Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures

Henrik Löf and Jarmo Rantakokko

October 28, 2004

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.

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 [14] 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 [7] 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 [8], bandwidth minimization of the iteration matrix [12] and graph partitioning using the MeTiS tool [15]. We have also used a page migration feature of Solaris 9 [21] 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).

2 Parallelizing the CG-algorithm using OpenMP

When parallelizing iterative solvers like the CG-algorithm [1] there are three basic types of operations to consider:

Vector operations (Lines (4),(5) and (8) of Algorithm 1) These operations are trivially parallelized and they require very large vectors to

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 $k = 0, 1, \ldots$

(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 $p_{k+1} = r_{k+1} + \beta_k p_k$

amortize the parallel overhead. In OpenMP this operation translates into a simple parallelized loop.

Inner products (Lines (2) and (6) of Algorithm 1) 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 1) 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 outermost 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 4 (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 cases these can be removed using the `NOWAIT` clause.

[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.

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 4.

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 2. 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.

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).

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 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.

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.

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;
     }
}
```

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) [9] developed by Gibbs, Pool and Stockmeyer (GPS). The GPS method give equal quality minimizations in less time as shown in [12].

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 5 (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

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

Algorithm 3 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

$$< Ar_{k+1}, r_{k+1} > \text{ and } < r_{k+1}, r_{k+1} >$$

(7)

$$\beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$$

(8)

$$\alpha_{k+1} = \frac{< r_{k+1}, r_{k+1} >}{< Ar_{k+1}, r_{k+1} > -(\beta_k/\alpha_k) < r_k, r_k >}$$

Chronopoulos et al. formed the *s-step* ($s = 1$) version of the CG algorithm, see Algorithm 3. Here, an extra vector operation is introduced to remove one inner product, hence, removing also one barrier and giving a total of two barriers per iteration (see Algorithm 6 Appendix A). However, the s-step formulation contain more arithmetical work than the standard formulation.

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 [10]. 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

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

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

3.4 Related Work

The CG algorithm has been extensively used and studied in the numerical community [13, 1]. Several attempts have been made to increase the performance of the SpMxV [20, 5]. Apart from reordering, which we also use, other optimizations are: register blocking to lower the amount of load instructions [20, 24, 23], address precomputation [23] and cache blocking [24].

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 [3], 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. [19] comes closest to our work. Here, the authors showed that for a similar implementation of the CG algorithm, running on an SGI Origin 2000 [16] 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 [18, 2, 4, 14]. 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.

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.

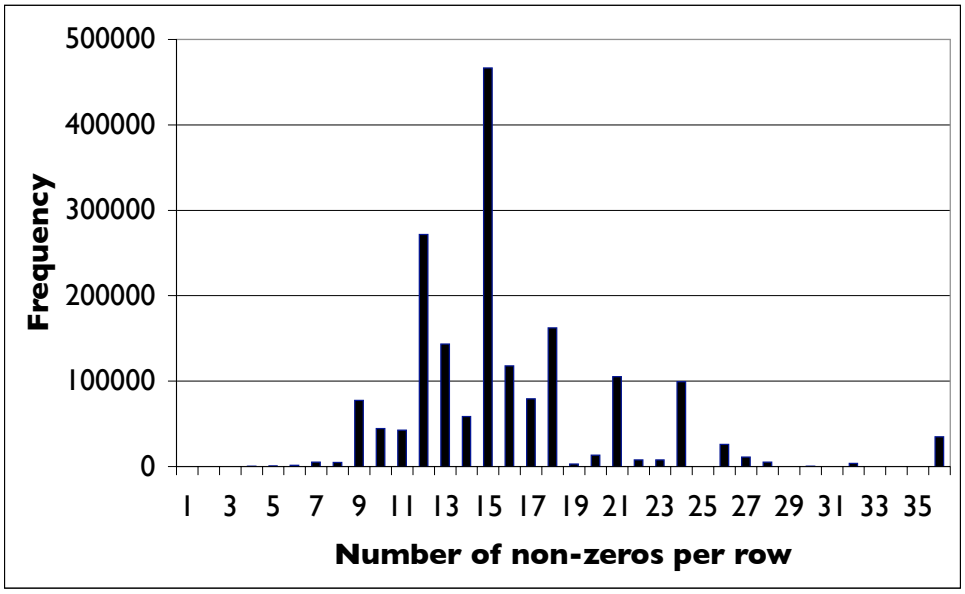
4.1 Execution environment

The results where measured on a dedicated 32 CPU domain of a SF15K [7] and on a 32 processor SUN E10K server [6]. 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 [22] 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.

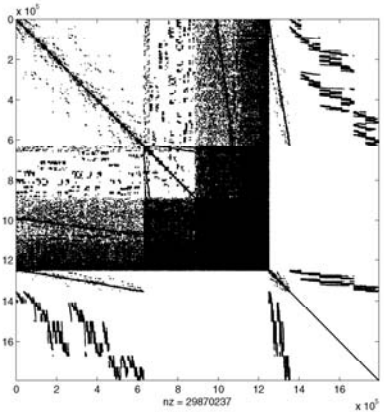
4.2 The FEM solver

The studied solver solves the Maxwell equations in electro-magnetics around a fighter jet configuration [11] 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 1) and the solver converge in 48 iterations without any preconditioning. Figure 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.

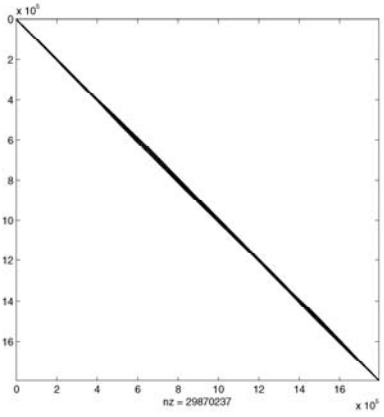
The main part of the code was implemented in Fortran90. The SpMxV however was implemented in C, to support more aggressive optimizations by the compiler. We do not run the entire FEM solver framework. Instead we have saved the iteration matrix to file and isolated the iterative solver. Hence, the matrix 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. [14, 17]. 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



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



(b) Base



(c) Bandwidth minimization

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

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 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.

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 1(b) and Algorithm 4.

S-Step This is an implementation of the S-Step algorithm, see Section 3.2 and Algorithm 6 using the original data, see Figure 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 3.1. The structure of the resulting reordered matrix can be found in Figure 1(c)

OPT Here, we use the optimized CG algorithm, see Algorithm 5

MeTiS Here, we use the optimized algorithm, see Algorithm 5 and the matrix is reordered according to the partitions produced by MeTiS as described in Section 3.3

GPS+OPT Here, we use the optimized CG algorithm, see Algorithm 5 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 3.3

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

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 [22]. 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 [17]. 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.

5 Experimental Results

From Figures 2 and 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 4 and 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 6, Table 1, Table 2). Figure 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 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 more detailed

	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 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.

measurements in Table 1 and Table 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.

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

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 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.

References

[1] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[2] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA machines. *Scientific Programming*, 8:163–181, 2000.

[3] Jürgen Brehm and Harry F. Jordan. Parallelizing algorithms for mimd architectures with shared memory. In *Proceedings of the 3rd international conference on Supercomputing*, pages 244–253. ACM Press, 1989.

[4] J. Mark Bull and Chris Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP*. `http://www.caspur.it/ewomp2002/`, 2002.

[5] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Technical report, Oxford University Computing Laboratory, Numerical Analysis Group, May 1995.

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

[7] Alan Charlesworth. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press, 2001.

[8] A.T. Chronopoulos and C.W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM Press, 1969.

[10] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.

[11] Fredrik Edelvik. *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`.

[12] Norman E. Gibbs, Jr. William G. Poole, and Paul K. Stockmeyer. An Algorithm for Reducing the Bandwith and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.

[13] Gene Golub and D.O'Leary. Some history of the conjugate gradient and Lanczos methods. *SIAM Review*, 31:50–102, 1989.

[14] Sverker Holmgren Henrik Löf, Markus Norden. 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*, volume 3037 of *LNCS*, pages 9–16, `http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&vo%lume=3037&spage=9`, 2004.

[15] George Karypsis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[16] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251. ACM Press, 1997.

[17] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In Mats Brorsson, editor, *Proceedings of the 6th European Workshop on OpenMP*, pages 3–8. Royal Institute of Technology, 2004.

[18] Dimitros S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8:143–162, 2000.

[19] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

[20] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30. ACM Press, 1999.

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

[22] Sun Microsystems, `http://www.sun.com/processors/manuals`. *UltraSPARC III Cu User's Manual*, 2003.

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

[24] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–35. IEEE Computer Society Press, 2002.

A Implementations in OpenMP

Here we present our implementations of th CG algorithm in OpenMP. The standard implementation, Algorithm 4 represents a direct implementation from the CG algorithm. In Algorithm 5 we have used features of OpenMP to reorder some calculations to save global barriers. Finally, Algorithm 6 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 4 Standard OpenMP implementation of CG

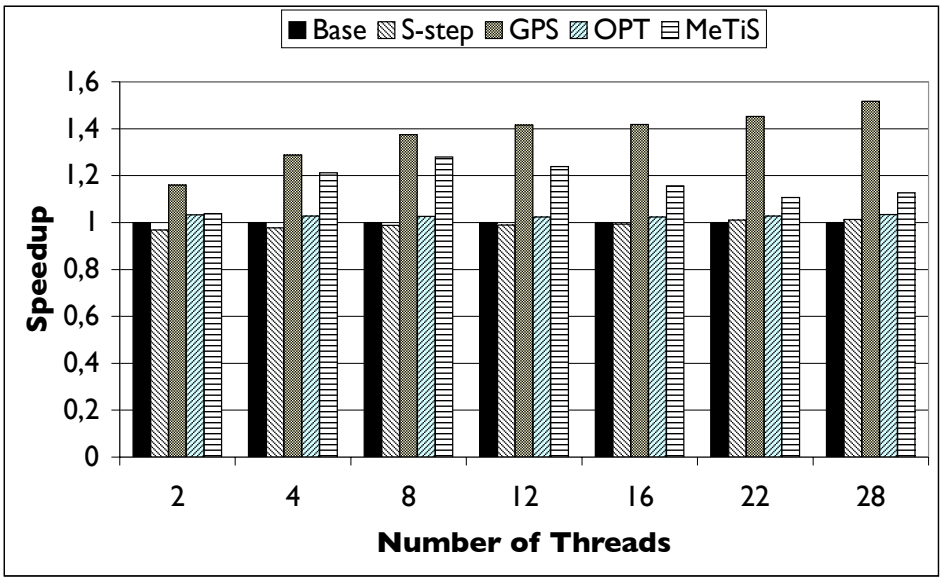
```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(cg_iter_count)

    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 5 Optimized OpenMP implementation of CG

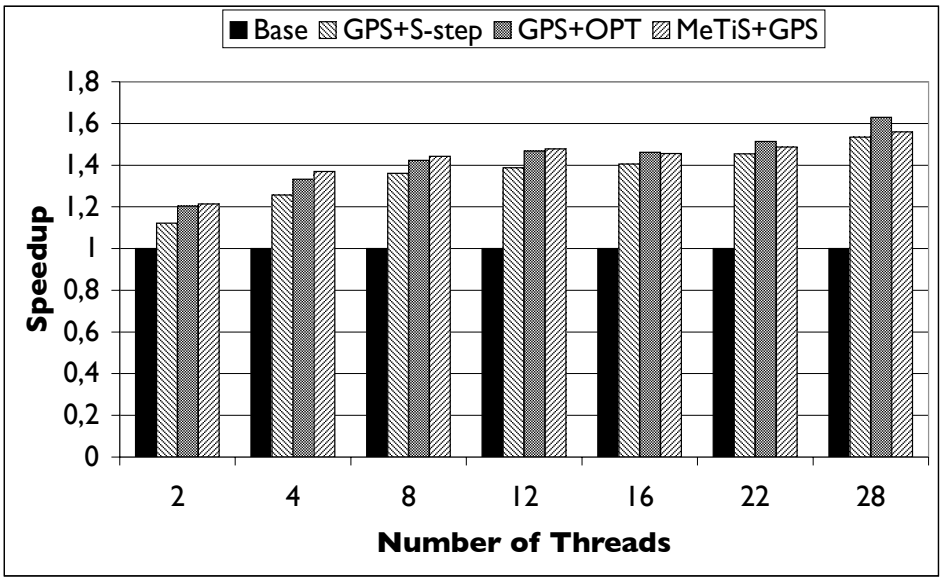
```
   pAp_norm = 0.0_rfp

!$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 6 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
```

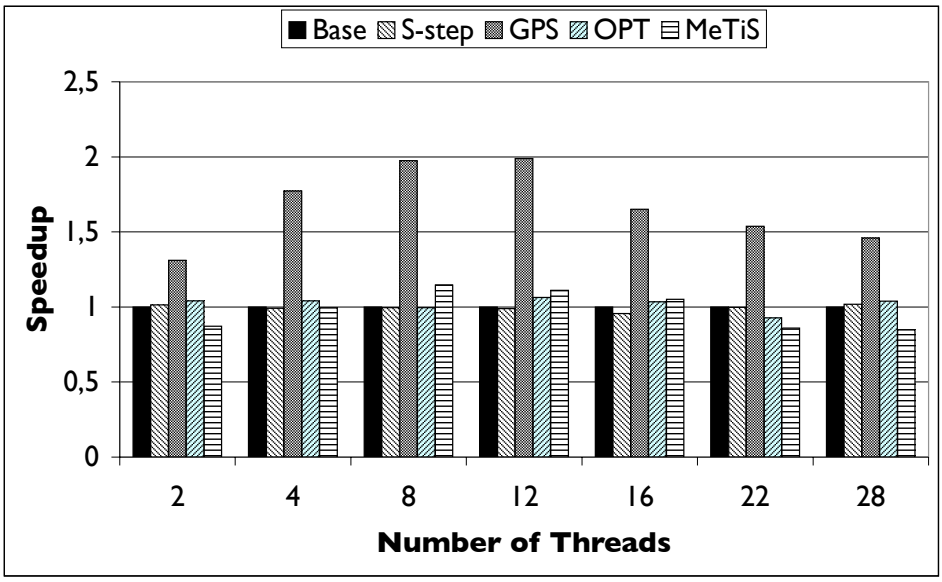


(a) Speedup of basic optimizations on the E10K

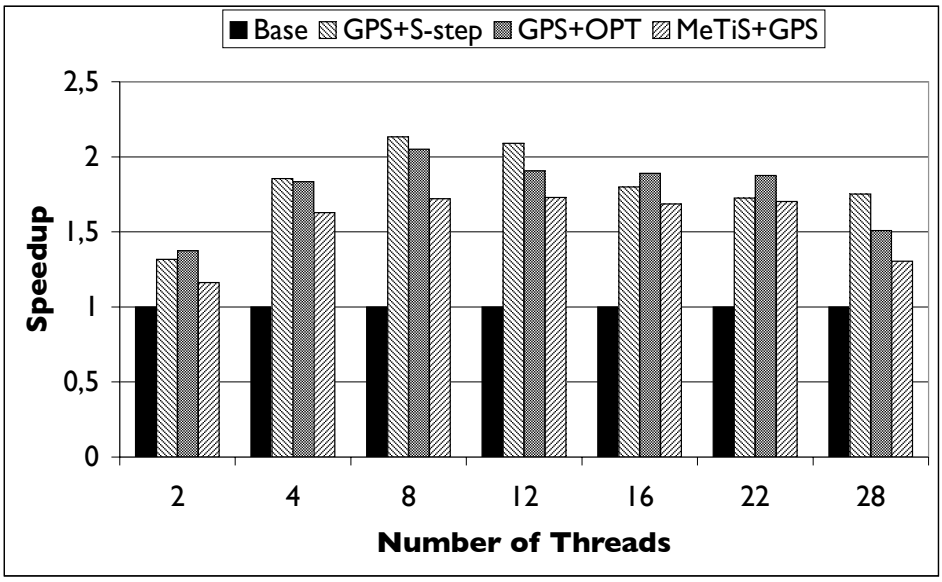


(b) Speedup of combinations of optimizations on the E10K

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

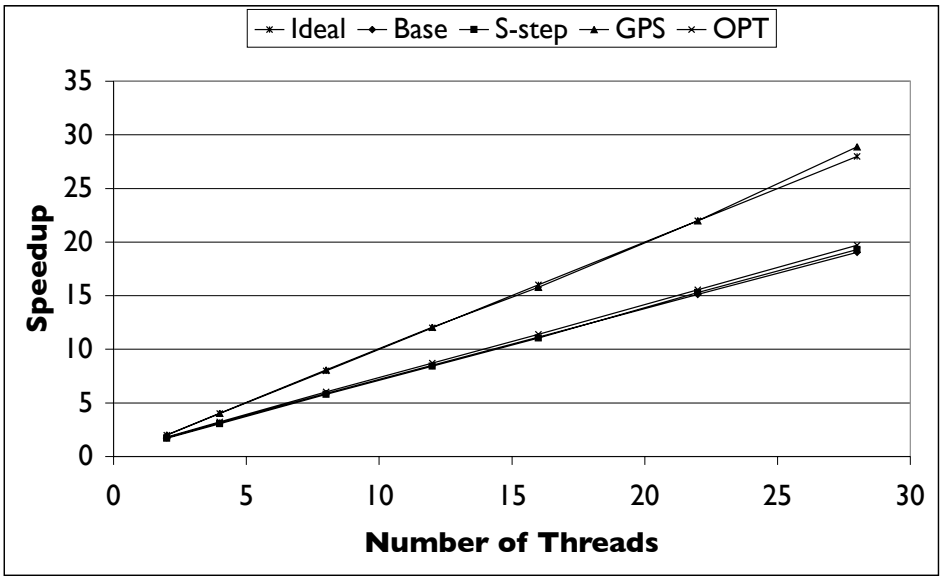


(a) Speedup of basic optimizations on the SF15K

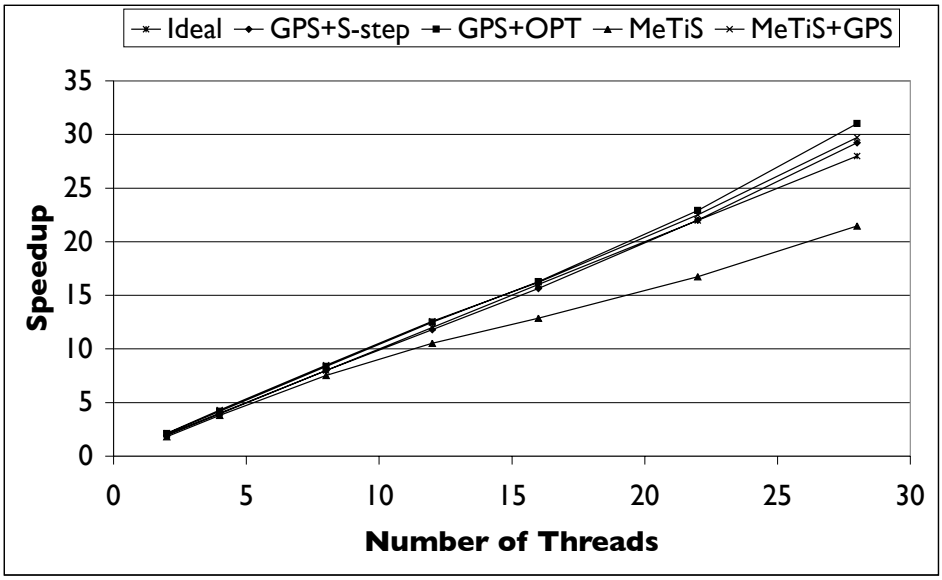


(b) Speedup of combinations of optimizations on the SF15K

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

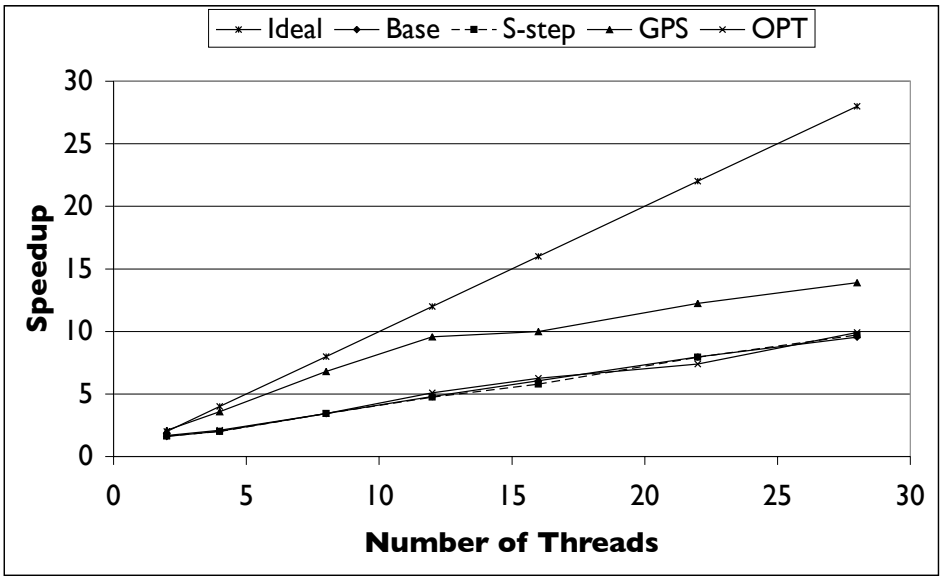


(a) Speedup of basic optimizations on the E10K

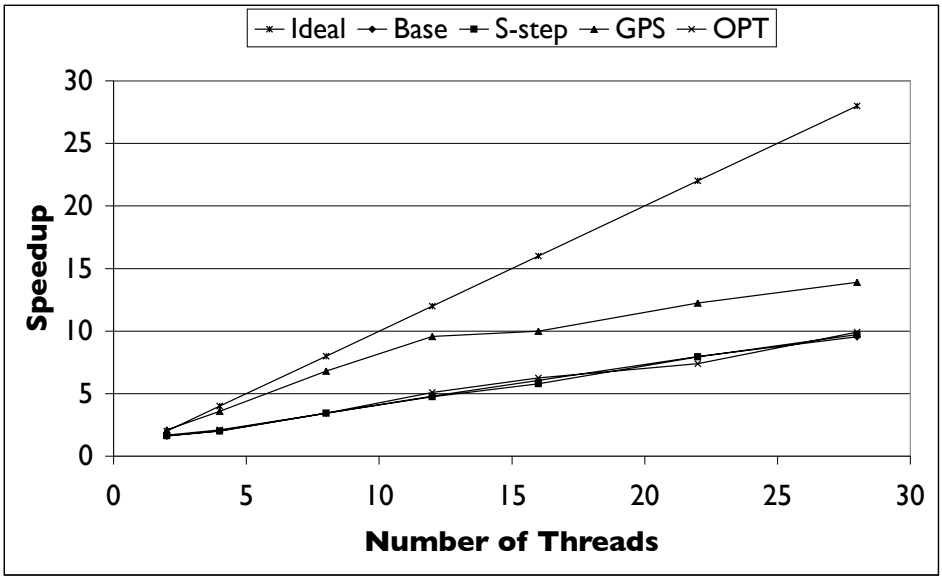


(b) Speedup of combinations of optimizations on the E10K

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

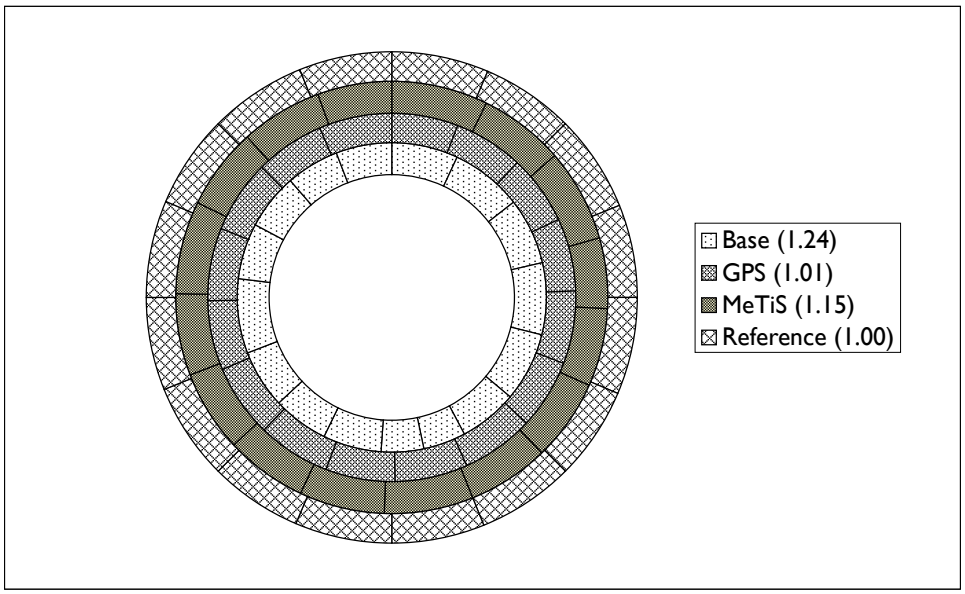


Figure 6: Visualization of the arithmetical load balance for the case of 16 partitions (threads). The reference ring corresponds to perfect balance which is impossible to generate using reorderings. The number in parenthesis in the legend is the γ parameter defined in Section 4.4

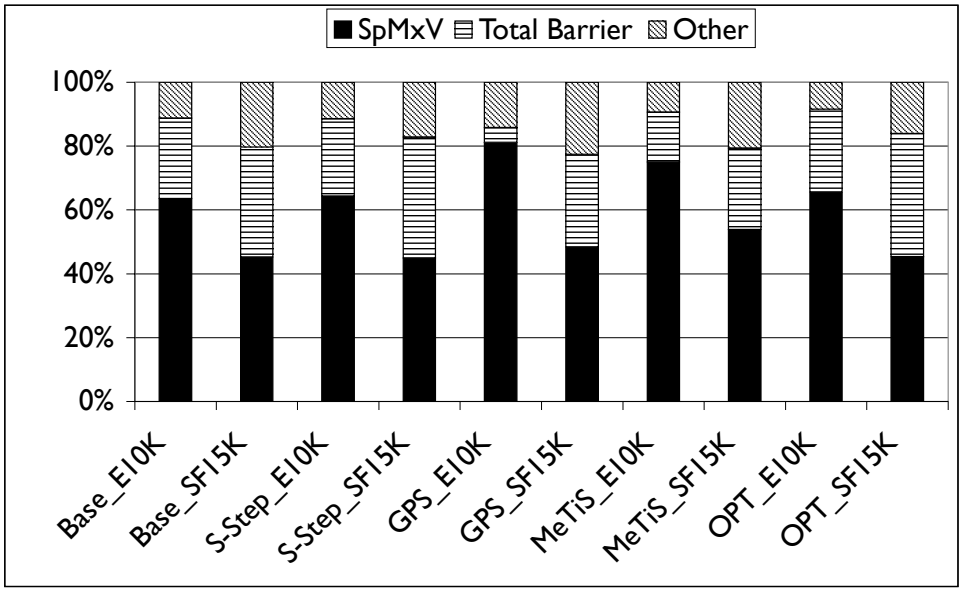


Figure 7: Distribution of CPU time