using the Sun analyzer profiling tool and hardware counters. All results are for 16 threads on the SF15K.

improves the load balance and edge-cut of the original matrix but does not give the same load balance and data locality as bandwidth minimization. Reducing the number of barriers alone has no significant effect on these small systems but gives some performance improvements in combination with the other optimizations when the load imbalance is reduced. On the uniform SMP system we can achieve perfect scaling using the different optimization techniques but on the cc-NUMA system we have less good scalability due to high barrier times coming from the reduction operations.

Our conclusion is that it is possible to achive high performance and still keep the software engineering efforts low by using an architecture independent shared memory model like OpenMP. But, special care have to be taken to algorithmic optimizations, data distributions and implementation of global operations. The bottleneck in our application on the NUMA system is the global reduction operations.

C.7. Implementations in OpenMP

Here we present our implementations of th CG algorithm in OpenMP. The standard implementation, Algorithm 6 represents a direct implementation from the CG algorithm. In Algorithm 7 we have used features of OpenMP to reorder some calculations to save global barriers. Finally, Algorithm 8 is an implementation of the S-Step version of CG. Here only 2 global barriers are needed. To achieve this an extra vector is introduced, resulting in more arithmetical work.

Algorithm 6 Standard OpenMP implementation of CG

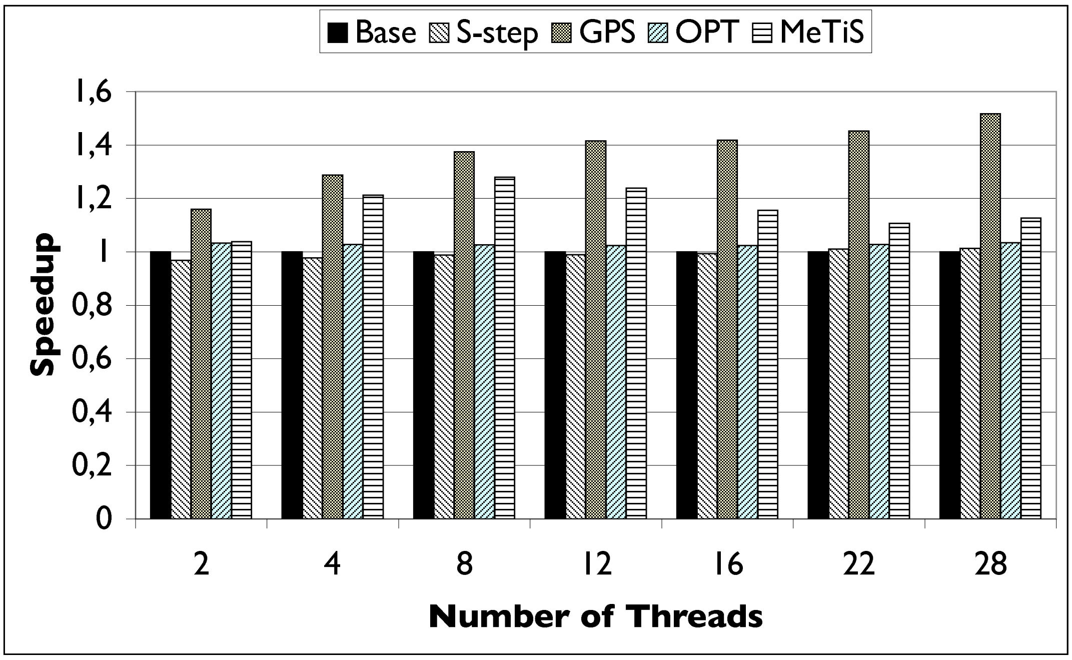
```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(cg_iter_count)

    cg_iter_count = 1

   do

       call SpMxV(A,p,temp) ! No barrier here

!$OMP SINGLE
       pAp_norm = 0.0_rfp
!$OMP END SINGLE
====================================
!$OMP DO REDUCTION(+:pAp_norm)
       do i = 1, matrix_size
          pAp_norm = pAp_norm + p(i)*temp(i)
       end do
!$OMP END DO
====================================

!$OMP SINGLE
       alpha = r_old_norm/pAp_norm
!$OMP END SINGLE
====================================
!$OMP WORKSHARE
       x = x + alpha * p
       r_new = r_old - alpha * temp
!$OMP END WORKSHARE NOWAIT

!$OMP SINGLE
       r_new_norm = 0.0_rfp
!$OMP END SINGLE
====================================
!$OMP DO REDUCTION(+:r_new_norm)
       do i = 1, matrix_size
          r_new_norm = r_new_norm + r_new(i)*r_new(i)
       end do
!$OMP END DO
===================================

!$OMP SINGLE
       beta = r_new_norm/r_old_norm
!$OMP END SINGLE
===================================
!$OMP WORKSHARE
       p = r_new + beta*p
!$OMP END WORKSHARE
===================================

      Check convergence

!$OMP SINGLE
       call Swap_pointers(r_old, r_new)
       r_old_norm = r_new_norm
!$OMP END SINGLE NOWAIT

       cg_iter_count = cg_iter_count + 1

   end do
!$OMP END PARALLEL
```

Algorithm 7 Optimized OpenMP implementation of CG

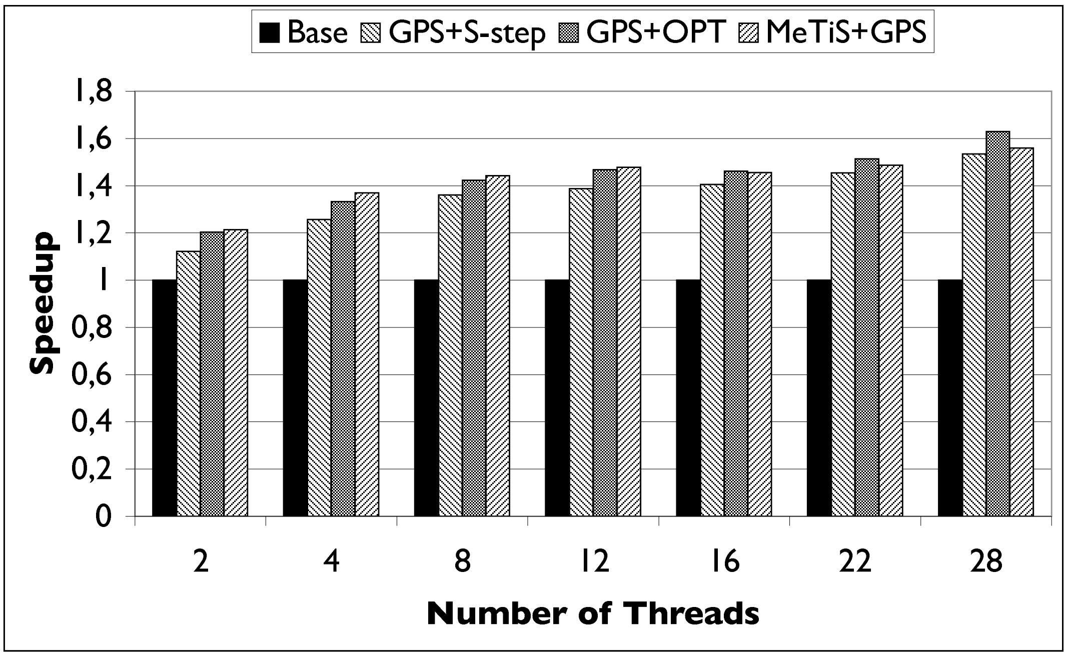
```
   pAp_norm = 0.0_rfp

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(cg_iter_count,alpha,beta)

    cg_iter_count = 1

   do

       call SpMxV(A,p,temp) ! No barrier here

!$OMP SINGLE
       r_old_norm = r_new_norm
       r_new_norm = 0.0_rfp
!$OMP END SINGLE NOWAIT

!$OMP DO REDUCTION(+:pAp_norm)
       do i = 1, matrix_size
          pAp_norm = pAp_norm + p(i)*temp(i)
       end do
!$OMP END DO
====================================

       alpha = r_old_norm/pAp_norm

!$OMP WORKSHARE
       x = x + alpha * p
       r_new = r_old - alpha * temp
!$OMP END WORKSHARE NOWAIT

!$OMP SINGLE
       pAp_norm = 0.0_rfp
!$OMP END SINGLE NOWAIT

!$OMP DO REDUCTION(+:r_new_norm)
       do i = 1, matrix_size
          r_new_norm = r_new_norm + r_new(i)*r_new(i)
       end do
!$OMP END DO
=====================================

       beta = r_new_norm/r_old_norm

!$OMP WORKSHARE
       p = r_new + beta*p
!$OMP END WORKSHARE
====================================


      Check convergence

!$OMP SINGLE
       call Swap_pointers(r_old, r_new)
!$OMP END SINGLE NOWAIT

       cg_iter_count = cg_iter_count + 1

   end do
!$OMP END PARALLEL
```

Algorithm 8 OpenMP implementation of S-step CG

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(idx, cg_iter_count, alpha, beta)

    cg_iter_count = 1
    alpha = r_old_norm / my
    beta = 0.0_rfp

    do

!$OMP WORKSHARE
          p = r + beta * p
          q = temp + beta * q
!$OMP END WORKSHARE NOWAIT

!$OMP SINGLE
       r_old_norm = r_new_norm
       r_new_norm = 0.0_rfp
       my = 0.0_rfp
!$OMP END SINGLE NOWAIT

!$OMP WORKSHARE
          x = x + alpha * p
          r = r - alpha * q
!$OMP END WORKSHARE
===================================

      call SpMxV(A,r,temp) ! No barrier here

!$OMP DO REDUCTION(+:r_new_norm,my)
       do idx = 1, matrix_size
          r_new_norm = r_new_norm + r(idx) * r(idx)
          my = my + r(idx) * temp(idx)
       end do
!$OMP END DO
====================================

       Check convergence

       beta = r_new_norm/r_old_norm
       alpha = r_new_norm/(my - ((beta*r_new_norm)/alpha))
       cg_iter_count = cg_iter_count + 1

    end do

!$OMP END PARALLEL
```

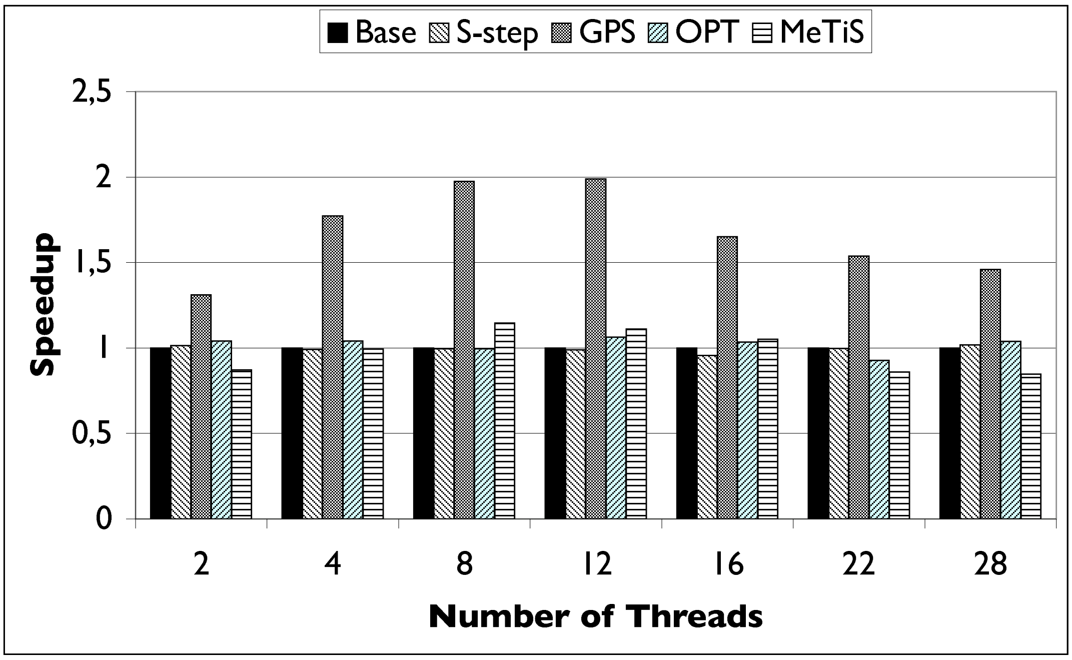


(a) Speedup of basic optimizations on the E10K

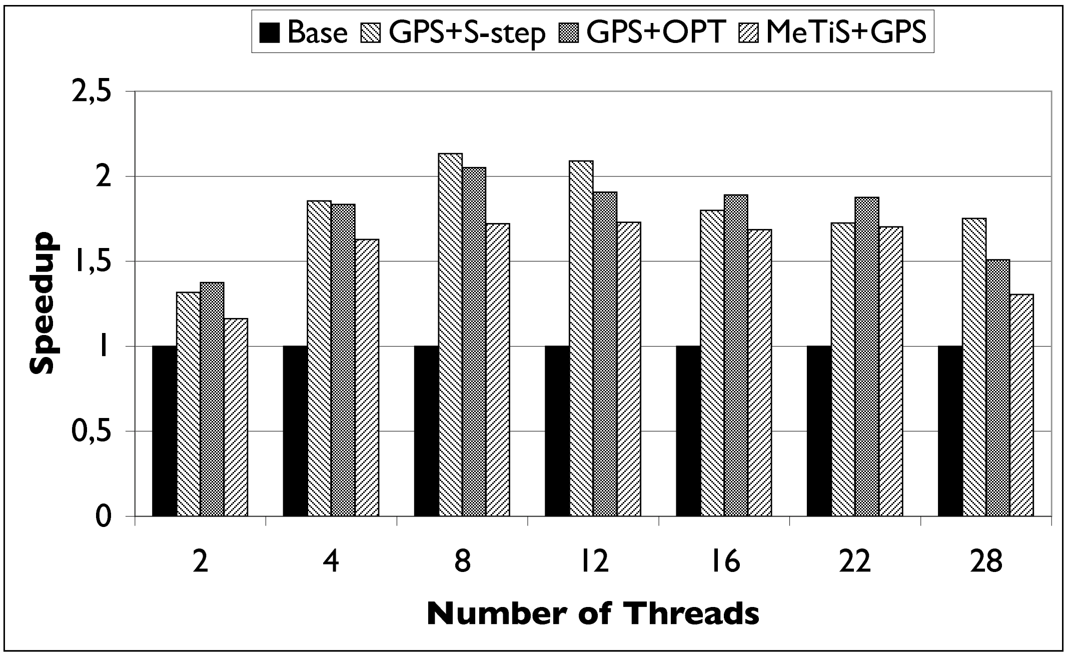


(b) Speedup of combinations of optimizations on the E10K

Figure C.2.: Speedup of algorithmic optimizations on the E10K. The execution times are normalized to the `Base` case for each set of threads.

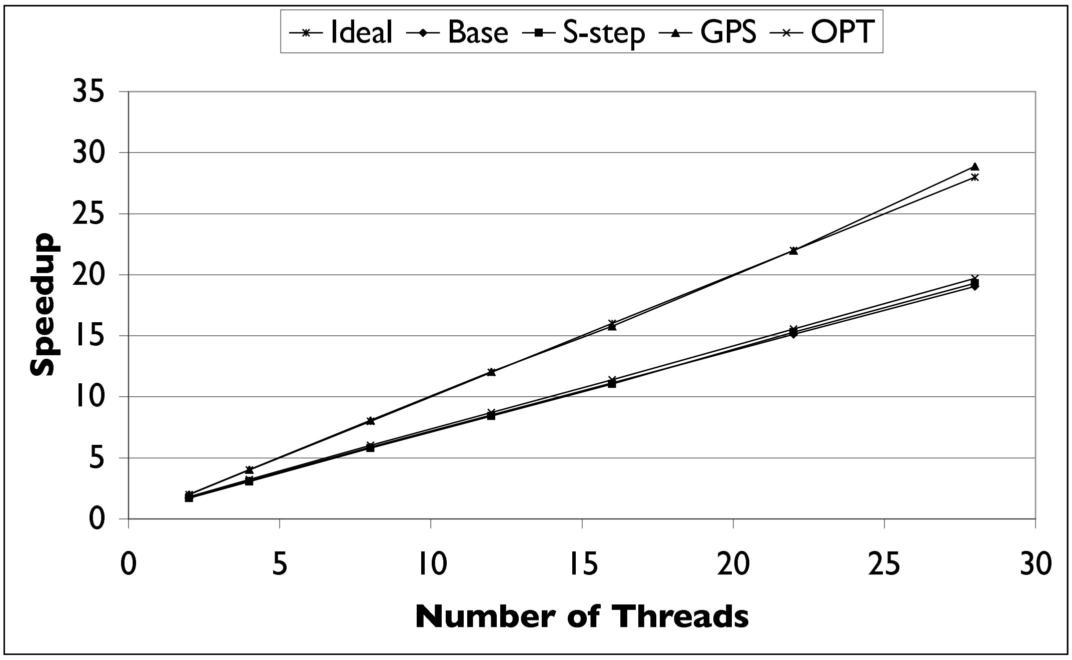


(a) Speedup of basic optimizations on the SF15K

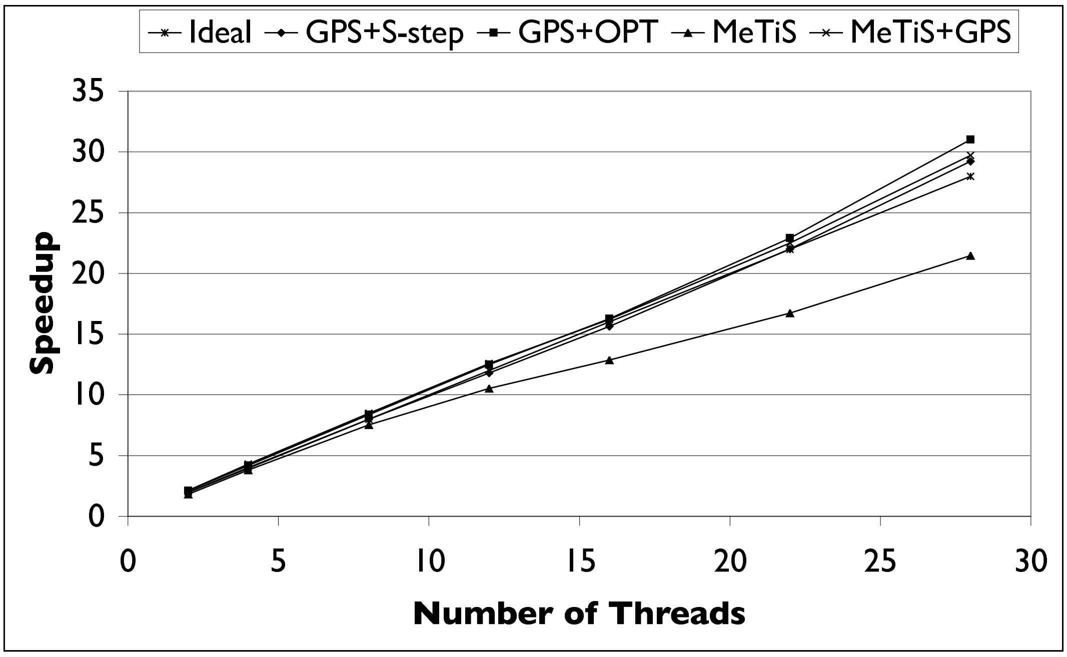


(b) Speedup of combinations of optimizations on the SF15K

Figure C.3.: Speedup of algorithmic optimizations on the SF15K. The execution times are normalized to the `Base` case for each set of threads.

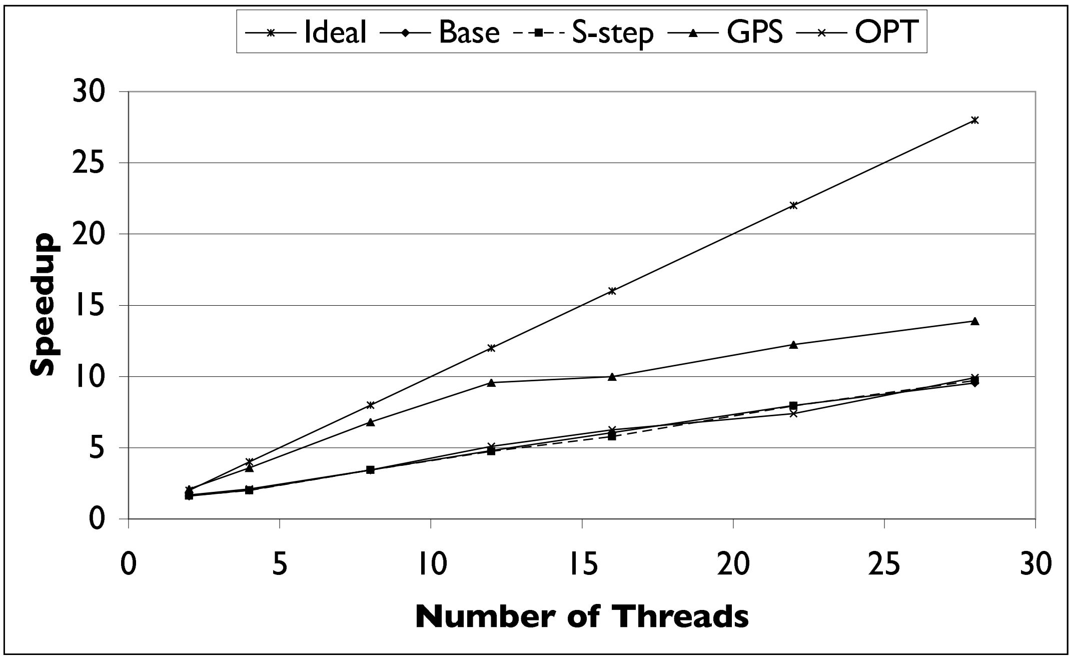


(a) Speedup of basic optimizations on the E10K

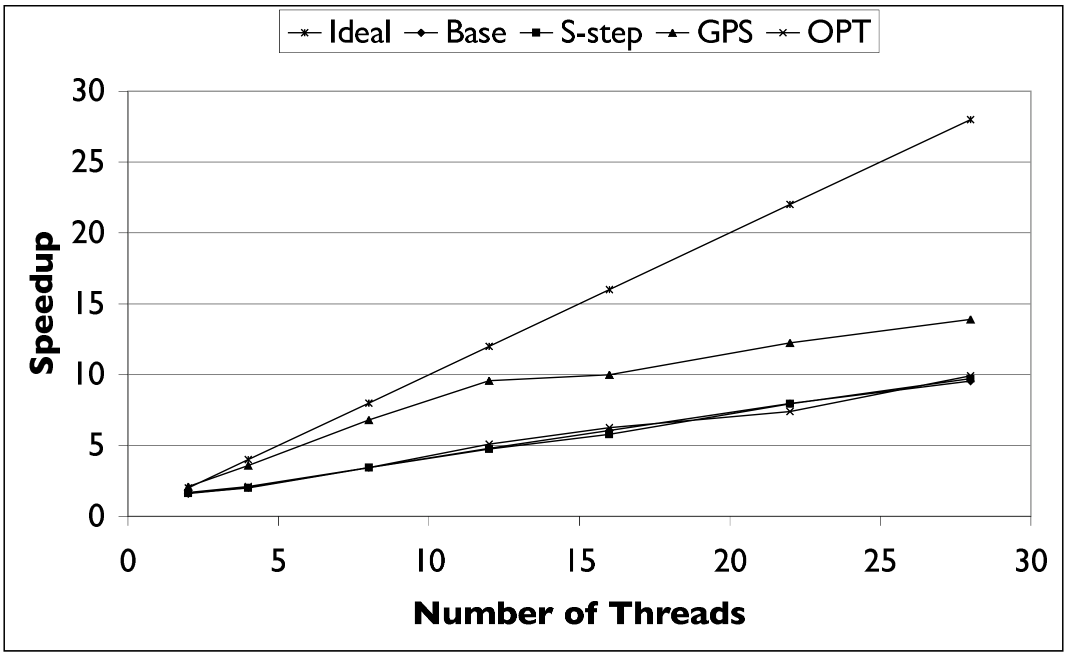


(b) Speedup of combinations of optimizations on the E10K

Figure C.4.: Speedup of algorithmic optimizations on the E10K. The execution times are normalized to a serial code using bandwidth minimizaton.



(a) Speedup of basic optimizations on the SF15K



(b) Speedup of combinations of optimizations on the SF15K

Figure C.5.: Speedup of algorithmic optimizations on the SF15K. The execution times are normalized to a serial code using bandwidth minimizaton.

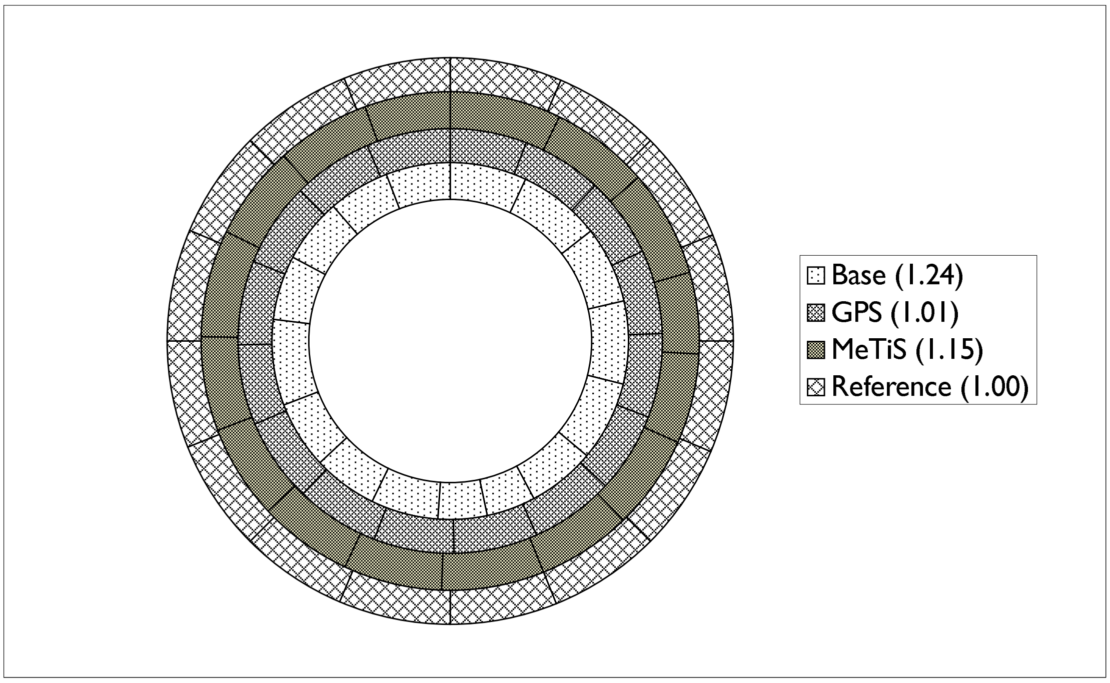


Figure C.6.: Visualization of the arithmetical load balance for the case of 16 partitions (threads). The reference ring corresponds to perfect balance which is impossible to generate using reorderings. The number in parenthesis in the legend is the γ parameter defined in Section C.4.4

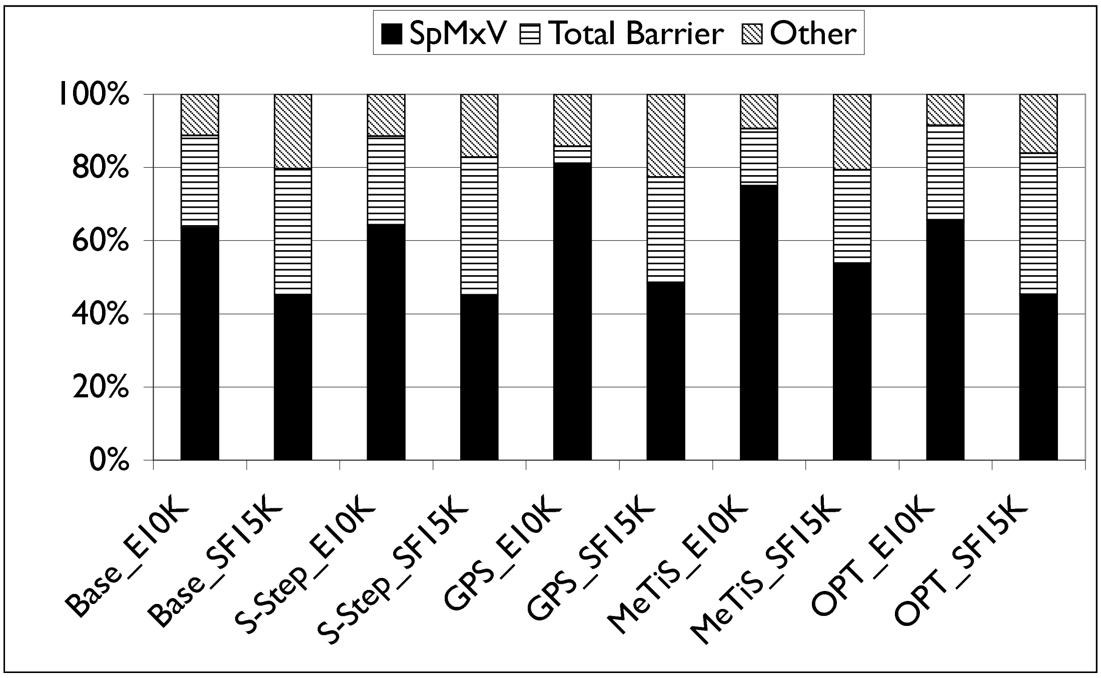


Figure C.7.: Distribution of CPU time

Paper D

D. THROOM - Supporting POSIX Multithreaded Binaries on a Cluster

Henrik Löf, Zoran Radović and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
henrik.lof,zoran.radovic,erik.hagersten@it.uu.se

Abstract

Today, most software distributed shared memory systems (SW-DSMs) lack industry standard programming interfaces which limit their applicability to a small set of shared-memory applications. In order to gain general acceptance, SW-DSMs should support the same look-and-feel of shared memory as hardware DSMs. This paper presents a runtime system concept that enables unmodified POSIX (Pthreads) binaries to run transparently on clustered hardware. The key idea is to extend the single process model of multi-threading to a multi-process model where threads are distributed to processes executing in remote nodes. The distributed threads execute in a global shared address space made coherent by a fine-grain SW-DSM layer. We also present THROOM, a proof-of-concept implementation that runs unmodified Pthread binaries on a virtual cluster modeled as standard UNIX processes. THROOM runs on top of the DSZOOM fine-grain SW-DSM system with limited OS support.

D.1. Introduction

Clusters built from high-volume compute nodes, such as workstations, PCs, and small symmetric multiprocessors (SMPs), provide powerful platforms for executing large-scale parallel applications. Software distributed shared memory (SW-DSM) systems can create the illusion of a single shared memory across the entire cluster using a software run-time layer, attached between the application and the hardware. In spite of several successful implementation efforts [58], [23], [39], [65], [55], SW-DSM systems are still not widely used today. In most cases, this is due to the relatively poor and unpredictable performance demonstrated by the SW-DSM implementations. However, some recent SW-DSM systems have shown that this performance gap can be narrowed by removing the asynchronous protocol overhead [55], [5], and demonstrate a performance overhead of only 30-40 percent in comparison to hardware DSMs (HW-DSM) [55]. One obstacle for SW-DSMs is the fact that they often require special constructs and/or impose special programming restrictions in order to operate properly. Some SW-DSM systems further alienate themselves from HW-DSMs by relying heavily on very weak memory models in order to hide some of the false sharing created by their page-based coherence strategies. This often leads to large performance variations when comparing the performance of the same applications run on HW-DSMs. We believe that, SW-DSMs should support the same look-and-feel of shared memory as the HW-DSMs. This includes support for POSIX [34] threads running on some standard memory model and a performance footprint similar to that of HW-DSMs, i.e., the performance gap should remain approximately the same for most applications. Our goal is that binaries that run on HW-DSMs could be run on SW-DSMs, without modifications.

In contrast to a HW-DSM system, where the whole address space of all processes are kept coherent by hardware, most SW-DSMs only keep coherence for specified segments in the user-level part of the virtual address space. This segment, which we call G_MEM, is mapped shared across the DSM nodes using the interconnect hardware. Furthermore, the text (program code), data and stack segments of the UNIX process abstraction are private to the parent process and its children on each node of the cluster. This creates a SW-DSM programming model where special constructs are needed to separate shared data, which must be allocated in G_MEM, from private data, which is allocated in the data and stack segments of the UNIX process at program loading. This is often done by creating a separate heap space in G_MEM with an associated primitive for doing allocation. In a standard multi-threaded world, there exist only one process and one address space which is shared among all threads. There is no distinction between shared and private data. Consider the following example: An application allocates a shared global array for its threads to operate on. This is often done by a single thread in an initialization phase. In a typical SW-DSM system such as TreadMarks [39], a special `malloc()`-type call has to be implemented to allocate the memory for the shared array inside the G_MEM. Also, the pointer variable holding the address, which is allocated in the static data segment of the process, has to be propagated to all remote nodes. This is often done by introducing a special propagation primitive.

In this paper, we present THROOM which is a runtime system concept that creates the illusion of a single process shared memory abstraction on a cluster. In essence, we want to make the static data and heap segments globally accessible by threads executing in remote nodes without introducing special DSM constructs in the application code. In the light of the example above, the application should use a standard `malloc()` call and the pointer variable should be replicated automatically. The rest of this paper is organized as follows: First, the THROOM concept is presented. Second, we give a brief presentation of the SW-DSM used. We also specify the requirements of THROOM on the SW-DSM. Third, we present a proof-of-concept implementation of THROOM on a single system image cluster and finally we discuss the performance of this implementation as well as the steps needed to take THROOM to a real cluster.

D.2. The THROOM concept

In many implementations of SW-DSMs, the different nodes of the cluster all run some daemon process to maintain the G_MEM mappings and to deal with requests for coherency actions. In this paper we use the term *user node* to refer to the cluster node in which the user executes the binary (the *user process*). All other nodes are called *remote nodes* and their daemon processes will be called *shadow processes*. In execution, THROOM consists of one process per node of the cluster.

The fundamental idea of THROOM is to distribute threads from the user process to shadow processes executing on remote nodes running different instances of a standard UNIX OS kernel. As discussed earlier, such systems exists, but they require non-standard programming models. To support a standard model such as POSIX, it is required that the whole address space of the user process can be accessed by all of the distributed threads. To accomplish this, we can simply place the text, data and stack segments inside a G_MEM-type segment made coherent by a SW-DSM. This will create the illusion of a large scale shared memory multiprocessor built out of standard software and hardware components.

D.3. DSZOOM - a Fine-Grained SW-DSM

Our prototype implementation is based on the sequentially consistent DSZOOM SW-DSM [55]. Each DSZOOM node can either be a uniprocessor, a SMP, or a CC-NUMA cluster. The node's hardware keeps coherence among its caches and its memory. The different cluster nodes run different kernel instances and do not share memory with each other in a hardware-coherent way. DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory in other nodes, similar to the remote `put/get` semantics found in the cluster version of the Scalable Coherent Interface (SCI), or the emerging InfiniBand standard that supports `RDMA READ/WRITE` as well as the atomic operations `CmpSwap` and `FetchAdd` [35].[1] Another example is the Sun Fire (TM) Link interconnect hardware [64].

While traditional page-based SW-DSMs rely on TLB traps to detect coherence "violations", fine-grained SW-DSMs like Shasta [59], Blizzard-S [61], Sirocco-S [60] and DSZOOM [55] insert the coherence checks in-line. In DSZOOM, this is done by replacing each load and store that may reference shared data of the binary with a code *snippet* (short sequence of machine code). In terms of THROOM, the only requirement on the SW-DSM system is that it uses binary instrumentation. THROOM will also inherit the memory consistency model of the SW-DSM system.

[1] Atomic operations are needed to support a *blocking directory* protocol [55].

D.4. Implementing THROOM

This section discuss how we can implement the THROOM concept using standard software components and the DSZOOM SW-DSM.

D.4.1. Achieving transparency

The most important aspect of THROOM is that it is totally transparent to the application code, no recompilation is allowed. To achieve this, we use a technique called *library interposition* or *library pre-loading* [68], which allow us to change the default behavior of a shared library call without recompiling the binary. Many operating systems implement the core system libraries such as `libc`, `libpthread` and `libm` as shared libraries. Using inter-positioning, we can catch a call, to any shared library and redirect it to our own implementations. In practice, this is done by redefining a symbol in a separate shared library to be pre-loaded at runtime. When an application calls the function represented by the symbol, the runtime linker searches its path for a match. Pre-loading simply means that we can insert an alternate implementation before the standard implementation in the search path of the linker. Pre-loading also allow us to reuse the native implementation. Original arguments can be modified in the interposer before the call to the native implementation is made.[2]

D.4.2. Distributing Threads

To distribute threads, the `pthread_create()` call is redefined in a pre-loaded library. The interposed implementation, first schedules the thread for execution in a remote shadow process. Second, the chosen shadow process is told to create a new thread, by calling the native `pthread_create()` from within the interposing library. The new distributed thread will start to execute in the shadow process, with arguments pointing to the software context of its original user process.

D.4.3. Creating a Global Shared Address Space

A minimal requirement for a distributed thread to execute correctly in a shadow process is that it must share the whole address space of the user process. To accomplish this, the `malloc()` call is redefined in a pre-loaded library to allocate memory from G_MEM instead of the original data segment of the user process. This will make all dynamically allocated data accessible from the shadow processes. Code and static data are made globally accessible by copying the segments containing code and static data from the user process to the G_MEM. The application code is then modified, using binary instrumentation, to access the G_MEM copy instead of the original segments. This will make the application execute entirely in the global shared memory segment.

[2]To our knowledge, Linux, Solaris, HP-UX, IRIX and Tru64 all support library pre-loading.

Hence, no special programming constructs are needed to propagate writes to static data. The whole process is also transparent in the sense that a user does not need access to the application source code, as binary instrumentation modifies the binary itself.

All references to the G_MEM must also be made coherent as the hardware only support remote reads and writes. This is taken care of by a fine-grain SW-DSM. If the SW-DSM use binary instrumentation to insert snippets for access control, we can simply add instructions needed for the static data access diversion to these snippets. In all cases, a maximum of four instructions were added to the existing snippets of DSZOOM. To lower the overheads associated with binary instrumentation, the present implementation does not instrument accesses to the stack. Hence stacks are considered thread private. Although this is not in full compliance with the POSIX model of multi-threading, it is sufficient to support a large set of pthread applications.

D.4.4. Cluster-Enabled Library Calls

Most applications use system calls and/or calls to standard shared libraries such as `libc`. If the arguments refer to static data, the accesses must be modified to use the G_MEM in order for memory operations to be coherent across the cluster. This can be done in at least two ways. We either instrument all library code or we overload the library calls to copy any changes from the user process original data segments to the G_MEM copies at each library call. Remember that un-instrumented code referencing static data of the application will operate in the original data segments of the user process. Hence, copying is needed to make any modifications visible to other nodes.

Instrumenting all library code is in principle, the best way to cluster-enable library calls. However, our instrumentation tool, EEL [40], was not able to instrument all of the libraries. Instead, we had to use the library interposition method for our prototype implementation. An obvious disadvantage of this method is that we have to redefine a large amount of library calls, especially if we want complete POSIX support. Another disadvantage is the runtime overhead associated with data copying, especially for I/O operations. A better solution would be to generate the coherence actions on the original arguments before the call is made in the application binary, see Scales et. al. [58]. This requires a very sophisticated instrumentation tool, which is outside the scope of this work.

D.5. Implementation Details

We have implemented the THROOM system on a 2-node Sun WildFire prototype SMP cluster [32], [31]. The cluster is running a single-system image version of Solaris 2.6 and the hardware is configured as a standard CC-NUMA architecture. Although, this system already supports a global shared address space, we can still use it to emulate a future THROOM architecture.

The runtime system is implemented as a shared library. A user simply sets the `LD_PRELOAD` environment variable to the path of the THROOM runtime library, and then executes the instrumented binary. As the system is a single system image we can use standard Inter Process Communication (IPC) primitives to emulate a real distributed cluster. The DSZOOM address space is set up during initialization using the `.init` section. This makes the whole initialization transparent. Control is then given to the application. The user process issues a `fork(2)` call to create a shadow process, which will inherit its parents mappings by the copy-on-write semantics of Solaris. The two processes are bound to the two nodes using the WildFire first-touch memory initialization and the `pset_bind()` call. The home process then reads its own `/proc` file system to locate the `.text`, `.data`, and `.bss` segments and copies them to the G_MEM.

The shadow process waits on a process shared POSIX conditional variable to create remote threads for execution in the G_MEM. Parameters are passed through a shared memory mapping separated from the G_MEM. Since the remote thread is created in another process, thread IDs are no longer unique. To fix this, the remote node ID is copied into the most significant eight bits of the thread type, which in the Solaris 2.6 implementation is an unsigned integer. Similar techniques are used for other pthread calls. Also, the synchronization primitives of the application were overloaded using pre-loading to pre-prepared `PROCESS_SHARED` POSIX primitives to allow for multi-process synchronization. More details on the implementation are available in Löf et al. [45].

D.6. Performance Study

First a set a test pthread programs were run to verify the correctness of the implementation. To produce a set of pthread programs to be used as a comparison to DSZOOM, ten SPLASH-2 applications [72] were compiled using the GCC v2.95.2 compiler without optimization (-O0)[3] and a standard Pthread PARMACS macro implementation (c.m4.pthreads.condvar_barrier) was employed. No modifications was made to the PARMACS run-time system or the applications. To exclude the initialization time for the THROOM runtime system, timings are started at the beginning of the parallel phase. All timings have been performed on the 2-node Sun WildFire [32] configured as a traditional CC-NUMA architecture. Each node has 16 UltraSPARCII processors running at 250 MHz. The access time to node-local memory is about 330ns. Remote memory is accessed in about 1800ns. In Table D.1, we see that more instructions are replaced in the case of THROOM since *all* references to static data have to be instrumented. This large difference in replacement ratio compared to DSZOOM is explained by the fact that DSZOOM can exploit the PARMACS programming model and use *program slicing* to remove accesses to static data that are not shared. Figure

[3]The code is compiled without optimization to eliminate any delay slots, which EEL cannot handle correctly.

Program	Problem size, Iterations	Replaced Loads (%)	Replaced Stores (%)
FFT	1 048 576 points (48.1 MByte)	44.6(19.0)	32.8(16.5)
LU-C	1024x1024, block 16 (8.0 MByte)	48.3(15.5)	23.0(9.4)
LU-NC	1024x1024, block 16 (8.0 MByte)	49.2(16.7)	27.7(11.1)
RADIX	4 194 304 items (36.5 MByte)	54.4(15.6)	31.4(11.6)
Barnes	16 384 bodies (8.1 MByte)	56.6(23.8)	55.4(31.1)
Ocean-C	514x514 (57.5 MByte)	50.6(27.0)	31.2(23.9)
Ocean-NC	258x258 (22.9 MByte)	51.0(11.6)	39.0(28.0)
Radiosity	room (29.4 MByte)	41.1(26.3)	35.1(27.1)
Water-NSQ	2197 mol, 2 steps (2.0 Mbyte)	50.4(13.4)	38.0(16.2)
Water-SQ	2197 mol, 2 steps (1.5 Mbyte)	48.5(15.7)	32.5(13.9)

Table D.1.: Problem sizes and replacement ratios for the 10 SPLASH-2 applications studied. Instrumented loads and stores are showed as a percentage of the total amount of load or store instructions. The number in parenthesis shows the replacement ratio for the DSZOOM SW-DSM without THROOM.

D.1 and D.2 shows execution times in seconds for 8- and 16-processor runs for the following THROOM configurations:

THROOM_RR THROOM runtime system using library pre-loading. Round-robin scheduling of threads between the two nodes. All references to static data are instrumented.

DSZOOM Used as reference. Aggressive slicing and snippet optimizations. Optimized for a two-node fork-exec native PARMACS environment, see [55].

CC-NUMA Uses the same runtime system as DSZOOM but without any instrumentations. Coherence is kept by the WildFire hardware [55]

A study of Figures D.1 and D.2 reveals that the present implementation is slower than a state-of-the-art SW-DSM such as DSZOOM. The average runtime overhead compared to DSZOOM for THROOM_RR is 65% on 8 CPUs and 78% on 16 CPUs. In order to put these numbers into the context of total SW overhead compared to a HW-DSM, the average slowdown comparing the CC-NUMA and the DSZOOM cases is only 26%. The most significant contribution to the high overhead when comparing DSZOOM to THROOM is the increased number of instrumentations needed to support the POSIX thread model. Another source of overhead is the inefficient implementation of locks and barriers. This can be observed by comparing the performance of Barnes, Ocean-C, Ocean-NC and Radiosity from Figures D.1 and D.2. The performance of these four applications drops when increasing the number of threads as they spend a significant amount of time executing in synchronization primitives. The DSZOOM runtime system uses its own implementations of spin-locks and barriers which are more scalable.

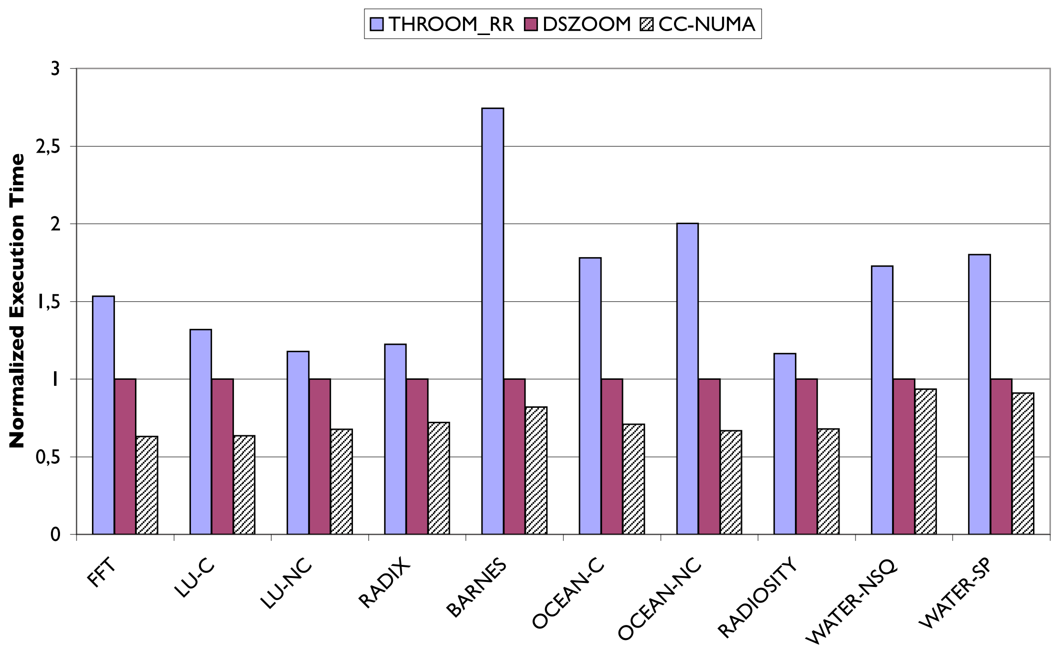


Figure D.1.: Runtime performance of the THROOM runtime system. Two nodes with 4 CPUs each.

D.7. Related work

To our knowledge, no SW-DSM system has yet been built that enables transparent execution of an unmodified POSIX binary. The Shasta system [58], [23] come closest to our work and this system has showed that it is possible to run an Oracle database system on a cluster using a fine-grain SW-DSM. Shasta has solved the OS functionality issues in a similar way as is done in THROOM although they support a larger set of system calls and process distribution. THROOM differs from Shasta in that it supports sharing of static data. THROOM also supports thread distribution. Shasta motivates the lack of multi-threading support by claiming that the overhead associated with access checks lead to lower performance [58].

Another system announced recently is the CableS system [36] built on the GeNIMA page-based DSM [5]. This system support a large set of system calls, but they have not been able to achieve binary transparency. Some source code modifications must be made and the code must be recompiled for the system to operate. Another work related to THROOM is the OpenMP interface to the TreadMarks page-based DSM [1] [39], where a compiler front-end translates the OpenMP pragmas into TreadMark fork-join style primitives. The DSM-Threads system [48] provide a page-based DSM interface similar to the Pthreads standard without binary transparency.

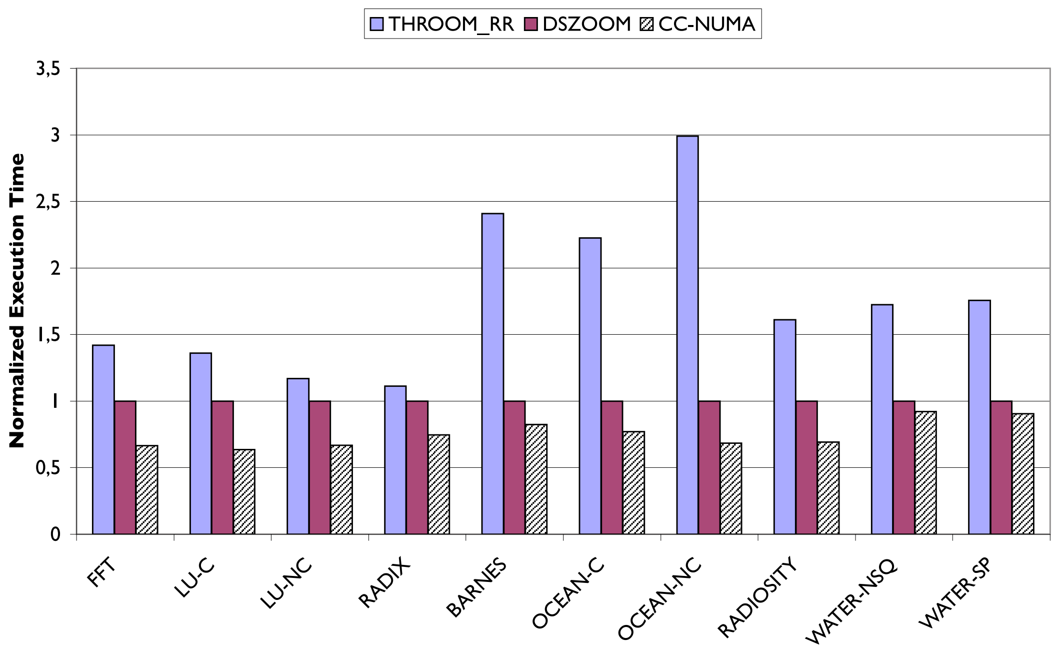


Figure D.2.: Runtime performance of the THROOM runtime system. Two nodes with 8 CPUs each.

D.8. Conclusions

We have showed that it is *possible* to extend a single process address space to a multi-process model. Even though the current THROOM implementation relies on some of the WildFire's single system image properties, we are convinced that the THROOM concept can be implemented on a real cluster. In a pure distributed setting, additional issues need to be addressed. One way of initializing the system could be to use a standard MPI runtime system for process creation and handshaking. The address space mappings must also be set up using the RDMA features of the interconnect hardware. Also, synchronization needs to be handled more efficiently (see Radović et. al. [56]), and we need to create more complete and more efficient support for I/O and other library calls. For complete POSIX compliance, we also need to address the problem of threads sharing data on the stack.

Bibliography

[1] A. Scherer, H. Lu, T. G., and Zwaenepoel, W. Transparent Adaptive Parallelism on NOWs using OpenMP. In *Principles Practice of Parallel Programming* (1999).

[2] Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture* (2000), ACM Press, pp. 248–259.

[3] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[4] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th annual international symposium on Computer architecture* (2000), ACM Press, pp. 282–293.

[5] Bilas, A., Liao, C., and Singh, J. P. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)* (May 1999).

[6] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. Extending OpenMP for NUMA machines. *Scientific Programming 8* (2000), 163–181.

[7] Blackford, L. S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., and Whaley, R. C. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2 (2002), 135–151.

[8] Brecht, T. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)* (San Diego, CA, Sept 1993).

[9] Brehm, J., and Jordan, H. F. Parallelizing algorithms for mimd architectures with shared memory. In *Proceedings of the 3rd international conference on Supercomputing* (1989), ACM Press, pp. 244–253.

[10] BULL, J. M., AND JOHNSON, C. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP* (2002), `http://www.caspur.it/ewomp2002/`.

[11] BURGESS, D. A., AND GILES, M. B. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Tech. rep., Oxford University Computing Laboratory, Numerical Analysis Group, May 1995.

[12] CHANDRA, R., CHEN, D.-K., COX, R., MAYDAN, D. E., NEDELJKOVIC, N., AND ANDERSON, J. M. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation* (1997), ACM Press, pp. 334–345.

[13] CHANDRA, R., DEVINE, S., VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems* (1994), ACM Press, pp. 12–24.

[14] CHARLESWORTH, A. Starfire: extending the SMP envelope. *IEEE Micro 18*, 1 (Jan/Feb 1998), 39–49.

[15] CHARLESWORTH, A. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (2001), ACM Press, pp. 7–7.

[16] CHRONOPOULOS, A., AND GEAR, C. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics 25* (1989), 153–168.

[17] CORBALAN, J., MARTORELL, X., AND LABARTA, J. Evaluation of the memory page migration influence in the system performance: the case of the sgi o2000. In *Proceedings of the 17th annual international conference on Supercomputing* (2003), ACM Press, pp. 121–129.

[18] CULLER, D., SINGH, J., AND GUPTA, A. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, 1998.

[19] CUTHILL, E., AND MCKEE, J. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference* (1969), ACM Press, pp. 157–172.

[20] DAGUM, L., AND MENON, R. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering, IEEE 5*, 1 (Jan.-March 1998), 46–55.

[21] DONGARRA, J., FOSTER, I., FOX, G., GROPP, W., KENNEDY, K., TORCZON, L., AND WHITE, A. *Sourcebook of Parallel Computing.* Morgan Kaufmann, 2003.

[22] DUFF, I. S., HEROUX, M. A., AND POZO, R. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw. 28*, 2 (2002), 239–267.

[23] DWARKADAS, S., GHARACHORLOO, K., KONTOTHANASSIS, L., SCALES, D. J., SCOTT, M. L., AND STETS, R. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture* (Jan. 1999), pp. 260–269.

[24] EDELVIK, F. *Hybrid Solvers fo the Maxwell Equations in Time-Domain.* Doctoral thesis, Mathematics and Computer Science, Department of Information Technology, University of Uppsala, may 2002. `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-2156`.

[25] EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., STAMM, R. L., AND TULLSEN, D. M. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro 17*, 5 (1997), 12–19.

[26] FREDRICKSON, N. R., AFSAHI, A., AND QIAN, Y. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on Supercomputing* (2003), ACM Press, pp. 140–149.

[27] GHARACHORLOO, K., SHARMA, M., STEELY, S., AND DOREN, S. V. Architecture and design of alphaserver gs320. *ACM SIGPLAN Notices 35*, 11 (2000), 13–24.

[28] GIBBS, N. E., WILLIAM G. POOLE, J., AND STOCKMEYER, P. K. An Algorithm for Reducing the Bandwith and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis 13*, 2 (April 1976), 236–250.

[29] GOLUB, G., AND D.O'LEARY. Some history of the conjugate gradient and Lanczos methods. *SIAM Review 31* (1989), 50–102.

[30] GRAMA, A., GUPTA, A., KARYPSIS, G., AND KUMAR, V. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley, 2003.

[31] HAGERSTEN, E., AND KOSTER, M. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture* (Feb. 1999), pp. 172–181.

[32] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture;A Quantative Approach*, 3rd ed. Morgan Kaufman, 2003.

[33] HOLMGREN, S., NORDÉN, M., RANTAKOKKO, J., AND WALLIN, D. Performance of PDE Solvers on a Self-Optimizing NUMA Architecture. *Parallel Algorithms and Applications 17*, 4 (2002), 285–299.

[34] IEEE STD 1003.1-1996, ISO/IEC 9945-1. *Portable Operating System Interface (POSIX)–Part1: System Application Programming Interface (API) [C Language]*, 1996.

[35] INFINIBAND(SM) TRADE ASSOCIATION. *InfiniBand Architecture Specification, Release 1.0*, oct 2000. Available from: `http://www.infinibandta.org`.

[36] JAMIESON, P., AND BILAS, A. CableS: Thread Control and Memory Mangement Extentions for Shared Virtual Memory Clusters. In *8th International Symposium on High-Performance Computer Architeture, HPCA-8* (2002).

[37] JIN, H., FRUMKIN, M., AND YAN, J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[38] KARYPSIS, G., AND KUMAR, V. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1999), 359–392.

[39] KELEHER, P., COX, A. L., DWARKADAS, S., AND ZWAENEPOEL, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference* (Jan. 1994), pp. 115–131.

[40] LARUS, J. R., AND SCHNARR, E. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* (June 1995), pp. 291–300.

[41] LAUDON, J., AND LENOSKI, D. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture* (1997), ACM Press, pp. 241–251.

[42] LÖF, H., AND HOLMGREN, S. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 6th European Workshop on OpenMP* (2004), M. Brorsson, Ed., Royal Institute of Technology, pp. 3–8.

[43] LÖF, H., NORDÉN, M., AND HOLMGREN, S. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers. Tech. Rep. 2004-006, Department of Information Technology, Uppsala University, 2004.

[44] LÖF, H., NORDEN, M., AND HOLMGREN, S. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers. In *Computational Science - ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part II* (`http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&v%olume=3037&spage=9`, 2004), vol. 3037 of *LNCS*, pp. 9–16.

[45] LÖF, H., RADOVIĆ, Z., AND HAGERSTEN, E. THROOM — Running POSIX Multithreaded Binaries on a Cluster. Tech. Rep. 2003-026, Department of Information Technology, Uppsala University, apr 2003.

[46] LÖF, H., AND RANTAKOKKO, J. Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures. Tech. Rep. 2004-048, Department of Information Technology, Uppsala University, 2004.

[47] MATTSON, T. G. How good is OpenMP. *Scientific Programming 11* (2003), 81–93.

[48] MUELLER, F. Distributed Shared-Memory Threads:DSM-Threads. In *Proc. of the Workshop on Run-Time Systems for Parallel Programming* (1997).

[49] NIKOLOPOULOS, D. S., PAPATHEODOROU, T. S., POLYCHRONOPOULOS, C. D., LABARTA, J., AND AYGUADE, E. A transparent runtime data distribution engine for OpenMP. *Scientific Programming 8* (2000), 143–162.

[50] NIKOLOPOULOS, D. S., POLYCHRONOPOULOS, C. D., AND AYGUADI, E. Scaling irregular parallel codes with minimal programming effort. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (2001), ACM Press, pp. 16–16.

[51] NOORDERGRAAF, L., AND VAN DER PAS, R. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)* (1999), ACM Press, p. 38.

[52] OLIKER, L., LI, X., HUSBANDS, P., AND BISWAS, R. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review 44*, 3 (2002), 373–393.

[53] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Fortran Specification v2.0*. `http:www.openmp.org`, Nov 2000.

[54] PINAR, A., AND HEATH, M. T. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)* (1999), ACM Press, p. 30.

[55] RADOVIĆ, Z., AND HAGERSTEN, E. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of Supercomputing 2001* (Nov. 2001).

[56] RADOVIĆ, Z., AND HAGERSTEN, E. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of Supercomputing 2002* (Nov. 2002).

[57] SAAD, Y. *SPARSKIT: a basic tool kit for sparse matrix computations*, 2:nd ed. University of Minnesota, `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`, 1994.

[58] SCALES, D. J., AND GHARACHORLOO, K. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France* (Oct. 1997).

[59] SCALES, D. J., GHARACHORLOO, K., AND THEKKATH, C. A. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (Oct. 1996), pp. 174–185.

[60] SCHOINAS, I., FALSAFI, B., HILL, M., LARUS, J. R., AND WOOD, D. A. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques* (Oct. 1998).

[61] SCHOINAS, I., FALSAFI, B., LEBECK, A. R., REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)* (Oct. 1994), pp. 297–306.

[62] SCOTT, S. L. Synchronization and communication in the t3e multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems* (1996), ACM Press, pp. 26–36.

[63] SILICON GRAPHICS. *SGI Origin 3000 series*, 2000.

[64] SISTARE, S., AND JACKSON, C. J. Ultra-High Performance Communication with MPI and the Sun Fire(TM) Link Interconnect. In *Proceedings of the IEEE/ACM SC2002 Conference* (nov 2002).

[65] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principle* (Oct. 1997).

[66] SUN MICROSYSTEMS. *Solaris Memory Placement Optimization and Sun Fire servers*. `http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf`, January 2003.

[67] SUN MICROSYSTEMS. *UltraSPARC III Cu User's Manual*. `http://www.sun.com/processors/manuals`, 2003.

[68] THAIN, D., AND LIVNY, M. Multiple Bypass, Interposition Agents for Distributed Computing. In *Cluster Computing* (2001).

[69] TOLEDO, S. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. Develop 41*, 6 (1997), 711–725.

[70] VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSENBLUM, M. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems* (1996), ACM Press, pp. 279–289.

[71] VUDUC, R., DEMMEL, J. W., YELICK, K. A., KAMIL, S., NISHTALA, R., AND LEE, B. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002), IEEE Computer Society Press, pp. 1–35.

[72] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)* (June 1995), pp. 24–36.

Recent licentiate theses from the Department of Information Technology

2003-008	Zoran Radovic: *Efficient Synchronization and Coherence for Nonuniform Communication Architectures*
2003-009	Martin Karlsson: *Cache Memory Design Trade-offs for Current and Emerging Workloads*
2003-010	Dan Wallin: *Exploiting Data Locality in Adaptive Architectures*
2003-011	Tobias Amnell: *Code Synthesis for Timed Automata*
2003-012	Olivier Amoignon: *Adjoint-Based Aerodynamic Shape Optimization*
2003-013	Stina Nylander: *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*
2003-014	Kajsa Ljungberg: *Numerical Methods for Mapping of Multiple QTL*
2003-015	Erik Berg: *Methods for Run Time Analysis of Data Locality*
2004-001	Niclas Sandgren: *Parametric Methods for Frequency-Selective MR Spectroscopy*
2004-002	Markus Nordén: *Parallel PDE Solvers on cc-NUMA Systems*
2004-003	Yngve Selén: *Model Selection*
2004-004	Mohammed El Shobaki: *An Integrated Hardware Monitor for Multipurpose Real-Time Observations*
2004-005	Henrik Löf: *Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden