A New Technique to Reduce False Sharing in Parallel Irregular Codes Based on Distance Functions*

Juan C. Pichel Dora B. Heras José C. Cabaleiro Francisco F. Rivera
Dept. Electrónica e Computación
Universidade de Santiago de Compostela, Galicia, SPAIN
jcpichel,dora,caba,fran@dec.usc.es

Abstract

*In this paper a technique to deal with the problem of poor locality and false sharing in irregular codes on shared memory multiprocessors (SMPs) is proposed. This technique is based on the locality model for irregular codes previously developed and extensively proven by the authors on monoprocessors and multiprocessors. In the model, locality is established in run-time considering parameters that describe the structure of the sparse matrix which characterizes the irregular accesses. As an example of irregular code with false sharing a particular implementation of the sparse matrix-vector product (*SpM×V*) was selected. The problem of increasing locality and decreasing false sharing for a irregular problem is formulated as a graph. An adequate distribution of the graph among processors followed by a reordering of the nodes inside each processor produces the solution. The results show important improvements in the behavior of the irregular accesses: reductions in execution time and an improved program scalability.*

1. Introduction

The hierarchical arrangement of memory in current computer architectures tries to exploit the locality properties in the execution of any code. The analysis and improvement of these properties are fundamental issues for increasing the performance in its execution. This fact is specially important for the case of irregular codes [14] because the accesses tend to present low spatial and temporal locality.

A large number of algorithms for evaluating and optimizing data locality can be found in the literature. In the case of regular codes, most approaches for increasing locality are based on decreasing conflict cache misses by using *blocking*, *strip–mining* or other code restructuring techniques [17]. There are a variety of static models for selecting appropriate memory hierarchy transformations and parameters such as tile sizes for regular codes [17, 8, 3].

In the case of irregular codes, the techniques for increasing the locality can be mainly divided into two categories: code restructuring techniques and data reordering techniques. Those based on restructuring the code, like blocking, have been applied with success, for example, to the product of a sparse matrix by a dense matrix [10].

In previous works we developed a model to characterize the locality and a procedure for increasing it using data reordering techniques. It was based on reorganizing the data instead of changing or restructuring the code [5]. The goal is to increase the grouping of elements in the pattern of the sparse matrix that characterizes the irregular accesses. As we demonstrated for monoprocessors and shared memory multiprocessors (SMPs) [11], our reordering is effective for matrices with any structure and for different sparse algebra kernels. Similar approaches, performing a reordering in certain data structures, have been successfully applied by other authors [15, 12] although considering only standard reordering techniques (Cuthill-McKee, nested dissection type orderings, etc.) and monoprocessors as experimental platforms.

However, there is another critical issue to be considered in the case of SMPs, namely, *false sharing*. False sharing occurs when multiple processors access (for both read and write) different words in the same cache block. In a write-invalidate coherency protocol, the overhead of false sharing implies extra invalidations when a processor updates data, and extra cache misses when other processors read different data that reside in the invalidated cache block [6]. This fact is specially time-consuming because remote accesses as well as coherence and consistence mechanisms are usually very costly [16]. The reduction in false sharing misses produces mainly two effects on the performance of the application: reductions in the execution time and improved program scalability.

In this paper we apply a procedure for increasing locality and reducing false sharing for irregular codes based on the

* This work was supported by the Spanish Ministry of Education and Science (MEC) under projects TIC2001-3694-C02-01 and TIN2004-07797-C02-01.

locality model developed previously [5]. The kernel that we have selected as case of study is an implementation of the sparse matrix-vector product (*SpM×V*). In this code false sharing is one of the main reasons for low performances. Our proposal can be generalized to other irregular codes in a straightforward way.

2. Characterization of the sparse matrix-vector product

In this section the locality model [5] is summarized. Moreover, the application of the model is analyzed for a particular implementation of the sparse matrix-vector product in which both locality and false sharing are important.

2.1. The locality model

From now on we will assume that when a high spatial or temporal locality in the accesses generated by the execution of a code is exhibited, there will be reuse of data in a particular level of the memory hierarchy under study [9]. The locality model is based on the evaluation of the data locality for the sparse algebra code considered. The model is general enough to be applied to any sparse algebra 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 if 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}). Considering accesses to the sparse matrix by columns, the number of entry matches between any pair of columns is defined as the number of nonzero elements in the same row of both columns. 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 column of the matrix pattern of the size of a cache line where there is at least one entry.

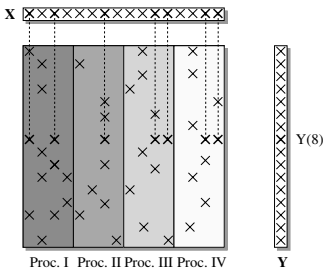
Based on these two parameters we have defined a magnitude called *distance between columns x and y*, denoted as $d_i(x, y)$. It is used to measure the locality displayed by the irregular accesses performed by the irregular code on these two columns when they are consecutively accessed. From different distance definitions proposed in [5], In this work we consider the following:

$$\begin{aligned} d_1(x,y) &= max_{\text{elems}} - a_{\text{elems}}(x,y) \\ d_2(x,y) &= n_{\text{blocks}}(x) + n_{\text{blocks}}(y) - 2{*}a_{\text{blocks}}(x,y) \\ d_3(x,y) &= n_{\text{elems}}(x) + n_{\text{elems}}(y) - 2{*}a_{\text{elems}}(x,y) \end{aligned}$$

where max_{elems} is the maximum number of entries in any column of the sparse matrix, $n_{\text{elems}}(x)$ is the number of elements in column *x*, and $n_{\text{blocks}}(x)$ is the number of groups of elements in column *x*. These definitions can be directly extended to rows. It can be shown that these distances define metrics.

```
DO i=1, N
    DO j=PTR(i), PTR(i+1)-1
        Y(Index(j)) += DA(j)*X(i)
    END DO
END DO
```

(a) Sequential algorithm



(b) Data accesses for the parallel execution

Figure 1. Algorithm for the sparse matrix-vector product by columns.

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

$$D_j = \sum_{i=0}^{N-2} d_j(i, i+1), \quad j = 1, 2, 3. \tag{1}$$

This model is suitable for any sparse matrix without limitations in the type of pattern that they present. 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. This issue is studied in the next section.

2.2. Locality and false sharing in the *SpM×V*

Let consider, for the *SpM×V* operation, that the matrix is stored using the standard Compressed-Column-Storage format (CCS) for unstructured sparse matrices [2]. DA, INDEX and PTR are the vectors (data, row indices and column pointer) that characterize this format. Using this format, and considering that X is the vector by which the product is performed and Y the result vector, the sequential product by columns could be implemented as displayed in Figure 1(a) being *N* the number of columns of the matrix. In this code, the sparse matrix is accessed in column major order. A closer grouping of elements in columns will lead to accesses to nearer elements of vector Y, improving the spatial locality in the accesses to that vector. A closer grouping of nonzero elements between two or more consecutive

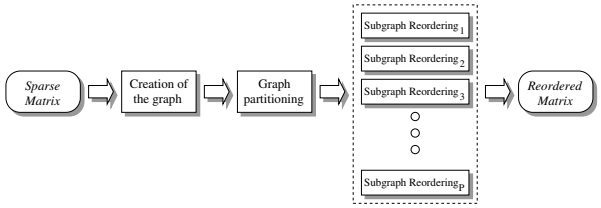


Figure 2. Locality improvement technique.

columns of the matrix will produce an increase in the temporal locality achieved in the accesses to Y. So, in general, the closer grouping of entries in the matrix pattern, the greater the locality in the accesses.

For shared memory multiprocessors the same code displayed in Figure 1(a) can be used. Only some directives to establish the group of consecutive columns that must be multiplied in each processor have to be added. However, some important issues have to be taken into account. In the parallelized code, false sharing is produced when a processor updates an element of Y in a block that has been previously written by other processor. As we have mentioned in Section 1, in a write-invalidate coherency protocol, the overhead of false sharing implies cache invalidations, and therefore, an increase in the number of cache misses when the invalidated data have to be accessed again.

A simple example considering four processors is shown in Figure 1(b). The sparse matrix is distributed by columns among four processors in such a way that each processor operates on different consecutive columns of the matrix. In order to compute each element of Y, it is necessary to perform the product of a row of the matrix by vector X. Each row is distributed among the processors, so the partial products must be accumulated to obtain the result. If, for example, one processor needs to update element Y(I), it has to invalidate the copies of the block in the local caches of the rest of processors in which Y(I) is stored. Therefore, when a processor reads some data in that block, a new cache miss is produced.

3. Data reordering technique

In the previous section we have outlined the relevance of increasing the locality in the execution of irregular codes. Furthermore, in many irregular codes, for achieving good performance false sharing has to be avoided. To deal with this problem we propose a heuristic technique that modifies the pattern of the sparse matrix according to the locality model described in the above section. We propose a technique that consists of three stages (shown in Figure 2):

Stage I *Defining a graph of the problem*: The problem is described using a weighted graph where each node represents a row or a column of the input sparse matrix depending on if the matrix is accessed by rows or by columns. Each edge of the graph has an associated weight that reflects the distance between pairs of columns (nodes) according to the description of locality given in the previous section. In theory, given a square sparse matrix irregularly accessed by columns, the model will compute a dense square matrix of distance values. This is computationally unacceptable because of the high memory requirements. But, given that in practice there are pairs of rows for which a_{elems} and a_{blocks} are zero, the distance matrix can be stored as a sparse matrix ignoring these values. Besides, the distance matrix is symmetric and therefore it can be stored as a triangular matrix so reducing the memory requirements.

Stage II *Graph partitioning*: The objective of this stage is two-fold. On one hand, to equally partition the graph among processors so that a good load balance, in terms of the number of nodes assigned to each processor, is achieved. On the other, to avoid false sharing. Reducing false sharing is related to inter-processors locality. In particular, the number of false sharing situations is smaller as this locality decreases. According to our locality model, this decrease is equivalent to maximize the distance between the subgraphs assigned to each pair of processors. This way, the number of invalidations is reduced, and therefore, cache misses due to false sharing are minimized. For distributing the nodes among the processors according to the objectives detailed above, we use the *pmetis* program from the METIS software package [7].

Stage III *Subgraph reordering*: After graph partitioning, P subgraphs were obtained, one per processor. In this stage the objective is to increase the intra-processor locality. In order to deal with this problem we use an analogy with the Traveling Salesman Problem (TSP). Solving this problem is equivalent to finding a path of minimum length that goes through all the nodes of each subgraph. This path is associated to a permutation vector that gives the appropriate order of the nodes, and therefore, a reordered matrix. Given that we have measures (distance values) to validate the adequateness of an ordering, we have focused on heuristic solutions. After a comparative study of different techniques, the *Chained Lin–Kernighan* heuristic proposed by Applegate et al. [1] was used.

In order to obtain a better understanding of the reordering technique consider the example of Figure 3. The three different stages of the technique are shown. In the first stage the graph is created from the input matrix. In this example we use d_1 as distance function. On the right of the graph, the distances between the nodes (columns of the matrix) are detailed. Note that, only the distances with a_{elems} and a_{blocks}

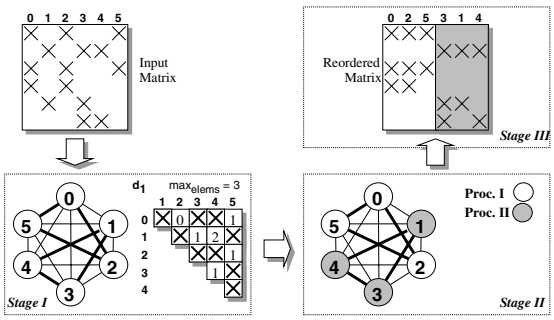
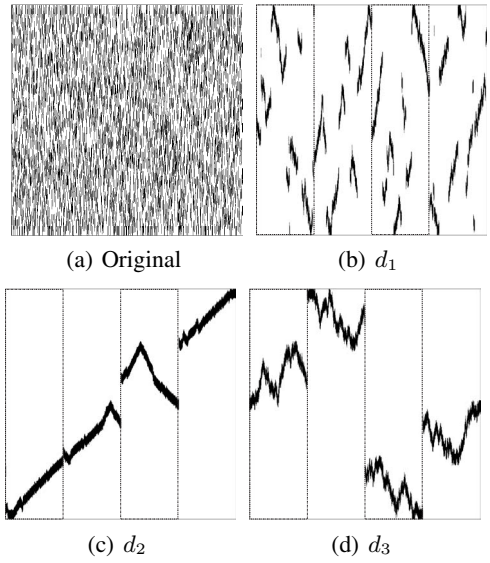


Figure 3. Application of the reordering technique.



(a) Original (b) d_1

(c) d_2 (d) d_3

Figure 4. Reordered patterns of matrix *2* using different distance functions.

not equal to zero are stored. In this case, max_{elems} is 3. In the second stage, the distribution of nodes among the processors is performed. Moreover, we try to maximize the distance between each pair of graphs assigned to each pair of processors. In this example, just two processors are considered. Nodes 0, 2 and 5 are assigned to processor I, and nodes 1, 3 and 4 to processor II. In this way, the distance between the two subgraphs is maximum. Finally, in the third stage, after distribution of the nodes two subgraphs are obtained. We search for a path of minimum length that goes through all the nodes of each subgraph. In this case, the new order of nodes for processor I is 0, 2 and 5. Whereas 3, 1 and 4 is the order for processor II. In summary, a new reordered matrix is obtained.

Therefore, applying the locality improvement technique implies changes in the pattern of the input sparse matrix. Figure 4 shows the original pattern of matrix *2* of our test set (see Section 4.1), and the patterns after reordering it for 4 processors using the three distance functions detailed in Section 2. The patterns also show the columns assigned to each processor. Note that for the reordered matrices there is a higher clustering of entries than for the original one. Besides, the group of elements of Y over which each processor will operate are disjoint, and therefore, false sharing decreases.

4. Performance evaluation

This section begins with a description of the experimental conditions. The overheads incurred in our approach are analyzed in Section 4.2. Later, in Section 4.3, the experimental results obtained after applying our technique as well as their influence on the performance of the code are shown.

4.1. Experimental conditions

As a test set we have selected seven square sparse matrices from different real problems that represent a variety of non-zero patterns. Table 1 summarizes some features of the matrices. N is the number of rows or columns, and N_Z is the number of entries. These matrices are from the University of Florida Sparse Matrix Collection (UFL) [4] (except matrices *1* and *2*). Matrix *1* was generated from a N-body simulation [13], and matrix *2* is a synthetic matrix with a random uniform distribution of entries over the pattern.

Our model has been validated on a SGI Origin 2000 system with MIPS R10k that operates at 250 MHz. The R10k is a four-way superscalar RISC CPU. The R10k uses a two-level cache hierarchy: a 32 KBytes L1 data and instruction caches, and a unified L2 cache of 4 MB. For L1 the line size is 32 bytes, and 128 bytes for L2. Both caches are two-way set associative with LRU replacement policy. The code was written in Fortran with OpenMP directives.

4.2. Overhead of the reordering technique

Given that a preprocessing of the sparse matrix (the application of the data reordering technique) must be performed before executing the parallel code, it is necessary to consider its cost. It includes the three stages described in

	Matrix	N	N_Z	Application Area
1	*nc5*	19652	1499816	N-Body simulation
2	*syn12000a*	12000	1436806	Synthetic matrix
3	*nmos3*	18588	237130	Semiconductor Simulation
4	*igbt3*	10938	130500	Semiconductor Simulation
5	*garon2*	13535	373235	FEM Navier-Stokes
6	*poisson3Da*	13514	352762	FEM
7	*sme3da*	12504	874887	FEM

Table 1. Matrix benchmark suite.

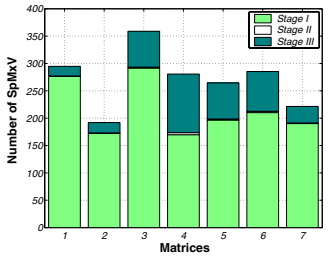
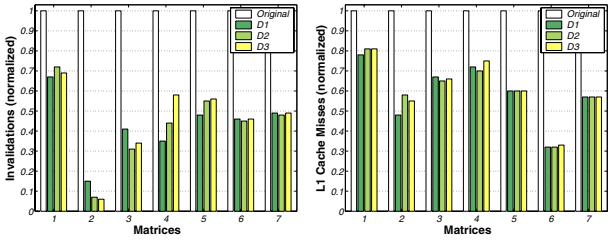
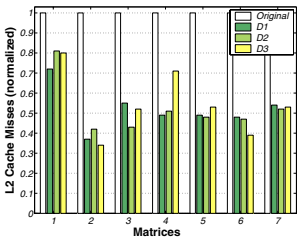


Figure 5. Overhead of the reordering technique.



(a) Invalidations (b) L1 Misses



(c) L2 Misses

Figure 6. Cache behavior using 8 processors.

Section 3. Figure 5 displays the number of sequential products that must be performed to compensate the time consumed by the preprocessing stage when it is also sequentially computed. Note that the most costly stage for all the matrices is stage I. And this can be easily parallelized, and consequently, reducing the overhead.

Therefore, the overhead of the reordering technique can be amortized by the repeated execution of the parallel product as in the case of iterative methods [2], which might lead to computational savings in subsequent executions. For example in a simulation of a semiconductor device. The overhead of the reordering technique is not included in the results along this section.

4.3. Experimental results

We have used the R10k hardware counters to measure the L1 an L2 cache misses as well as the number of L2 cache invalidations. Figure 6 shows, as an example, the results of

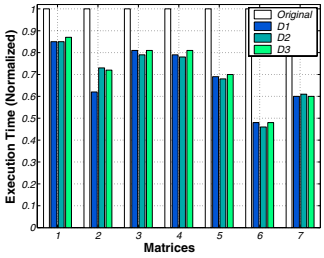
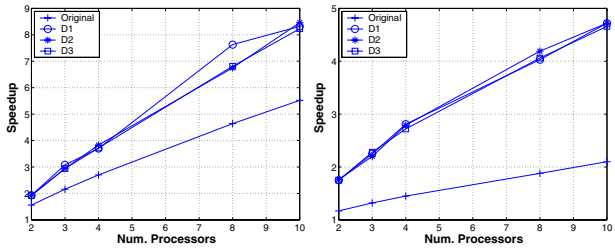


Figure 7. Execution time using 8 processors.

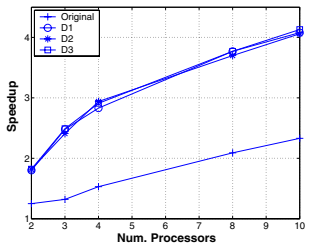
applying our technique (normalized with respect to the case of the original matrix) for the matrices of the test set on 8 processors. As we have previously commented, the overhead due to false sharing implies invalidations when a processor updates data, and cache misses when other processor reads data included in the invalidated cache block. Therefore, increasing performance in the execution is equivalent to reduce the number of invalidations, which indeed affect the number of cache misses produced. The behavior of the cache, in terms of the number of invalidations, is shown in Figure 6(a). An important reduction in percentage is observed. This reduction is specially noticeable for matrix *2*, obtaining reductions up to 95% with respect to the original matrix. The behavior of the cache in terms of the number of L1 and L2 misses is shown in Figures 6(b) and 6(c) respectively. The reductions in both cases are very important. There are few differences in the cache behavior using the different distance functions and therefore, none of them can be considered the best for a relevant number of cases.

The reduction in false sharing has two important effects on run-time performance: a reduction in the execution time and an improved program scalability. Figure 7 details the execution time of the code (normalized with respect to the case of the original matrix) using the reordered matrices for 8 processors. Note that important reductions in the execution time, that go from 15% to 55%, have been reached.

In addition, note that applying our technique, the scalability of the code is increased as shown in Figure 8. In these figures the speedup for three matrices of the test set is detailed using up to 10 processors. The speedup is measured using the parallel code and the reordered matrices with respect to the sequential code using the original matrix. Note that a better behavior is observed as the number of processors increases for all the cases compared to those corresponding to the original matrices. For instance, speedup of matrix *2* using 10 processors is less than 6, while when the reordered matrices are considered, the speedup is over 8 (Figure 8(a)). Another example is the evolution of the speedup for matrix *6*, shown in Figure 8(b). In this case, with the original matrix, a low speedup using 10 processors is obtained, just around 2. Whereas with the reordered matrices the speedup is around 5. The behavior in terms of



(a) Matrix *2* (b) Matrix *6*



(c) Matrix *7*

Figure 8. Speedup obtained for different matrices.

speedup for matrices *6* and *7* (Figure 8(c)), is similar.

5. Conclusions

In this paper we propose a technique to deal with the problem of the locality and false sharing for irregular codes on SMPs. Our technique is based on the locality model that we have previously developed. In the model, locality is established in run-time considering parameters that describe the structure of the sparse matrix that characterizes the irregular accesses. The problem is solved as a graph partitioning among processors followed by a reordering of the subgraph assigned to each processor.

This technique produces important improvements. We have observed reductions up to 95% in the number of invalidations and also important decreases in terms of cache misses. These reductions benefit the execution time and the program scalability. The average decrease in the total execution time using the technique is about 35%. This reduction grows as the number of processors increases. Therefore, the scalability of the code also improves significatively. For instance, using 10 processors, the average speedup of the code with the original matrices is around 4, while with the reordered matrices is more than 7.

References

[1] D. Applegate, R. Bixby, V. Chvátval, and W. Cook. Finding tours in the TSP. *Draft available from* http://www.math.princeton.edu/tsp, 1998.

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

[3] S. Chaterjee, E. Parker, J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 286–297, June 2001.

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

[5] 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.

[6] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 179–188, 1995.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998.

[8] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, June 1996.

[9] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[10] J. J. Navarro, E. García, J. L. Larriba-Pey, and T. Juan. Block algorithms for sparse matrix computations on high performance workstations. In *Proc. IEEE Int'l. Conf. on Supercomputing (ICS'96)*, pages 301–309, 1996.

[11] 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, PDP2004*, pages 66–71. IEEE Computer, 2004.

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

[13] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, 1995.

[14] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *IEEE Int'l Conf. on Supercomputing (ICS'92)*, pages 578–587, 1992.

[15] 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.

[16] E. Torrie, M. Martonosi, C. Tseng, and M. W. Hall. Characterizing the memory behavior of compiler–parallelized applications. *IEEE Transactions on Parallel and Distributed Systems*, 7(6), Dec. 1996.

[17] M. E. Wolf and M. S. Lam. A data locality optimization algorithm. In *Proc. SIGPLAN'91 Conf. on Programming Language Design and Implementation*, June 1991.