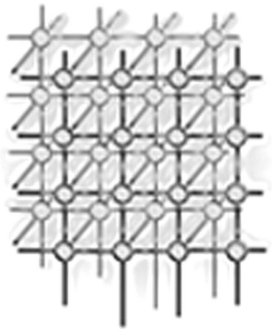
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures

J. C. Pichel[1,*,†], D. B. Heras[2], J. C. Cabaleiro[2] and F. F. Rivera[2]

[1] *Computer Science Dpt., Universidad Carlos III de Madrid, Spain*
[2] *Electronics and Computer Science Dpt., Universidade de Santiago de Compostela, Spain*

SUMMARY

In this paper the problem of the locality of sparse algebra codes on simultaneous multithreading architectures is studied. In this kind of architectures many hardware structures are dynamically shared among the running threads. This puts a lot of stress on the memory hierarchy, and a poor locality, both inter-thread and intra-thread, may become a major bottleneck in the performance of a code. This behavior is even more pronounced when the code is irregular, which is the case of sparse matrix ones. Therefore, techniques that increase the locality of irregular codes on simultaneous multithreading architectures are important to achieve high performance.
This paper proposes a data reordering technique specially tuned for this kind of architectures and codes. It is based on a locality model developed by the authors in previous works. The technique has been tested, first, using a simulator of a simultaneous multithreading architecture, and subsequently, on a real architecture as Intel's Hyper-Threading. Important reductions in the number of cache misses have been achieved, even when the number of running threads grows. When applying the locality improvement technique we also decrease the total execution time and improve the scalability of the code.

KEY WORDS: sparse matrix; irregular codes; data reuse; locality; multithreading

*Correspondence to: Univ. Carlos III de Madrid, Av. de la Universidad 30, 28911 Leganés, Spain.
†E-mail: jcpichel@arcos.inf.uc3m.es
Contract/grant sponsor: Spanish Ministry of Education and Science (MEC); contract/grant number: TIN2007-67537-C03-01 and TIN2007-63092

Copyright © 2007 John Wiley & Sons, Ltd.
Received
Revised



1. INTRODUCTION

Simultaneous Multithreading (SMT) is one of the most successful implementations of the multithreaded architectures. SMT allows several independent threads to issue instructions to a superscalar's multiple functional units in a single cycle [1]. Unlike conventional multithreaded architectures, which depend on fast context switching to share processor execution resources, all hardware contexts in a SMT processor are active simultaneously, competing each cycle for all available resources. SMT architectures can be implemented without adding undue complexity to a conventional superscalar processor design. To support SMT, the base processor architecture requires significant changes in two primary areas: the instruction fetch mechanism, and a larger register file [2]. Moreover, few structures are replicated, i.e. statically assigned to single threads, namely program counters, subroutine return stacks, retirement and trap mechanisms. All the other structures (functional units, caches, TLBs, branch prediction mechanisms, etc.) are dynamically shared among threads, and all are available even when sequential code (only one thread) is being executed. Tullsen *et al.* [3] show that a SMT architecture supporting 8 threads could achieve up to four times the instruction throughput of a conventional superscalar architecture with the same issue bandwidth.

However, performance on a simultaneous multithreading architecture can be degraded by the following factors. First, the simultaneous execution of several threads puts a lot of stress on the memory hierarchy. Hily *et al.* [4] point out that ignoring the contention on the L2 cache leads to overestimate the performance of the application on a SMT architecture. That is, the L2 cache is an important limiting factor in the performance when several threads are running simultaneously. Secondly, parallel applications have the potential to compete more heavily for shared resources than a multiprogrammed workload. And if the threads of the parallel program each have similar characteristics, they will likely each put pressure on the same resources in the same way, and a single resource will become a bottleneck more quickly [2].

Therefore, data locality, both inter-thread and intra-thread, is especially relevant in SMT architectures. This fact becomes even more important when an irregular code is executed on them. These codes are characterized by irregular accesses, and they tend to have low spatial and temporal localities. Consequently, low performance can be expected when executing these codes on a SMT architecture. For this reason, techniques that increase the locality are very profitable to improve the performance. In this paper, we propose a technique to deal with the problem of increasing the locality of irregular codes as sparse matrix ones on SMT architectures.

The technique consists of reorganizing the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. The goal is to increase the grouping of elements in the pattern of the sparse matrix that characterizes the irregular accesses and, as a consequence, increasing the locality in the execution of the code. In the experiments we show that our reordering is effective for matrices with any structure and for different sparse matrix kernels. Obviously, the reordering algorithm must be tuned for the different architectures taking into account the characteristics of the particular architecture and the code considered. When our locality improvement technique is applied, as we show in this paper, both inter-thread and intra-thread localities are increased.

The irregular kernel that we have selected as case of study is a particular implementation of the product of a sparse matrix by a set of dense vectors. In particular, this operation occurs in

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

practice when there are multiple right-hand sides in an iterative solver, in recently proposed blocked iterative solvers [5], and in blocked eigenvalue algorithms such as block Lanczos [6] or block Arnoldi [7]. Our technique has been tested, both in a simulator of SMT architectures, and on a real architecture, the Intel's *Hyper-Threading* (HT) [8].

The paper is organized as follows: in Section 2 the related work is discussed. Section 3 presents a summary of the locality model, on which the technique presented in this paper is based. Moreover, the locality properties of the selected irregular kernel are detailed. The locality improvement technique for SMT architectures is described in Section 4. The experimental setup and the benefits of applying our technique on a SMT architecture are discussed in Section 5. Section 6 is devoted to a brief description of the Hyper-Threading technology, as well as to the results obtained on this architecture. The paper finishes with the main conclusions derived from this work.

2. RELATED WORK

The techniques for increasing the locality of irregular codes can be mainly divided into two groups: data reordering techniques and code restructuring techniques.

Standard reordering techniques are considered beneficial methods for dealing with the problem of increasing the locality in the execution of irregular codes. Typical examples of the standard reordering algorithms are bandwidth reduction algorithms, which derive from the Cuthill-McKee algorithm [9], and Minimum Degree algorithm-based heuristics such as the Approximate Minimum Degree algorithm [10]. Oliker *et al.* [11] show the benefits, in terms of locality, that offers the application of some of these reordering algorithms to sparse codes when executing on different multiprocessor architectures.

Techniques based on restructuring the code, like *blocking*, have been applied with success to different irregular codes, such as the product of a sparse matrix by a dense matrix [12, 13].

More recently some authors have demonstrated that both groups of techniques are complementary in the sense that when they are applied together the locality is increased. In particular, Toledo [14] evaluates different standard reordering techniques and combines them with blocking, showing that performance increases significantly depending on the size and sparseness of the considered matrix. Pinar and Heath [15] introduce a reordering technique that favors the creation of dense blocks on the sparse matrix pattern, and in this way, the efficiency of the blocking technique proposed by Toledo is increased. Also a comparison between their reordering technique and some standard reordering techniques is presented. In other work [16], a combination of data reordering algorithms and register blocking in different orders have been successfully applied to the sparse matrix–vector product. Good results were achieved, but only for some specific matrices, and just considering standard reordering algorithms. In the same way, in [17], a systematic study combining reordering and register blocking for a broad range of matrices is presented, also including reordering algorithms developed by the authors. These reorderings are guided by the locality properties of the code and obtain important improvements on four different platforms. Strout *et al.* [18] propose a run-time reordering technique for improving intra-iteration and inter-iteration data locality for the Gauss-Seidel iterative method based on the inspector/executor paradigm.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



In order to reduce the cache miss rate on SMT architectures, some works propose a dynamic cache partitioning scheme [19]. However the partitioning scheme is not able to improve the performance if caches are too small for the workloads. Nikopoulos [20] proposes software solutions for dynamically partitioning the shared cache of an SMT processor, via the use of several methods such as dynamic tiling, copying and blocking data layouts. In other work, Puppin and Tullsen [21] discuss some general and ad-hoc parallelization techniques for SMT, and also propose a method for determining the best number of threads to be used for parallel loops. In these works, only regular codes are considered.

In a recent work [22], the authors propose several optimization techniques for the sparse matrix–vector multiplication, which are evaluated on different multicore platforms. Authors examine a wide variety of techniques including: software pipelining, prefetching, register and cache blocking, etc. Nevertheless, they do not evaluate data reordering techniques in order to increase inter and intra-thread localities.

3. LOCALITY MODELING ON MULTITHREADING ARCHITECTURES

As noted above, on a multithreading architecture different threads interleave their execution and share some resources and, in particular, the cache memory. The main effect of this fact are the conflict misses. In this section a locality model for characterizing and increasing the locality of the accesses performed by the threads is presented.

This model was applied to different irregular codes and architectures in previous papers [23], and it is extended to the execution of sparse matrix codes on multithreading architectures in this work. The locality model is used to guide a locality improvement process. We assume that, when a high spatial or temporal locality in the accesses generated by the execution of a code is exhibited, there is reuse of data in a particular level of the memory hierarchy under study [24].

3.1. Locality properties of the product of a sparse matrix by a set of vectors

In this work the process of locality improvement on multithreading architectures is illustrated considering the operation of the product of a sparse matrix by a set of dense vectors. Like in other sparse matrix codes, the performance is highly dependent on the nonzero structure of the sparse matrix. In order to apply the locality model, the locality properties of the selected irregular code have to be analyzed.

Let's consider the operation R=A×B, where R and B are dense matrices, and A is a $N \times N$ sparse matrix stored using the standard Compressed-Sparse-Row format (CSR) [7]. DA, INDEX and PTR are the three vectors (data, column indices and row pointer) that characterize this format. In particular, B is the sequence of M dense vectors being $M \ll N$ (as it is usual in real codes mentioned in Section 1) by which the product is performed. The product using the ijk nesting can be implemented for its sequential execution as displayed in Figure 1.

For this code the entries of the sparse matrix are accessed by rows. Note that the accesses to vectors DA, INDEX and PTR present high spatial locality. In the same way, accesses to the elements of result matrix R are sequential, and therefore, a high locality is expected. For each

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
for (i=0; i < N; i++) {
   for (j=0; j < M; j++) {
      reg=0;
      for (k=PTR[i]; k < PTR[i+1]; k++) {
         reg = reg + DA[k]*B[j][INDEX[k]];
      }
      R[i][j]=reg;
   }
}
```

Figure 1. Multiplication of a sparse matrix by a set of M vectors using the ijk nesting.

iteration of the outermost loop i, M rows of dense matrix B are accessed in order to perform the product of row i of the sparse matrix for each dense vector. Note that the elements of each vector are stored in contiguous memory locations. The positions of the accessed elements of each vector (i.e. each row of B) depend on the positions of the entries in row i of A given by INDEX(k). Consequently, B presents irregular accesses, whose locality properties depend directly on the pattern of the sparse matrix. In this way, the locality for this nesting can be improved by increasing the grouping of entries over the sparse matrix pattern that will basically affect the locality in the accesses to B.

When the code is executed on a multithreading architecture there are additional effects that must be considered. This is the case when consecutive rows of the sparse matrix A are accessed by different threads. For multithreading parallelism better speedups are expected when using a cyclic loop distribution rather than for the traditional block distribution [25], because it clusters the accesses of multiple threads onto fewer pages of the memory, reducing the TLB footprint of the parallel code. Note that sharing data among threads can also be beneficial because it leads to accesses of different threads to the same data placed in a cache architecture that is shared by these threads. Consequently, threads reuse the data previously accessed in the cache by a different thread.

In order to illustrate the influence of both inter-thread and intra-thread localities, consider the example of Figure 2. In this figure a small 10×10 sparse matrix is considered. This is the matrix to be multiplied by the set of M vectors according to the code from Figure 1. Consider two threads and a standard cyclic distribution of matrix rows. For each thread, Figures 2(a) and 2(b) show the list of terms of the vectors accessed according to the indirection array INDEX of the original and a reordered matrix respectively. Lines connecting elements accessed by the same thread illustrate intra-thread reuse, and lines connecting elements accessed by both threads represent inter-thread reuse. Note that long lines represent poor locality. In this way, if rows are reordered to group entries in the sparse matrix, as in the case of Figure 2(b), the degree of both types of locality increases. Furthermore, a reordering by columns of the matrix that increase the grouping of elements in a particular row of the matrix will lead to accesses to nearer elements of each dense vector of B, improving the spatial locality.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

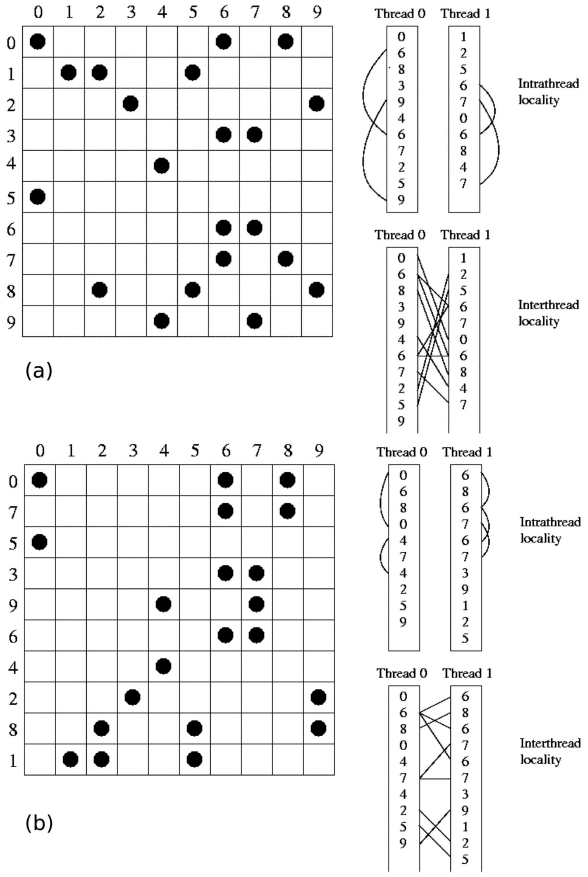


Figure 2. Example of data reuse using a cyclic distribution of the matrix among two threads.

Depending on the storage scheme of the sparse matrix, different versions of the code are available. Moreover, the storage scheme usually implies a certain reordering of the data. For example, when blocking is applied to a nesting, the data is stored in blocks instead of in rows or columns in order to increase the locality [26, 27]. The locality improvement technique in this work can also be applied to codes where data are stored with other storage schemes. An example was published in [17] where a reordering of the sparse matrix involved in the product of a sparse matrix by a vector in combination with blocking techniques was applied. The technique was evaluated on different uniprocessors and distributed memory multiprocessors.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

3.2. Locality model

The locality model is based on the evaluation of the data locality of the considered sparse code performing measurements over the sparse matrix involved in the computations. The model is general enough to be applied to any sparse matrix code that can take profit, in terms of data reuse, of a clustering of entries in the pattern of the matrix. The locality model was previously applied to different computer platforms such as uniprocessors, shared and distributed memory multiprocessors [28, 17]. In terms of performance, the most relevant level of the memory hierarchy is different depending on the architecture. For multithreading architectures the biggest interest is in the levels of the cache memory.

According to our model, locality is measured for pairs of rows of the sparse matrix given that the prevailing accesses to the sparse matrix are row-wise. The measurement of locality is based on evaluating a parameter: the number of *entry matches* (a_{elems}). The number of entry matches between any pair of rows is defined as the number of nonzero elements in the same column of both rows. This parameter was selected from a set of possibilities through a statistical analysis. Based on it, a magnitude called *distance between rows x and y*, denoted by $d_i(x, y)$ is defined. It indirectly measures the locality displayed by the accesses performed by the irregular code on these two rows when they are consecutively accessed. We have proposed in [23] several different distance definitions that exhibit the same common properties, namely, that they are semidefinite positive, bounded and symmetric. In this work we have performed a preliminary analysis for selecting only one distance function as a case of study. Similar results were obtained with any of the definitions of distance function. In this work we have selected the following:

$$d_1(x, y) \quad = \quad n_{\text{elems}}(x) + n_{\text{elems}}(y) - 2{*}a_{\text{elems}}(x, y) \tag{1}$$

where $n_{\text{elems}}(x)$ is the number of elements in row x. It can be shown that this distance define a metric over the N rows of the matrix.

For the irregular code executed on a sparse matrix, we define a quantity that is inversely proportional to the locality induced by the whole sparse matrix. This quantity is evaluated by summing the distances between pairs of rows consecutively accessed. In other words:

$$D_1 = \sum_{x=0}^{N-2} d_1(x, x+1) \tag{2}$$

Small values of D_1 indicate large reuse of data in each iteration of loop i executed by the thread. The described model can be applied to any sparse matrix code, whose locality is determined by the level of grouping of the entries in the pattern of the matrix. And it is also suitable for any sparse matrix without limitations in the type of pattern that they present, as we show in the analysis of the results. The objective of the locality model is to guide a locality improvement process for increasing the reuse of data at any level of the memory hierarchy.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



Table I. Distance graph characteristics.

	Number of edges		
Matrices	**Complete Graph**	**Incomplete Graph**	**Reduction (%)**
bcsstk29	97881036	975847	99.0%
e40r0100	149307840	1254328	99.2%
garon2	91591345	725467	99.2%
memplus	157664403	2492122	98.4%
msc10848	58834128	3087399	94.7%
Na5	17003196	1649759	90.3%
ncvxbpq1	1249975000	524716	99.9%
nmos3	172747578	524226	99.7%
psmigr_1	4928230	4822253	2.1%
sme3Da	78168756	2912376	96.3%
syn12000a	71994000	5479088	92.4%
tandem_vtx	170265831	571925	99.7%

4. DATA REORDERING TECHNIQUE FOR MULTITHREADING ARCHITECTURES

The relevance of increasing the locality in the execution of irregular codes on multithreading architectures was outlined in the previous section. In order to deal with this problem we propose a heuristic technique consisting of modifying the pattern of the sparse matrix involved in the computations. The technique can be guided by evaluating the appropriateness of the modifications performed by means of the locality model. The objective of the data reordering technique for SMT architectures is to increase the sharing of data among threads, while in other cases like shared memory multiprocessors [28], the goal is to increase the locality in the accesses performed by each thread separately, avoiding the sharing of data among threads mapped to different processors. According to this, in order to increase the locality in the accesses performed by the irregular code on SMT architectures, we search for a permutation of rows (and columns) of the whole matrix that minimizes its total distance (D_1). This way, given that for SMT architectures the distribution of the sparse matrix among threads is cyclic by rows, a global reordering of the matrix by rows will increase inter and intra-thread reuse as we have explained in the example of Section 3.1. Likewise, a global reordering of the matrix by columns will increase the spatial locality in the accesses to elements of B. Note that if the matrix pattern is symmetric, the same ordering is applied by rows and columns.

We formulate the problem of locality improvement as a classic NP-complete optimization problem [29], and we propose to solve it as a graph problem using an analogy to the Traveling Salesman Problem (TSP). The problem can be described using a complete graph where each node represents a row (or a column) of the sparse matrix, and each edge has an associated weight that indicates the distance between pairs of rows (or columns) according to the description of locality given previously. Nevertheless, it is not necessary to work with a complete graph. Given that sparse matrices have a very low density of nonzero elements, most of the weights in the graph correspond with distances where $a_{elems} = 0$. These values, according to

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

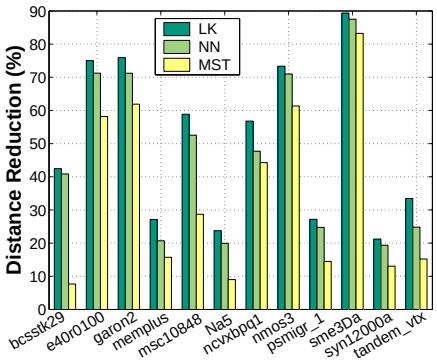


Figure 3. Reductions in the distance D_1 with respect to the original matrices using different heuristics.

the distance definitions, represent the worst cases regarding the locality. So, without losing relevant information about the locality, we can use an incomplete weighted graph, for which only distances with values of a_{elems} different from zero are considered. As a consequence, the graph size is reduced considerably. In order to illustrate this reduction a comparison of the complete and incomplete distance graph size (in terms of the number of edges) is provided in Table I for the matrices of our test set (see Table III for matrix details). Note that the reduction in the graph size when using the incomplete distance graph is noticeable, decreasing, in some cases, up to 99% in comparison with the size of the complete graph (matrices *bcsstk29*, *e40r0100*, *garon2*, *ncvxbpq1*, *nmos3* and *tandem_vtx*).

Solving the problem of reordering is equivalent to finding a path of minimum length that goes through all the nodes of the graph. This path is represented as a permutation vector that gives the appropriate order of the nodes of the graph, and therefore, a reordered sparse matrix. Given that we have measures (distance values) to validate the adequateness of an ordering, we have focused on heuristic solutions that give a good approximation to the best possible solution. After a comparative study of different techniques, the *Chained Lin–Kernighan* heuristic proposed by Applegate *et al.* [30] was chosen. As example of this study, Figure 3 shows the reduction of D_1 compared to the distance of the original matrix using three different ordering heuristics for each matrix of the test set (see Section 5.1 for details). These techniques are: the *Chained Lin–Kernighan* (LK) algorithm, the greedy heuristic known as *Nearest Neighbor* (NN) and the *Minimum Spanning Tree* (MST) using depth-first search for visiting the nodes of the graph. LK obtains the most important reductions in the distance in all the cases that we examined experimentally. Moreover, decreases in the distance for all the matrices are achieved. Therefore, the LK heuristic was selected.

As we have explained, a solution of the locality problem implies a reordering of the sparse matrix, which produces a reordered matrix with a higher clustering of entries in its pattern. As an example, Figure 4 shows the pattern of the *garon2* matrix of our test set when the reordering process is applied. The reordered matrix presents a higher level of clustering of

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

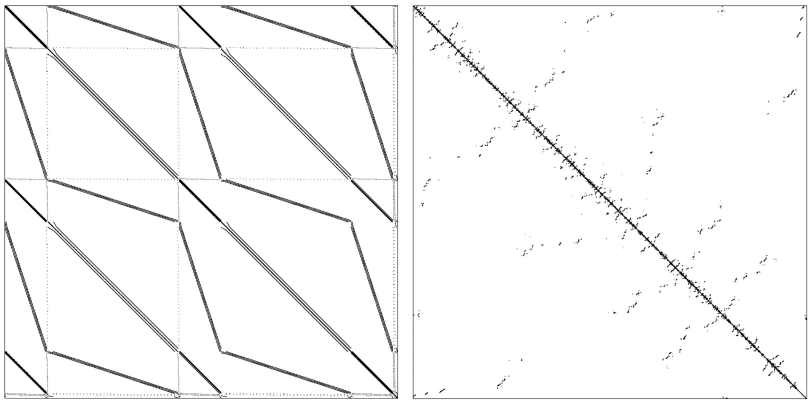


Figure 4. Matrix *garon2*: (a) Original pattern, and (b) reordered matrix.

entries than the original one, which will lead to improved locality as Section 3.1 points out. Note that although the original matrix (Figure 4(a)) intuitively will be better to use due to its regular sparsity pattern, results in the paper indicate the opposite (see Section 6).

In the following sections the reordering technique for increasing the locality of the product of a sparse matrix by a set of vectors is applied. The experiments were developed on two different multithreading platforms: a SMT simulator and a Intel's Hyperthreading system.

5. PERFORMANCE EVALUATION ON SMT ARCHITECTURES

According to Section 1, when executing a parallel code which computes the product of a dense matrix by a set of vectors on a SMT architecture, a poor performance is expected, especially if the code presents irregular accesses as in our case of study. This behavior is more critical when the number of running threads increases because they compete for the shared resources.

In this section the details of the simulated SMT architecture are shown, as well as the main characteristics of the matrices test set used in this work. Finally, the results, using the reordered matrices obtained after applying our technique, are displayed and discussed.

5.1. Experimental setup

The results in this section were obtained using the SMTSIM simulator [1]. The code of Figure 1 was written in C and compiled using gcc on an Alpha platform. In order to parallelize it some directives used by the simulator were included. In the parallel code, a cyclic distribution of

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

Table II. Details of the simulated cache hierarchy.

	L1 I	L1 D	L2	L3
Size	32 KB	32 KB	256 KB	4 MB
Associativity	2	2	2	2
Line Size (bytes)	64	64	64	64

Table III. Matrix benchmark suite.

	N	N_Z		N	N_Z
bcsstk29	13992	619448	*ncvxbpq1*	50000	349968
e40r0100	17281	553562	*nmos3*	18588	386594
garon2	13535	390607	*psmigr_1*	3140	543162
memplus	17758	126150	*sme3Da*	12504	874887
msc10848	10848	1229778	*syn12000a*	12000	1436806
Na5	5832	305630	*tandem_vtx*	18454	253350

the iterations of loop i among the threads was used. The main characteristics of the simulated SMT architecture are the ones detailed in [1], except the characteristics of the cache hierarchy described in Table II. The simulated cache hierarchy is similar to those present in modern processors.

As a test set to evaluate the improving technique we have selected twelve square sparse matrices from different real problems that represent a variety of non-zero patterns. These matrices are from the University of Florida Sparse Matrix Collection (UFL) [31]. Table III summarizes some features of the matrices, where N is the number of rows or columns (all the matrices are square), and N_Z is the number of entries. For all the results in this paper, we perform the product of a sparse matrix by a set of 20 dense vectors. This way, as in the real codes mentioned in Section 1, the number of vectors is usually much lower than the number of rows of the sparse matrix, that is, $M \ll N$. In practice the number of dense vectors changes depending on the problem.

5.2. Experimental results

Next the performance of the irregular code using the matrices after applying our reordering technique on the SMT architecture is evaluated. The results shown in this section have been obtained using six matrices of our test set. These matrices present a variety of patterns. They are *bcsstk29*, *ncxvbpq1*, *nmos3*, *sme3Da*, *syn12000a* and *tandem_vtx*.

First, we have measured the cache miss rate in order to illustrate the L2 cache contention. Figure 5 displays the results achieved considering original and reordered matrices (D_1) for different number of threads. In the case of original matrices, cache miss rate increases as the number of running threads becomes higher (except for matrix `bcsstk29`). This behavior

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

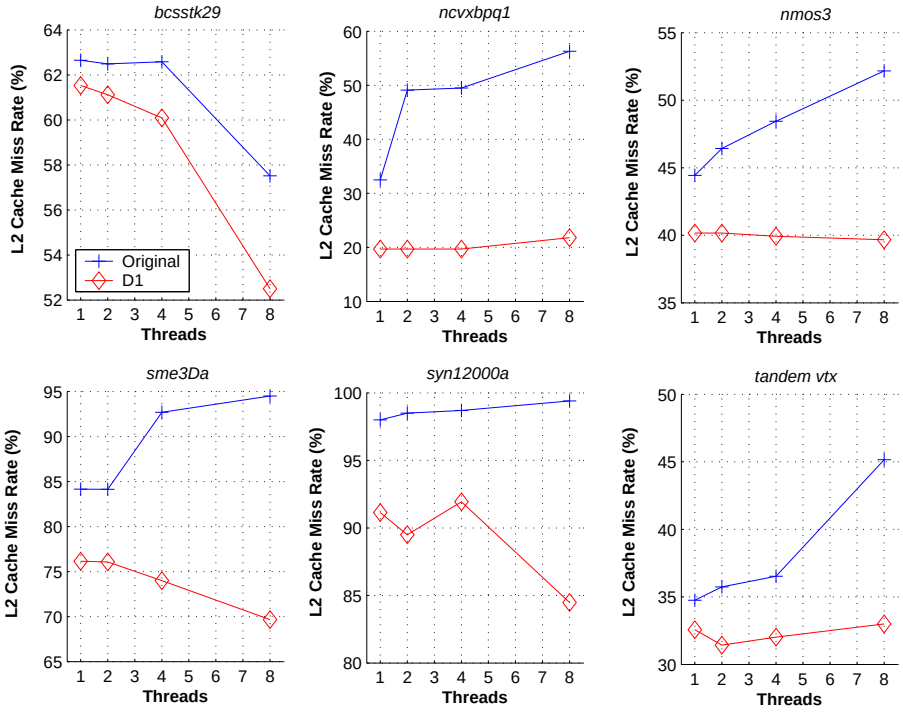


Figure 5. Evolution of L2 cache miss rate as the number of threads increases on the SMT simulator.

reveals a high L2 cache contention which, as shown later, limits the performance of original matrices when several threads are running concurrently on a SMT processor. However, when reordered matrices are considered, the trend of the cache miss rate changes completely. For these matrices, cache miss rate decreases (or increases slightly) as the number of threads grows. Therefore, using reordered matrices reduces noticeably cache contention due to the fact that threads running simultaneously take advantage of data brought to cache by another thread.

Next, we compare the cache behavior of the code (product of a sparse matrix by a set of dense vectors) for original and reordered matrices. In order to complete the comparison, we have also included reordered matrices obtained after applying standard methods such as *Reverse Cuthill-McKee* (RCM), *Approximate Minimum Degree* (AMD) and a particular implementation of the nested dissection ordering algorithm included in the METIS package [32]. Figures 6(a) and 6(b) show, as an example, the cache misses in L1 and L2 caches respectively using 8 threads. These measures are normalized with respect to the values obtained by the original matrices. Results for L3 are not displayed because no conflict misses are produced in L3 given its large size, that is, there are only capacity misses in L3. Important reductions in the number of L1 and L2

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

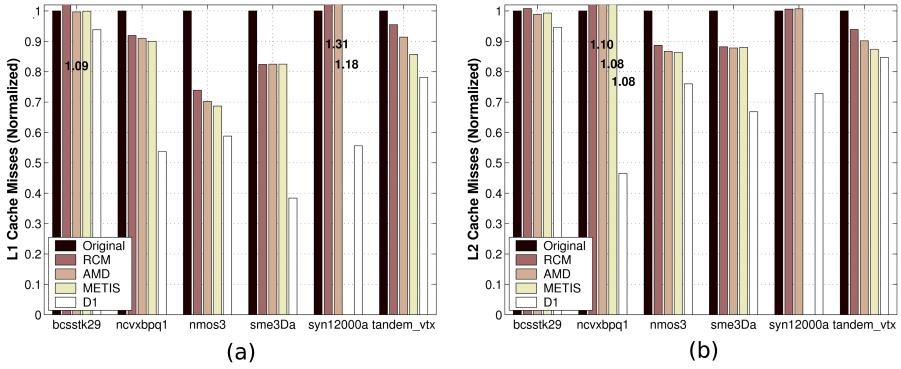


Figure 6. Cache behavior using 8 threads on the SMT simulator: (a) L1 misses and (b) L2 misses.

cache misses have been achieved by the reordered matrices obtained after applying the locality improvement technique. The average of the reductions using 8 threads and D_1 matrices is 36% for the L1 cache and 27% for the L2 cache. These reductions are especially noticeable in the L1 for the *sme3Da* reordered matrix (up to 61% with respect to the original one). Note that as the number of threads increases, the improvements in the number of cache misses become more significant. For example, for L1, reductions in average range from 28% to 36% when using 1 and 8 threads, respectively. For L2, the average reductions increase from 15% for 1 thread to 27% for 8. The reordering is even effective when only one thread is running.

The standard reordering techniques exhibit a very irregular behavior. With respect to the L1 cache, in two cases (*bcsstk29* and *syn12000a*) an increase in the number of misses is obtained with these standard techniques. Nevertheless, for the other matrices decreases in the number of cache misses can be observed, although they are always lower than the ones obtained using the proposed technique. For the L2 cache, the behavior is similar. Note that the reordering performed using METIS can only be applied to matrices with symmetric sparsity pattern (it is not the case for matrix *syn12000a*).

Next, the performance in terms of the speedup achieved by the original and the reordered matrices running from 1 to 8 threads is shown. The results obtained by reordered matrices using the proposal developed in this work and standard methods (RCM, AMD and METIS) were also compared in Figure 7. In order to define the speedup we consider as reference the execution time of the original matrices running only one thread. As an example, note that using 8 threads, speedups up to 3.4 have been reached in some cases with matrices reordered with our technique, while for the standard methods the speedup is never higher than 2.8. Moreover, our reordered matrices always outperform those obtained by standard methods. The average speedup obtained using matrices reordered by the developed technique (D_1) is about 2.2 for 4 threads, and 2.9 for 8 threads, while, for example, the speedup in the case of RCM is just 1.8 and 2.4 when using 4 and 8 threads, respectively.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

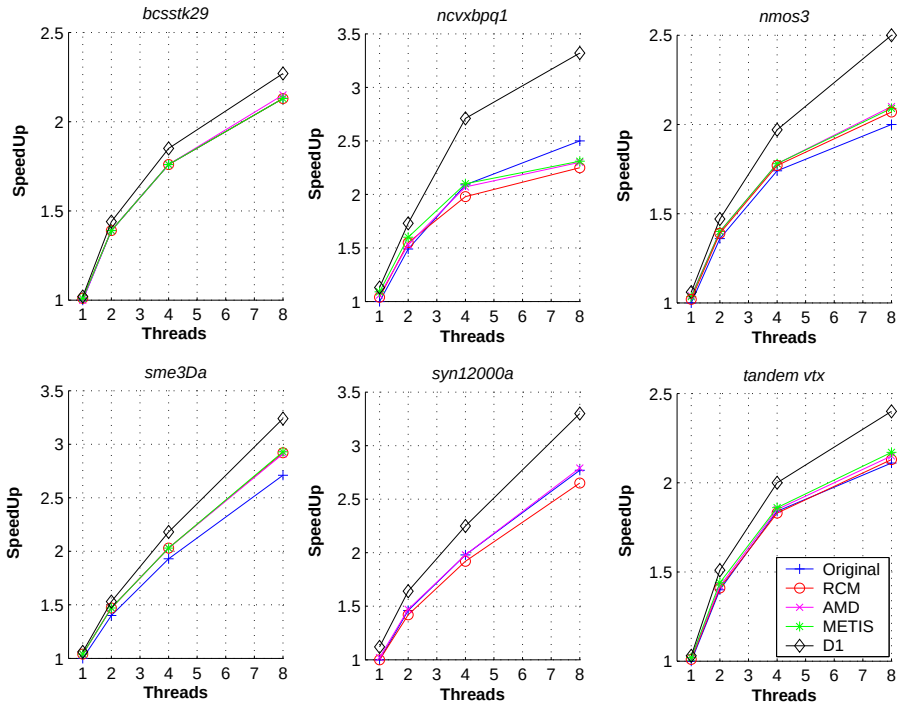


Figure 7. Speedup obtained on the SMT simulator.

From these results we can conclude that our technique increases the inter-thread and the intra-thread localities. The reason is that the conflict misses are reduced when several threads are running simultaneously on a SMT architecture. As a consequence, important reductions in the execution time have been achieved. Moreover, our reordering technique improves the scalability of the code allowing better speedups as the number of running threads increases.

6. PERFORMANCE EVALUATION ON REAL SMT ARCHITECTURES

In this section, first, we present a brief description of the Hyper-Threading technology of the Intel Pentium 4 processors family. The Pentium 4 is the first commercially available general-purpose processor to implement a simultaneously multithreading core [33]. Afterwards the experimental conditions are detailed, and finally, the reordering technique overhead and the experimental results obtained after applying our technique are shown.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

6.1. Hyper-Threading technology

Hyper-Threading technology (HT) brings the SMT approach to the Intel architecture [8]. The Xeon processor family was the first to implement HT, and later the technology was also incorporated to the desktop and the mobile version of the Pentium 4. HT technology makes a single physical processor appear as two logical processors. In order to do this, there is one copy of the architecture state for each logical processor, and the logical processors share a single set of physical resources. The architecture state consists of registers including general-purpose registers, control registers, advanced programmable interrupt controller (APIC) registers, and some machine state registers. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses. Consequently, from a software perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multiprocessor system. From a microarchitecture perspective, this means instructions from logical processors will persist and execute simultaneously on shared execution resources.

The main difference between the HT implementation and the SMT architecture proposed by Tullsen *et al.* (see Section 1) is the way of sharing the hardware structures [34]. While the research with SMT indicates that virtually all structures are more efficient when they are dynamically shared among threads rather than statically partitioned, some structures in the HT implementation are statically divided. The reasons for this difference are basically two. First of all, static partitioning isolates a thread from a poorly behaving co-scheduled thread more effectively. And second, any negative effects of the partitioned approach are minimized with only two hardware contexts. The HT implementation does not scale well to a larger number of hardware contexts, such as the eight threads that were typically modeled in the SMT research. However, with two threads, a scheme that uses the entire instruction queue in single-thread mode, and divides it in half for dual-thread mode provides a reasonable approximation of the performance of a dynamically shared queue. With more threads, the static partitioning would likely hamper performance excessively. Accesses to several non-duplicated, non-partitioned parts of the pipeline are handled in a round-robin manner.

Like in the SMT case described in Section 5, when HT is enabled, the simultaneous execution of several threads puts a lot of stress on the memory hierarchy. When running two threads, the number of inter-thread cache misses can be critical in the performance of the application. This behavior is even more important in the case of irregular codes due to their poor locality in the accesses. Therefore, techniques that improve the locality also become very important for SMT architecture implementations as for the case of HT technology.

6.2. Experimental setup

Results in this section were obtained using a single 3.2 GHz HT Pentium 4 processor with 1 GB of memory. This processor belongs to the Prescott processors family. It provides a 16KB 8-way L1 data cache and an unified 1024KB 8-way L2 cache. The cache latencies are 3 and 28 cycles respectively, and the line size is 64 bytes in both cases. Moreover, it has a 12K-μops 8-way trace cache.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

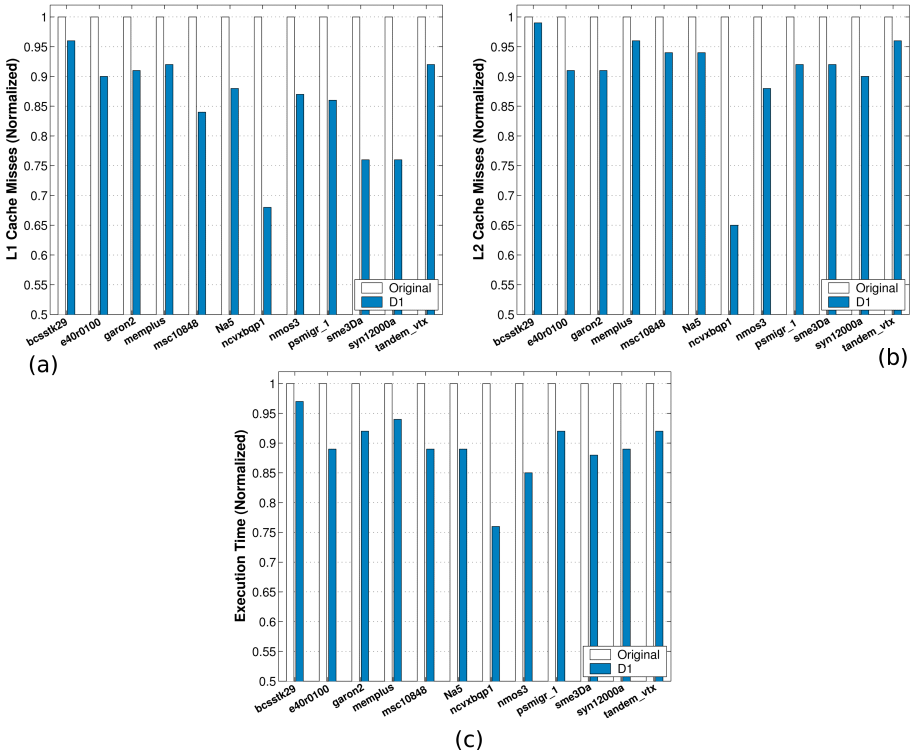


Figure 8. Cache behavior and execution time obtained using the 2-HT configuration.

All the codes were written in C and compiled with the Intel's 8.1 Linux compiler. OpenMP directives were used to parallelize the irregular code of Figure 1. In the parallel code performing the product of a sparse matrix by a set of dense vectors, a cyclic distribution of the iterations of loop i among the threads is used. Furthermore we have used the PAPI library [35] to read the hardware performance counters of the Pentium 4 processor.

6.3. Experimental results

Three different running configurations were available: sequential (one thread and HT disabled), 1-HT (one thread and HT enabled) and 2-HT (two threads and HT enabled). The behavior of the sequential and the 1-HT configurations are very similar.

First, we compare the results obtained using our technique with respect to the results when using the original matrices from our test set (see Section 5.1 for details) on the Pentium 4 processor. The behavior of the cache and the execution times using the 2-HT configuration are

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

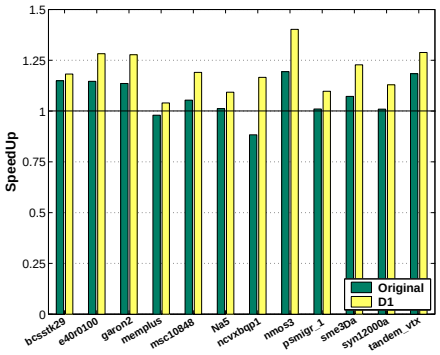


Figure 9. Speedup obtained using the 2-HT configuration.

shown in Figure 8. The results of the reordered matrices are normalized taking as reference the original ones. Important reductions in terms of the number of L1 and L2 cache misses are observed (Figures 8(a) and 8(b), respectively). These reductions are especially noteworthy for the L1 cache, where they are up to 37% in some cases. The average reduction of misses for the L1 is about 16%, and for the L2, 9%. As shown in Figure 8(c), the improvement in the cache behavior of the reordered matrices is reflected in the execution time. The average execution time reduction is about 12%, and ranges from 3% to 24%. Therefore, reordered matrices provide a significant improved global performance with respect to the original ones when using a 2-HT configuration.

Finally, speedups obtained by the reordered matrices using the 2-HT configuration with respect to the sequential case using the original matrices are depicted in Figure 9. Speedups are always over 1 for the reordered matrices. Note that for *memplus* and *ncvxbpq1* speedups over 1 are achieved, unlike the original ones. The average speedup using the reordered matrices increases up to 1.2, ranging from 1.05 to 1.4.

6.4. Reordering technique overhead

We have analyzed the time required by our technique in order to perform the matrix reordering. The cost of the reordering strongly depends on different parameters of the Lin-Kernighan heuristic, such as the convergence criterion or the starting solution. Moreover, it also depends on the pattern of the considered sparse matrix. Therefore, it is very difficult to obtain an accurate characterization of this cost. However, in order to estimate the overhead introduced by the reordering technique for all the cases studied in this work (see Table III for details of the matrix test set), Table IV is introduced. The table displays the reordering times normalized with respect to the time required to perform a sparse matrix-vector product. According to these results, the cost of reordering the sparse data structure must be amortized when the sparse operation is repeatedly performed, as for instance, in iterative methods, which usually

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



Table IV. Overhead of the reordering technique.

Matrices	Number of SpM×V
bcsstk29	394
e40r0100	440
garon2	317
memplus	177
msc10848	348
Na5	94
ncvxbpq1	773
nmos3	427
psmigr_1	334
sme3Da	512
syn12000a	523
tandem_vtx	366
Average	388

require thousands of sparse matrix-vector multiplications. The tests have been performed using the system detailed in Section 6.2.

7. CONCLUSIONS AND FUTURE WORK

Simultaneous Multithreading (SMT) is one of the most successful implementations of the multithreaded architectures. In SMT architectures the locality of the data accessed by different threads plays a crucial role given that the simultaneous execution of several threads puts a lot stress on the memory hierarchy, and a poor locality, both inter-thread and intra-thread, may become a major bottleneck.

This paper analyzes and improves the execution properties of irregular codes on SMT architectures. As a case of study, we consider the product of a sparse matrix by a set of dense vectors.

The improvement process is based on the application of a heuristic technique that reorders the sparse data structure. This process is guided by a quality function which is evaluated on the pattern of the sparse matrix involved in the calculations. The cost of reordering the sparse data structure is high, but can be amortized when the sparse operation is repeatedly performed, as for instance, in iterative methods. Nevertheless, we are currently working on performing a simpler reordering, avoiding the movement of individual rows or columns of the matrix by grouping them.

The number of cache misses, the execution time and the speedup varying the number of threads are studied. First, results using the SMTSIM simulator of a SMT architecture were achieved, and they were compared to those obtained by standard reordering methods like Reverse Cuthill-McKee, Approximate Minimum Degree and METIS. Next, the data reordering technique is evaluated on a real implementation of a simultaneous multithreading architecture: the Intel's Hyper-Threading. In both cases the increase of the locality achieved reduces

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

meaningfully the number of cache misses and the execution time, while increasing the speedup as the number of running threads grows. For example, for the case of the simulator using 8 threads, the average reduction in the number of L2 cache misses is about 27%, and 20% for the execution time. These results were obtained considering sparse matrices with a wide variety of patterns and characteristics.

Note that the proposed reordering technique could be applied to other sparse matrix codes, whose locality properties depend on the pattern of a sparse matrix. In the same way, the technique could also be adapted to other parallel architectures, such as multicore and constellations.

ACKNOWLEDGEMENTS

The authors would like to thank Diego Puppin for his useful help with the SMT simulator.

REFERENCES

1. Tullsen DM, Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *ISCA*, 1996; 191–202.
2. Lo JL, Emer JS, Levy HM, Stamm RL, Tullsen DM. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems* 1997; **15**(3):322–354.
3. Tullsen DM, Eggers S, Levy HM. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the22th Annual International Symposium on Computer Architecture*, 1995; 392–403.
4. Hily S, Seznec A. Standard memory hierarchy does not fit Simultaneous Multithreading. *Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4)*, 1998.
5. Baker AH, Jessup ER, Manteuffel T. A technique for accelerating the convergence of restarted GMRES. *SIAM J. Matrix Anal. Appl.* 2005; **26**(4):962–984.
6. Guennounia AE, Jbilou K, Sadok H. The block lanczos method for linear systems with multiple right-hand sides. *Applied Numerical Mathematics* 2004; **51**(2–3):243–256.
7. Saad Y. *Iterative methods for sparse linear systems*. SIAM (Society for Industrial and Applied Mathematics), 2003.
8. Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, Upton M. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal Q1* 2002; .
9. Cuthill E, McKee J. *Several strategies for reducing the bandwidth of matrices*. Rose and Willoughby, 1972.
10. Amestoy PR, Davis TA, Duff IS. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications* 1996; **17**(4):886–905.
11. Oliker L, Li X, Husbands P, Biswas R. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review* 2002; **44**(3):373–393.
12. Fraguela BB, Doallo R, Zapata EL. Memory hierarchy performance prediction for blocked sparse algorithms. *Parallel Processing Letters* Sep 1999; **9**(3):347–360.
13. Navarro JJ, García E, Larriba-Pey JL, Juan T. Block algorithms for sparse matrix computations on high performance workstations. *Proc. IEEE Int'l. Conf. on Supercomputing (ICS'96)*, 1996; 301–309.
14. Toledo S. Improving memory–system performance of sparse matrix–vector multiplication. *Proc. of the 8th SIAM Conf. on parallel processing for scientific computing*, 1997.
15. Pinar A, Heath M. Improving performance of sparse matrix-vector multiplication. *Proc. of Supercomputing*, 1999.
16. Im EJ, Yelick K. Optimizing sparse matrix vector multiplication on SMPs. *Proc. of the 10th SIAM Conf. on parallel processing for scientific computing*, 1999.
17. Pichel JC, Heras DB, Cabaleiro JC, Rivera FF. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing* 2005; **31**(8–9):858–876.
18. Strout MM, Carter L, Ferrante J, Kreaseck B. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications* 2004; **18**(1):95–114.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



19. Suh G, Devadas S, Rudolph L. Dynamic cache partitioning for simultaneous multithreading systems. *Proceeding of the Thirteenth IASTED International Conference on Parallel and Distributed Computing System*, 2001.
20. Nikolopoulos DS. Code and data transformations for improving shared cache performance on SMT processors. *Int. Symposium on High Performance Computing*, 2003; 54–69.
21. Puppin D, Tullsen DM. Maximizing TLP with loop-parallelization on SMT. *5th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.
22. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrix-vector multiply on emerging multicore platforms. *Proc. of Supercomputing (SC)*, 2007.
23. Heras DB, Cabaleiro JC, Rivera FF. Modeling data locality for the sparse matrix-vector product using distance measures. *Parallel Computing* 2001; **27**:897–912.
24. McKinley KS, Temam O. A quantitative analysis of loop nest locality. *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.
25. Lo JL, Eggers SJ, Levy HM, Parekh SS, Tullsen DM. Tuning compiler optimizations for simultaneous multithreading. *International Symposium on Microarchitecture*, 1997; 114–124.
26. Duff I, Heroux M, Pozo R. An overview of the sparse basic linear algebra subprograms: the new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software* 2002; **28**(2):239–267.
27. Im EJ, Yelick KA, Vuduc R. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications* February 2004; **18**(1):135–158.
28. Pichel JC, Heras DB, Cabaleiro JC, Rivera FF. Improving the locality of the sparse matrix–vector product on shared memory multiprocessors. *Euromicro Conf. on Parallel, Distributed and Network-based Processing, PDP2004*, 2004; 66–71.
29. Heras DB, Blanco V, Cabaleiro JC, Rivera FF. Modelling and improving locality for the sparse matrix-vector product on cache memories. *Future Generation Computer Systems. Special Issue on High Performance Numerical Methods and Application* 2001; **18**(1):55–67.
30. Applegate D, Bixby R, Chvátal V, Cook W. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.
31. Davis T. University of Florida Sparse Matrix Collection. *NA Digest* Jun 1997; **97**(23). `http://www.cise.ufl.edu/research/sparse/matrices [15 October 2007]`.
32. Karypis G, Kumar V. *METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices* 1997.
33. Hinton G, Sager D, Upton M, Boggs D, Carmean D, Kyker A, Roussel P. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1* 2001.
34. Tuck N, Tullsen DM. Initial observations of the simultaneous multithreading Pentium 4 processor. *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 2003.
35. Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications* 2000; **14**(3):189–204.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls