Reordering Algorithms for Increasing Locality on Multicore Processors

Juan C. Pichel, David E. Singh and Jesús Carretero
Computer Science Dpt.
Universidad Carlos III de Madrid, Spain
{*jcpichel,desingh,jcarrete*}@*arcos.inf.uc3m.es*

Abstract

In order to efficiently exploit available parallelism, multicore processors must address contention for shared resources as cache hierarchy. This fact becomes even more important when irregular codes are executed on them, which is the case for sparse matrix ones.

In this paper a technique for increasing locality of sparse matrix codes on multicore platforms is presented. The technique consists on reorganizing the data guided by a locality model which introduces the concept of windows of locality. The evaluation of the reordering technique has been performed on two different leading multicore platforms: Intel Core2Duo and Intel Xeon.

Experimental results show important performance improvements when using our reordered matrices with respect to original ones. In particular, an average execution time reduction of about 30% is achieved considering different number of running threads. These results are due to an improved overall cache behavior. Likewise, a comparison of our proposal with some standard reordering techniques is included in the paper. Results point out that the reordering technique always outperforms standard algorithms and is effective for matrices with any structure.

1 Introduction

Due to scaling limitations of uniprocessors, modern processors are now moving to Chip Multiprocessor (CMP) architectures in order to extract more performance from available chip area. CMPs contain multiple cores on a single chip allowing more than one thread to be executed at a time. Each core has its own resources as well as shared resources. Several recent studies have shown that in order to efficiently exploit available parallelism, CMPs must address contention for shared resources [3, 24]. In particular, CMPs typically share the L2 cache and its lower level memory hierarchy. Consequently, reuse of data cached on-chip is especially important in CMPs in order to reduce off-chip accesses, which compete for the available memory bandwidth [4].

Therefore, data locality, both inter-thread and intra-thread, is especially relevant in CMPs architectures. This fact becomes even more important when irregular codes are executed on them. These codes are characterized by irregular accesses, and they tend to have low spatial and temporal localities. So low performance can be expected when executing these codes on a CMP architecture.

In this paper, we propose a technique to deal with the problem of increasing the locality of irregular codes as sparse matrix ones on multicore platforms. The technique consists on 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 sparse matrix pattern 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.

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 practice when there are multiple right-hand sides in an iterative solver, in recently proposed blocked iterative solvers, and in blocked eigenvalue algorithms such as block-Lanczos or block-Arnoldi, among other important codes [18].

In order to evaluate our reordering technique two different leading multicore platforms are considered: Intel Core2Duo and Intel Xeon. The first corresponds to a typical dual-core CMP architecture with a shared L2 cache. While dual-core Xeon processor belongs to Chip Multithreading (CMT) architectures [19]. CMT processors combine Chip Multiprocessing (CMP) and Simultaneous Multithreading (SMT) [22]. SMT allows several independent threads to issue instructions to a superscalar's multiple functional units in a single cycle. In particular, Xeon's implementation of SMT architecture is Intel's Hyper-Threading (HT) [12].

2 Related work

SMT, CMP and CMT architectures have been well studied and analyzed in order to determine their potential bottlenecks. In the analysis of SMT processors, several works point out that the simultaneous execution of several threads puts a lot of stress on the memory hierarchy [8, 21]. For this reason cache contention cannot be ignored because it leads

```
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.

to overestimate the performance of the considered applications on a SMT architecture.

Recent years have witnessed several investigations regarding to CMP and CMT platforms. For example, Chen *et al.* [4] analyze the impact of different thread schedulers in order to reduce the number of off-chip cache misses. In other work [3], the authors study the impact of L2 cache sharing with multiprogram workloads. They propose several performance models to predict the impact of cache sharing on co-scheduled threads. In addition, other papers have analyzed the performance of OpenMP applications on CMP and CMT platforms [6, 13]. But, as in the SMT case, research is focused on regular applications.

Nevertheless, for uniprocessor and multiprocessor systems, we can find in the literature many works that deal with the problem of locality for irregular codes. Proposed techniques can be mainly divided into two categories: 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 [14]. Some authors have demonstrated that both groups of techniques are complementary with successful results in terms of the exploitation of the memory hierarchy [16, 17, 20].

In a recent work [23], 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 among others: software pipelining, prefetching approaches, register and cache blocking, etc. Nevertheless, they do not evaluate data reordering techniques in order to increase inter and intra-thread locality.

3 Locality Properties of the Product of a Sparse Matrix by a Set of Vectors

In this work the process of locality improvement on multicore 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 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) [18]. 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 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 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 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). So, B presents irregular accesses whose locality properties depend directly on the pattern of the sparse matrix. 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.

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 [9]. 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 [16] 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.

4 Reordering for Multicore Architectures

In this section we introduce a brief summary of the locality model in which the proposed reordering technique is based on. This model was previously applied to different computer platforms such as uniprocessors, shared and distributed memory multiprocessors [15, 16]. Next, a detailed description of the locality optimization technique for multicore processors is presented.

4.1 Locality Model

The locality model is based on the evaluation of the data locality for the considered sparse matrix code. The model is general enough to be applied to any sparse matrix code that can take profit of a clustering of entries in the pattern of the matrix.

In the model, locality is measured for consecutive pairs of rows or columns of the sparse matrix depending on

whether the prevailing irregular accesses to the sparse matrix are row-wise or column-wise. In both cases, the locality is based on two parameters: the number of *entry matches* (a_{elems}) and the number of *block matches* (a_{blocks}). For instance, considering accesses to the sparse matrix by rows, 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. The concept of entry matches can be extended to block matches by considering instead of single entries (an entry is defined as a nonzero element), pieces of consecutive positions in a row of the matrix pattern of the size of a cache line where there is at least one entry.

Based on these two parameters a magnitude called *distance between rows x and y* can be defined, denoted as $d_i(x, y)$. It is used to measure the locality displayed by the irregular accesses performed by the irregular code on these two rows when they are consecutively accessed. From different distance definitions proposed in [7], in this work we consider the following:

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

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

For a given sparse matrix accessed by rows, a quantity that is inversely proportional to the locality of the data for the whole sparse matrix can be defined as follows:

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

These definitions can be directly extended to columns. The model is suitable for any sparse matrix without limitations in the type of pattern that they present.

The locality model detailed above only provide results based on the locality evaluated on pairs of consecutive rows (or columns) of the sparse matrix. Nevertheless, the reuse of data could be possible in any level of the memory hierarchy during the product of two or more consecutive rows (or columns) of the matrix. For this reason, a generalization of the distance functions based on the concept of *windows of locality* is presented.

A *window of locality* is a set of w consecutive rows (or columns) of the matrix between which there is a high probability of data reuse when executing the sparse matrix code. Based on the distance function $d_1(x, y)$, we can define the distance between windows of locality g and h as:

$$d_{w1}(g, h) = n(g) + n(h) - 2{*}a_{\text{elems}}(g, h) \quad (3)$$

where $n(g) = n_{\text{elemsw}}(g) - a_{\text{elemsw}}(g)$. Parameter $a_{\text{elems}}(g, h)$ is a direct extension of the entry matches between windows g and h. $n_{\text{elemsw}}(g)$ is the number of elements of window g, and $a_{\text{elemsw}}(g)$ generalizes the concept of entry matches considering matches that take place on two or more rows within window g. Note that, introducing $n(g)$ the possible reuse of data inside g is also considered. Figure 2 shows an example of the calculation of these parameters when $w = 2$.

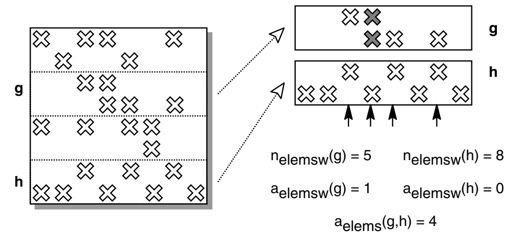


Figure 2. Example of the calculation of $n_{\text{elemsw}}(g)$, $a_{\text{elemsw}}(g)$ and $a_{\text{elems}}(g, h)$.

These distances are equivalent to distances measured over pairs of consecutive rows (or columns) of the matrix when the window size is $w = 1$. This way, distances evaluated on windows of locality preserve the same properties. Therefore, the indirect estimation of locality defined for a sparse matrix in Equation 2 can now be calculated as a sum over the whole matrix:

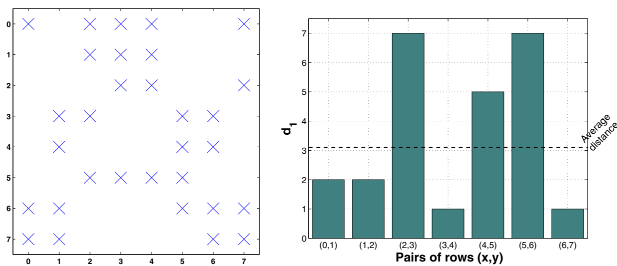
$$D_{w1} = \sum_{g} d_{w1}(g, g+1), \quad \forall\, g \mid 0 \leq g < \lceil N/w \rceil \quad (4)$$

where N is the number of rows or columns of the sparse matrix. The model described can be applied to any sparse matrix code whose locality is determined by the entries grouping level in the matrix pattern such as the product analyzed. 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 results. The final 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.

4.2 Data Reordering Algorithm

We propose a heuristic technique that modifies the pattern of the sparse matrix according to the locality model described before. In order to increase the locality in the accesses performed by the irregular code, we search for a permutation of rows (and columns) of the matrix that minimizes its total distance D_{w1} (Equation 4). We formulate the problem of locality improvement as a classic NP-complete optimization problem, and we solve it as a graph problem using its analogy to the *Traveling Salesman Problem* (TSP).

The problem is described using a complete weighted graph where each node represents a window of locality of the input sparse matrix. Each edge of the graph has an associated weight that reflects the distance between pairs of windows of locality 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 cases where $a_{elems} = 0$. Those values, according to the distance definitions, represent the worst cases regarding to locality. So, without losing relevant information about locality, we can use an incomplete weighted graph where only values of a_{elems} different



(a) Sparse matrix example (b) Distance histogram

Figure 3. Example of the creation of windows of locality of variable size.

from zero are considered. As a consequence, the graph size is noticeably reduced.

Solving the problem of reordering is equivalent to find 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 matrix. Note that if the pattern of the matrix is symmetric just one permutation vector is required, that is, the same ordering is applied by rows and columns. On the other hand, given that we have measures (distance values) to validate the quality of an ordering, we have opted for focusing on heuristic solutions. After a comparative study of different techniques we have chosen the *Chained Lin–Kernighan* heuristic proposed by Applegate et al. [1].

In order to select the window size (w), two types of windows of locality are analyzed: fixed and variable. When windows of fixed size are considered, the number of nodes in the weighted graph is $\lceil N/w \rceil$. Therefore, for high values of w, graph is noticeably reduced. It implies an important decrease in the computational time needed for its calculation, together with reductions in the problem size to manage by the reordering heuristic. Note that we are not taking into account any locality property of the input sparse matrix in order to create the windows. Therefore, windows of locality can be formed by consecutive rows (or columns) of the matrix which exhibit low locality according to our model. This way, as it will be shown in Section 5, windows of locality with a fixed size present a strong dependence with the matrix pattern as values of w become higher. For this reason, windows of variable size are introduced.

The objective of using windows of locality of variable size is two-fold. On one hand, as in the fixed size case, to decrease the number of nodes in the weighted graph, which benefits are detailed in the above paragraph. On the other, to avoid the grouping of consecutive rows (or columns) of the matrix with low locality in the windows creation process. Figure 3 shows an example of the proposed technique to create windows of variable size considering the rows of the matrix. First, a histogram is created from the input matrix. It represents the distance between each pair of consecutive rows. Therefore, there are $N - 1$ values in the histogram.

Core Architecture	Intel Core2Duo	Intel Xeon
Clock (GHz)	2.13	2.6
Cores	2	2
L1D Cache	32 KB	16 KB
L2 Cache	2 MB (shared)	2 MB (1 MB/core)
L3 Cache	–	4 MB (shared)
Hyper-Threading	No	Yes

Table 1. Architectural summary of Intel Core2Duo and Intel Xeon.

In order to decide if two consecutive rows x and y will be included within the same window a simple criterion must be fulfilled: $d_1(x, y) < D_1/N$, that is, the distance must be lesser than the average distance of the whole sparse matrix. According to this, in the example of Figure 3, four windows of locality are created: $\{0, 1, 2\}$, $\{3, 4\}$, $\{5\}$ and $\{6, 7\}$. This way, windows creation process is guided by the locality model and, as a consequence, locality among rows within each window is increased. This process can be directly extended to columns. Note that when the matrix pattern is unsymmetric, the windows creation process must be applied considering rows and columns independently.

5 Performance Evaluation

In this section, first, the experimental conditions are established. Finally, an evaluation of the performance obtained by the reordered matrices generated after applying our reordering technique is presented. An analysis of the overhead introduced by the reordering process is also included.

5.1 Experimental Conditions

The proposed data reordering technique has been tested on two different leading multicore processors: Intel Core2Duo (Conroe) and Intel Xeon (Tulsa). The main characteristics of each architecture are summarized in Table 1. Note that Xeon processor supports Hyper-Threading (HT) technology. This feature allows a single, physical processor to appear as two logical processors to the operating system, where each processor maintains a separate run queue. Therefore, when HT is enabled, two hardware contexts per core are active simultaneously, competing each cycle for all available resources.

All the codes were written in C and compiled with the Intel's 10.0 Linux compiler. OpenMP directives were used to parallelize the irregular code of Figure 1. Furthermore, we have used the PAPI library [2] to read the hardware performance counters of the Core2Duo processor. PAPI is not available for dual-core Xeon processor.

As a test set to evaluate the locality optimization technique we have selected fourteen square sparse matrices from different real problems that represent a variety of nonzero patterns. These matrices are from the University of Florida Sparse Matrix Collection (UFL) [5]. Table 2 summarizes some features of the matrices, where N is the num-

	N	N_Z		N	N_Z
msc10848	10848	1229778	nc5	19652	1499816
sme3Da	12504	874887	nmos3	18588	386594
av41092	41092	1683902	mixtank_new	29957	1995041
gyro_k	17361	1021159	psmigr_1	3140	543162
syn12000a	12000	1436806	e40r0100	17281	553562
garon2	13535	390607	rajat15	37261	443573
bcsstm36	23052	320606	Na5	5832	305630

Table 2. Matrix benchmark suite.

ber 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 commented 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.

Reorderings are performed using both types of windows of locality: fixed and variable size. Specifically, in this paper we have considered windows of five fixed sizes: $w = 1, 2, 8, 32$ and 128. For comparison purposes, we have also included reordered matrices obtained after applying standard methods as *Reverse Cuthill-McKee* (RCM), *Approximate Minimum Degree* (AMD) and a particular implementation of the nested dissection ordering algorithm included in the METIS package [10]. Note that METIS algorithm can only be applied to matrices with symmetric pattern. Results shown in this section are always expressed in terms of the improvement percentage with respect to the original matrices.

5.2 Intel Core2Duo Performance

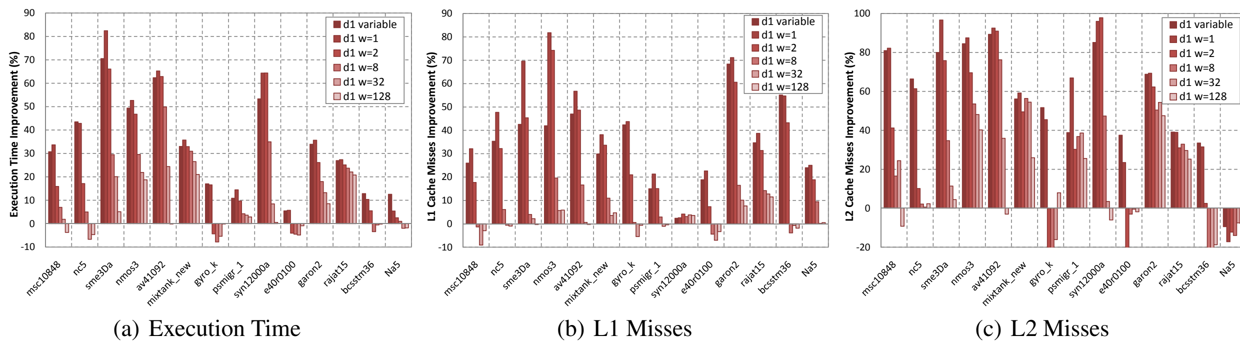
In this platform, L2 cache is shared among cores. Therefore, sharing data among threads will lead to an improved performance of the considered parallel code. Lo *et al.* [11] demonstrate that threads should be parallelized with a cyclic, rather than a blocked algorithm in a SMT architecture where cache hierarchy is shared, as in the Core2Duo processor. For this reason, in order to parallelize the code of Figure 1, a cyclic distribution of the loop i iterations among the threads is used. This way, consecutive rows of the sparse matrix are accessed by different threads, which allows the reuse of data (elements of the dense matrix B) previously accessed in the shared cache by a different thread.

In order to evaluate the data reordering technique, both execution time and cache behavior are analyzed. As example, Figure 4 shows the experimental results achieved when two threads are used. An overview of these results indicates that the best performance is achieved by reorderings performed using windows of variable size or windows of small size (especially, when $w = 1$ is considered). Normally, as the windows size grows, the performance is degraded. We find the explanation of this behavior when windows of fixed size are constituted (see Section 4.2), because rows and columns of the sparse matrix are grouped without taking into account any locality property. Therefore, distance among rows (or columns) within a window depends directly on the matrix pattern. As a consequence, larger sizes only are effective when sets of consecutive rows or columns with a high probability of data reuse (low distances according to our locality model) are permuted. This is the case of some finite element matrices (FEM), such as nmos3, mixtank_new, garon2 or rajat15.

A more detailed analysis of the results shows that only those matrices reordered using windows of variable size and windows with $w = 1$ reduce the execution time with respect to the original matrices for all considered cases (see Figure 4(a)). This is due to an improved cache behavior, which implies important reductions in the number of L1 an L2 cache misses (Figures 4(b) and 4(c) respectively). For example, decreases up to 80% for L1 (matrix nmos3) and 90% for L2 (matrix av41092) are achieved when using windows of variable size. Note that while L1 misses refer only to intra-thread locality (there is a L1 cache per core), L2 misses indicate the influence of both intra-thread and inter-thread locality. As a consequence, performance of the sparse operation is noticeably increased, achieving, in some cases, reductions higher than 80% in the total execution time (matrix sme3Da).

Next, a summary of the results obtained on the Core2Duo processor is displayed in Table 3 in order to estimate the benefits of our proposal in comparison with standard reordering methods. Best overall results are obtained by our reordering technique using windows with $w = 1$, followed by the reorderings performed using windows of variable size. In both cases, reductions higher than 50% in the number of L2 misses are achieved, indicating a high level of data reuse in the shared cache. Note that our technique presents a very good behavior even in the sequential case (one thread). The main drawback of using windows of small fixed size (especially with $w = 1$) with respect to windows of variable size is, as we show in Section 5.4, a higher computational time required to perform the reordering.

Regarding to the results provided by the standard techniques several conclusions can be made. First, our proposal outperforms standard techniques except for large window sizes. At the same time, these algorithms show an irregular cache behavior with remarkable improvements in some cases (for example, matrices sme3Da and garon2), while a degraded performance is showed in others (for example, with matrices e40r0100 and Na5). Note that METIS algorithm can only be applied to matrices with symmetric pattern. Nevertheless, even if we consider only this type of matrices, results displayed in Table 3 would not vary significantly. For example, using two threads and considering windows of variable size and windows with $w = 1$, the



(a) Execution Time (b) L1 Misses (c) L2 Misses

Figure 4. Results obtained on the Core2Duo platform using two threads (improvement percentage with respect to the original matrix).

	1 Thread			2 Threads		
	Exec. Time	L1 Misses	L2 Misses	Exec. Time	L1 Misses	L2 Misses
d_1 *variable*	27.8	36.3	51.5	33.0	34.5	57.3
$d_1\ w = 1$	30.8	45.8	51.7	35.2	43.3	59.6
$d_1\ w = 2$	21.8	33.5	31.7	25.8	32.4	34.2
$d_1\ w = 8$	13.0	9.7	24.7	15.2	6.7	23.3
$d_1\ w = 32$	6.7	1.0	19.1	9.5	1.1	17.3
$d_1\ w = 128$	2.8	1.3	12.9	5.4	1.6	9.5
AMD	11.7	15.3	25.6	10.2	13.4	17.1
RCM	13.2	16.2	29.9	14.9	14.9	21.2
METIS*	15.9	14.1	22.5	16.3	12.9	27.3

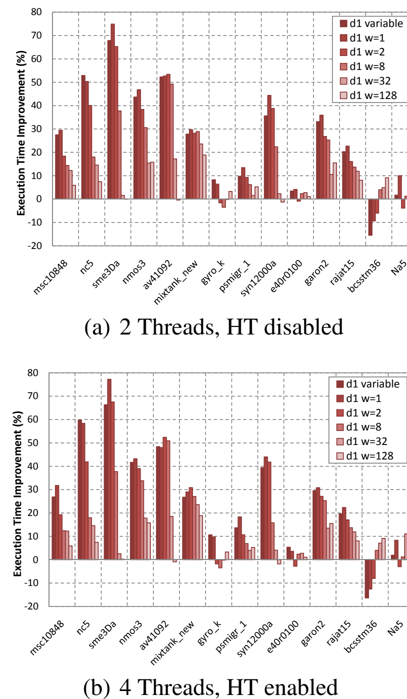
Table 3. Summary of the Core2Duo results (improvement percentage with respect to the original matrix).

average execution time reduction is 32.5% and 33.8% respectively.

5.3 Intel Xeon Performance

On this platform three different running configurations are analyzed: serial execution (one thread), two threads with HT disabled, and four threads with HT enabled. Xeon processor contains three cache levels (see Table 1 for details). Unlike Core2Duo processor, L2 is not shared among cores. Therefore, threads assigned to different cores cannot take profit of data reuse in L2, only in the shared L3. For this reason, we have opted for a different parallelization strategy of Figure 1 code on this platform. When two threads are running in parallel, a block distribution of the loop i iterations is used. This distribution is preferred to cyclic one because it favors the intra-thread locality, allowing each thread to reuse elements of the dense matrix B previously accessed by itself. Nevertheless, when HT is enabled, two threads are mapped to each core sharing the whole cache hierarchy. In this case, we use an hybrid parallelization of loop i. First, a block distribution among cores is used, while within each core, iterations are cyclicly distributed among threads assigned to it. This way, threads mapped to the same core exploit inter-thread and intra-thread localities.

Figure 5 shows the experimental results achieved when two threads (on the left) and four threads (on the right) are used. In both cases overall behavior is similar. As



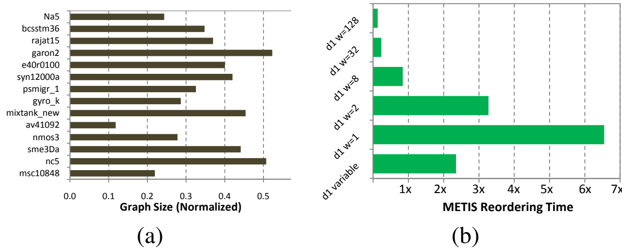
(a) 2 Threads, HT disabled

(b) 4 Threads, HT enabled

Figure 5. Results obtained on the Xeon platform (improvement percentage with respect to the original matrix).

	Execution Time		
	1 Thread	2 Threads, HT off	4 Threads, HT on
d_1 $variable$	25.7	25.8	26.7
d_1 $w = 1$	27.5	28.7	29.4
d_1 $w = 2$	21.9	22.6	23.7
d_1 $w = 8$	17.2	17.5	18.0
d_1 $w = 32$	9.0	9.0	9.2
d_1 $w = 128$	7.0	6.9	6.9
AMD	13.8	11.2	11.0
RCM	14.3	9.2	9.2
METIS*	15.8	12.9	12.2

Table 4. Summary of the Xeon results (improvement percentage with respect to the original matrix).



(a) (b)

Figure 6. Analysis of the overhead introduced by the reordering technique: (a) graph size using windows of variable size, and (b) comparison of the reordering time.

in the Core2Duo results, best performance is obtained by reorderings performed using windows of variable size and windows with small fixed sizes. Nevertheless, here there is one case where only reorderings using bigger window sizes improve the performance with respect to the original matrix (matrix `bcsstm36`). Reductions in the execution time slightly higher than 70% are reached on this platform (matrix `sme3Da`). If we compare these results and those obtained on Core2Duo processor (Figure 4), we conclude that the best results are achieved by the same set of matrices, that is, matrices `nc5`, `sme3Da`, `nmos3`, `av41092` and `syn12000a`.

Finally, a summary of the results obtained on the Xeon processor is showed in Table 4. We have also included the performance provided by the standard reordering techniques. The most remarkable improvements are obtained when using windows with $w = 1$ and windows of variable size. Note how the average improvement percentage increases as the number of running threads grows. This way, reorderings performed using our technique exploit cache hierarchy even when HT is enabled, taking profit of the shared caches within each core. This behavior is not observed when standard techniques are considered. Note that, in order to compare with METIS reorderings, if we just consider symmetric pattern matrices, results would hardly vary.

5.4 Reordering Technique Overhead

Next, we analyze the time required by our proposal in order to perform the matrices reordering. This time depends directly on the number of nodes of the weighted distance graph (see Section 4.2). As we have commented, when considering fixed size windows the number of nodes in the graph is $\lceil N/w \rceil$. Nevertheless, using windows of variable size, we do not know a priori the size of the graph. It depends, at the same time, on the sparse matrix pattern and on the selected criterion to group the consecutive rows and columns of the matrix. Figure 6(a) shows, for all the studied matrices, the graph size when using windows of variable size normalized with respect to the graph when $w = 1$ (that is, when the number of nodes is equal to N). These results show an average reduction of about 65%, equivalent to use a fixed size $w \simeq 3$.

Finally, in order to evaluate objectively our proposal, a comparison with METIS reordering time is provided in Figure 6(b). Nowadays, METIS is considered as a standard in terms of graph partitioning and reordering. We use the Core2Duo processor as evaluation platform. Note that reordering times correspond with serial executions. Results indicate that our reordering technique using windows with $w \geq 8$ outperforms METIS. However, reorderings with windows of variable size and small size ($w < 8$) require more time than METIS. In particular, reorderings performed using windows of variable size introduce about two times the overhead of METIS, ranging from 0.6 sec (matrix `Na5`) to 5.8 sec (matrix `mixtank_new`). In other words, for these two cases, reordering times are equivalent to perform 20 and 23 sparse matrix-vector products when two threads are considered. The worst case corresponds with `garon2`, which requires 67 products. Consequently, 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 require thousands of sparse matrix-vector multiplications.

6 Conclusions and Future Work

In this paper a technique for increasing locality of sparse matrix codes on multicore platforms is presented. The technique consists on reorganizing the data guided by a locality model which introduces the concept of windows of locality. As a case of study, the product of a sparse matrix by a set of dense vectors was considered. The evaluation of the reordering technique has been performed on two different leading multicore platforms: Intel Core2Duo and Intel Xeon.

Several remarks can be made with respect to the windows of locality. Reorderings performed using small fixed size windows obtain the best results in terms of performance. In particular, an average execution time reduction of about 30% with respect to original matrices is observed when $w = 1$. However, a higher computational time is required in order to perform the reordering. For bigger sizes,

the overhead introduced by the reordering technique is reduced (improving the METIS reordering time), at the expense of obtaining lower performance. In addition, these reorderings show a strong dependence with the matrix sparsity pattern. According to this, we conclude that windows of locality of variable size are the best solution considering both performance and overhead.

On the other hand, a comparison study between our proposal and standard reordering techniques was made. Results on both considered multicore platforms show that our reordering technique always outperforms standard algorithms. Moreover, unlike standard techniques, the reordering technique is effective for matrices with any structure.

Future work will include the proposal and evaluation of new criteria in order to create windows of locality of variable size. Moreover, locality optimization technique will be applied to other important numerical kernels on different CMP systems.

Acknowledgements

This work has been partially funded by project TIN2007-63092 of Spanish Ministry of Education and project CCG07-UC3M/TIC-3277 of Madrid State Governement.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of the 11th Int. Symposium on High-Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

[4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proc, of the 19th ACM Symposium on Parallel algorithms and architectures (SPAA)*, pages 105–115, 2007.

[5] T. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997. http://www.cise.ufl.edu/research/sparse/matrices.

[6] R. E. Grant and A. Afsahi. A comprehensive analysis of OpenMP applications on dual-core intel Xeon SMPs. In *Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[7] D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Modeling data locality for the sparse matrix-vector product using distance measures. *Parallel Computing*, 27:897–912, 2001.

[8] S. Hily and A. Seznec. Standard memory hierarchy does not fit Simultaneous Multithreading. In *Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4)*, 1998.

[9] E. J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[10] G. Karypis and V. Kumar. *METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices*, 1997.

[11] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal Q1*, 2002.

[13] R. Noronha and D. K. Panda. Improving scalability of OpenMP applications on multi-core systems using large page support. In *Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[14] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

[15] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Improving the locality of the sparse matrix–vector product on shared memory multiprocessors. In *Euromicro Conf. on Parallel, Distributed and Network-based Processing (PDP)*, pages 66–71, 2004.

[16] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing*, 31(8–9):858–876, 2005.

[17] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. of Supercomputing*, 1999.

[18] Y. Saad. *Iterative methods for sparse linear systems*. SIAM (Society for Industrial and Applied Mathematics), 2003.

[19] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proc. of the 11th Int. Symposium on High-Performance Computer Architecture (HPCA)*, pages 248–252, 2005.

[20] S. Toledo. Improving memory–system performance of sparse matrix–vector multiplication. In *Proc. of the 8th SIAM Conf. on parallel processing for scientific computing*, Mar. 1997.

[21] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proc. of the 12th Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[22] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22th Int. Symposium on Computer Architecture*, pages 392–403, 1995.

[23] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiply on emerging multicore platforms. In *Proc. of Supercomputing (SC)*, 2007.

[24] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proc. of the 16th Int. Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 339–352, 2007.