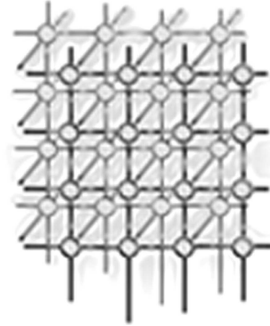


---

# Using sampled information, is it enough for the SpMV locality optimization?



Juan C. Pichel\*, Juan A. Lorenzo,  
Francisco F. Rivera, Dora B. Heras and  
Tomás F. Pena

*Centro de Investigación en Tecnologías da Información (CITIUS)  
Universidade de Santiago de Compostela, Spain*

---

## SUMMARY

One of the main factors that affect the performance of the sparse matrix-vector product (SpMV) is the low data reuse caused by the irregular and indirect memory access patterns. Different strategies to deal with this problem have been proposed such as data reordering techniques. The computational cost of these techniques is typically high since they consider all the nonzeros of the sparse matrix in order to find an appropriate permutation of rows and columns that improves the SpMV performance.

In this paper we analyze the possibility of increasing the locality of the SpMV using incomplete information in the reordering process. This partial information comes as a consequence of considering only a subset of the nonzero elements of the matrix. These nonzeros are obtained from the original matrix through a sampling process. In particular, two different sampling methods have been considered: a random sampling and an event-based sampling (EBS) using hardware counters.

We have detected that a small number of samples is enough to obtain quality reorderings. As a consequence, using sampling-based reorderings leads to noticeable performance improvements with respect to the non-reordered matrices, reaching speedup values up to  $2.1\times$ . In addition, an important reduction in the computational time required by the reordering technique has been observed.

KEY WORDS: sparse matrix; locality; hardware counters; sampling; performance

---

\*Correspondence to: Centro de Investigación en Tecnologías da Información (CITIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain.

\*E-mail: juancarlos.pichel@usc.es

Contract/grant sponsor: This work was supported by Hewlett-Packard Spain S.L., Ministry of Education (Spain) and Xunta de Galicia; contract/grant number: 2008/CE377 contract, TIN2007-67537-C03-01 and 09TIC002CT respectively.



---

## 1. Introduction

Sparse matrix-vector product (SpMV) is one of the most important computational kernels in scientific and engineering applications. It is notorious for sustaining low fractions of peak performance (typically, about 10%) on modern processors. Some of the factors affecting the SpMV performance are the memory bandwidth limitation and the low data reuse caused by the irregular and indirect memory access patterns. Note that the sparse matrix pattern that characterizes the irregular accesses is only known at run-time. To this end, different strategies focusing on some of the SpMV performance problems have been proposed. This is the case of reordering techniques. In these techniques the whole sparse matrix pattern is evaluated to find an appropriate permutation of rows and columns that improves the SpMV performance. Considering all the nonzeros of the matrix normally implies high computational costs that must be amortized when the sparse operation is repeatedly performed as, for instance, in iterative methods for solving systems of linear equations.

In this work we analyze the possibility of increasing the locality of the SpMV when only a subset of the memory accesses performed is available in the optimization process. This is equivalent to considering only a subset of the nonzero elements of the matrix. To the best of our knowledge, researchers have never dealt with the locality optimization of the SpMV using reordering techniques and incomplete information. Note that the reduction in the amount of information managed by the optimization technique must have an impact on its computational cost.

As locality optimization strategy we have selected a data reordering technique previously developed by the authors [27]. It consists of reorganizing the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. The goal of this technique is to increase the grouping of nonzero elements in the sparse matrix pattern that characterizes the irregular accesses and, as a consequence, improving the locality in the execution of the SpMV code. According to this, the technique must find the appropriate order of rows and columns of the original matrix for improving the locality using only the information provided by a subset of its nonzero elements. These nonzeros are obtained from the original matrix through a sampling process. In particular, two different sampling methods have been considered in this work: a random sampling and an event-based sampling (EBS) using hardware counters.

In the first case the nonzero elements of the sampled matrices used as input of the reordering technique are selected randomly from the original matrix in such a way that each nonzero has equal chances of being sampled. Despite the fluctuations in the performance measurements caused by the randomly sampling method, we have detected that a small number of samples is enough for the reordering technique to generate quality reorderings. Tests have been performed on two systems consisting of different multicore processors: Itanium2 and Xeon (with Nehalem microarchitecture).

An event-based sampling process using hardware counters is then considered. A new methodology to obtain the position of the nonzero elements of a sparse matrix from the sampled information provided by the hardware counters was introduced. The performance evaluation shows that reorderings generated using the information provided by these sampled matrices obtain very similar results with respect to the ones generated by the original technique. In



addition, we have observed that there are very small differences in the SpMV performance achieved by reordered matrices generated using the information provided by different samplings of the same original matrix. A comparison of the SpMV performance with the original (non-reordered) matrices is also provided. Speedups up to  $2.1\times$  were observed when using reordered matrices. On the other hand, a consequence of using sampled information to perform the reordering is an important reduction in the overhead introduced by the reordering technique. In particular, reductions higher than 90% were detected in comparison with the computational cost of the original technique.

The remainder of this paper is structured as follows: Section 2 discusses previous work on the SpMV optimization. Section 3 presents the main characteristics of the SpMV kernel and the hardware platform used in the tests, together with a summary of the locality optimization technique. Section 4 shows the results of the study when considering the randomly sampled matrices to perform the reorderings. Section 5 analyzes the behavior of the reordering technique when using the hardware counters for sampling the memory accesses. The paper finishes with the main conclusions extracted from the work.

## 2. Related Work

Many works dealing with the optimization of the sparse matrix-vector product can be found in literature. Techniques for increasing the locality of SpMV can be mainly divided into two groups: data reordering and code restructuring techniques.

Standard data reordering techniques are considered classical methods for increasing the locality in the execution of the SpMV code, although this is not their main objective. The most used techniques are the bandwidth reduction algorithms, which derive from the Cuthill-McKee algorithm [9]. In a recent work [27], authors evaluate *minimum degree*-based heuristics such as the *approximate minimum degree* algorithm [1] on multicore processors. Oliner *et al.* [24] show the benefits that are offered by the application of some of these reordering algorithms to sparse codes when executed on different multiprocessor architectures. Note that, unlike standard reordering algorithms, the main objective of the data reordering technique considered in this work is to increase the data reuse and, as a consequence, the data locality. Coutinho *et al.* [8] perform a comparison of different data reordering algorithms for the SpMV in edge-based unstructured grid computations. However, they only focus on serial executions.

Techniques based on restructuring the code, like *blocking* or *tiling*, have been successfully applied to different irregular codes such as the product of a sparse matrix by a dense matrix [14, 23] and stationary iterative methods [29]. Im *et al.* [16] propose register and cache blocking as optimization techniques for the SpMV. In [5], a performance model for the blocked SpMV is presented, which allows to pick in nearly all cases the actual optimal blocksize. In these last two works the authors use a randomly sampled matrix at runtime to detect the best blocking size. Vuduc *et al.* [32] extend the notion of blocking in order to exploit variable block shapes by decomposing the original matrix to a proper sum of submatrices storing each submatrix in a variation of the blocked CSR format. In a recent work [19], a comparative study of different blocking storage techniques for sparse matrices on several multicore platforms is performed. One of the main drawbacks of these techniques is the



strong dependence with the sparsity pattern of the matrix. For example, register blocking only achieves good performance for matrices with small dense-blocks in the pattern. Unlike these solutions, our locality optimization technique is effective for matrices with any pattern [27]. Finally, Belgin *et al.* [3] introduce a representation for sparse matrices based on the observation that many matrices can be divided into blocks that share a small number of different patterns. The goal is to reduce the SpMV memory bandwidth requirements by reducing the index overhead.

Some authors have demonstrated that both groups of techniques are complementary. In particular, Toledo [31] evaluates different standard reordering techniques and combines them with blocking, showing that SpMV performance increases significantly depending on the size and sparseness of the considered matrix. Pinar and Heath [28] introduce a reordering technique that favors the creation of dense blocks on the pattern of the sparse matrix, and in this way the efficiency of the blocking technique proposed by Toledo is increased. Moreover, a comparison between their reordering technique and some standard reordering techniques is carried out. In another work [15], a combination of data reordering algorithms and register blocking has been applied to the SpMV on shared memory multiprocessors, finding little benefit. The locality optimization technique used in the present paper can also be applied to codes where data are stored using a blocked scheme. An example was published in [26] where a reordering of the sparse matrix in combination with blocking techniques was successfully applied to the SpMV. The technique was evaluated on different uniprocessors and on distributed memory multiprocessors.

Moreover, there are several papers that deal with the SpMV optimization problem using compression. In a recent work [20], the authors propose two different compression methods targeting index and numerical values. Williams *et al.* [33] apply an index reduction technique, in which 16-bits indices are used when it is possible. In the same work the authors propose several additional optimization techniques for the SpMV, which are evaluated on different multicore platforms. Authors examine among others: software pipelining, prefetching approaches, register and cache blocking, etc. Nevertheless, they do not consider data reordering techniques in order to increase locality.

Research regarding the use of hardware counters is mainly focused on the characterization and on the analysis of possible bottlenecks in the performance of the applications [13, 21]. However, some works use hardware counters for different optimizations such as improving cache utilization [4], reducing memory access stalls [7], selecting compiler optimization settings [6] and dynamic page migration [30]. To the best of our knowledge, researchers have never dealt with the locality optimization of the SpMV using only the information provided by the hardware counters.

### 3. Experimental Conditions

#### 3.1. Hardware Platforms

Table I summarizes the key features of the hardware platforms considered in this work. The first platform is based on Itanium2 Montvale processors. These processors comprise two cores



Table I. Specifications of the considered hardware platforms.

Company Processor	Intel	
	Itanium2 9140N (Montvale)	Xeon X5570 (Gainestown)
Clock (GHz)	1.6	2.93
Cores/Socket	2	4
Private L1 DCache		
Size (KB)	16	32
Latency (cycles)	1	4
Private L2 DCache		
Size (KB)	256	256
Latency (cycles)	5-7	10
L3		
Size	9 MB/core	8 MB (shared)
Latency (cycles)	14-21	35
Flops/cycle	4	4
GFlop/s	6.4	11.72
System		
# Sockets	4 (SMP)	1

and three cache levels per core. L1D (write-through) is a 4-way set-associative, 64-byte line-sized cache. L2D (write-back) is a 8-way set-associative, 128-byte line-sized cache. And finally, there is an unified 12-way L3 cache per core, with line size of 128 bytes and write-back policy. Cache latencies are 1, 5-7 and 14-21 cycles respectively. Note that floating-point operations bypass the L1 in this system. Each Itanium2 core can perform 4 FP operations per cycle. The peak performance per core is 6.4 GFlop/s.

In the platform under study, Itanium2 processors are arranged in a SMP-cell configuration. A cell has two buses at 8.5 GB/s (6.8 GB/s sustained), each connecting two sockets (that is, four cores) to a 64 GB memory module through a sx2000 chipset (Cell Controller). The Cell Controller maintains a cache-coherent memory system using a directory-based protocol. It yields a theoretical processor-cell controller peak bandwidth of 17 GB/s and a cell controller-memory peak bandwidth of 17.2 GB/s (four buses at 4.3 GB/s).

The second platform is based on Xeon 5500 series processors (Nehalem-EP), which have a native quad-core design. Each core has three cache levels. L1D and L2D are 8-way set-associative and 64-byte line-sized caches. L3 is a 16-way unified, inclusive and shared cache. Cache latencies are 4, 10 and 35 cycles respectively. All the caches use the write-back policy. Processors based on the Nehalem microarchitecture feature a dynamic overclocking mechanism (Intel Turbo Boost Technology [18]) that allows the processor to raise core frequencies as the thermal limit is not exceed. In particular, Turbo Boost provides a frequency-stepping mode that enables the processor frequency to be increased in increments of 133 MHz. Note that the



[t]

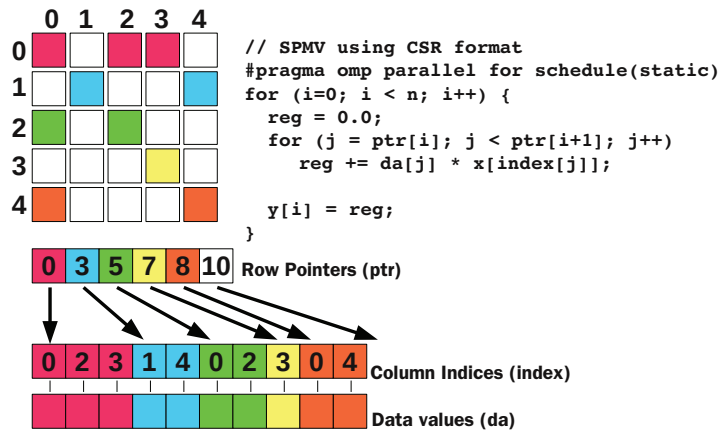


Figure 1. Compressed-Sparse-Row (CSR) format example, and a basic CRS-based sparse matrix-vector product (SpMV) implementation.

peak performance shown in Table I (11.72 GFlop/s) is calculated using the base frequency (2.93 GHz). Moreover, these processors have an on-chip memory controller, which supports three DDR3 memory channels. The peak memory bandwidth for the Nehalem processor is 32 GB/s.

### 3.2. Sparse Matrix-Vector Product (SpMV)

In this work the sparse matrix-vector product (SpMV) operation is considered. This kernel is notorious for sustaining low fractions of peak processor performance due to its indirect and irregular memory access patterns. Let us consider the operation  $y=A \times x$ , where  $x$  and  $y$  are dense vectors, and  $A$  is a  $n \times m$  sparse matrix. The most common data structure used to store a sparse matrix for SpMV computations is the Compressed-Sparse-Row format (CSR).  $da$ ,  $index$  and  $ptr$  are the three arrays (data, column indices and row pointer) that characterize this format. Figure 1 shows, in addition to an example of the CSR format for a  $5 \times 5$  sparse matrix, a basic implementation of SpMV for this format. This implementation enumerates the stored elements of  $A$  by streaming both  $index$  and  $da$  with unit-stride, and loads and stores each element of  $y$  only once. However,  $x$  is accessed indirectly, and unless we can inspect  $index$  at run-time, it is difficult or impossible to reuse the elements of  $x$  explicitly. Note that the locality properties of the accesses to  $x$  depend on the sparsity pattern of the considered matrix.



Table II. Matrix benchmark suite.

Matrix	# rows (n)	# nonzeros (nnz)	nnz/row
crystk03	24696	1751178	71
garon2	13535	390607	29
gyro_k	17361	1021159	59
mixtank_new	29957	1995041	67
msc10848	10848	1229778	113
nd3k	9000	3279690	364
nmos3	18588	386594	21
pct20stif	52329	2698463	52
sme3Da	12504	874887	70
tsyl201	20685	2454957	119

Codes in this work were written in C and compiled with the Intel's 10.0 Linux C compiler (icc). OpenMP directives were used to parallelize the irregular code of Figure 1. In particular, the partitioning scheme splits the matrix row-wise with a block distribution. All the results shown in the next sections were obtained using the compiler optimization flag `-O2`. Note that HyperThreading is available for both systems detailed above but it was disabled in the tests shown in this paper.

As matrix test set we have selected ten 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) [10]. Table II summarizes the features of the matrices.

### 3.3. Locality Optimization Technique

A data reordering technique has been used to check if it is possible to optimize the locality only considering a subset of the memory accesses performed by the SpMV code. We have introduced this technique in a previous work [27]. It consists of reorganizing the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. Unlike other existent locality models that predict in a precise way the data movement among the levels of the memory hierarchy, our model is able to characterize, sacrificing accuracy, the trend of this movement in general terms. In particular, locality is evaluated using a distance function that depends on the number of entry matches ( $a_{\text{elems}}$ ). 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. In this way, distance between rows  $i$  and  $j$  is defined as:

$$d(i, j) = n_{\text{elems}}(i) + n_{\text{elems}}(j) - 2*a_{\text{elems}}(i, j) \quad (1)$$

where  $n_{\text{elems}}(i)$  is the number of entries in row  $i$ . This function is used to measure the locality displayed by the accesses performed by the SpMV code on these two rows when they are consecutively accessed.

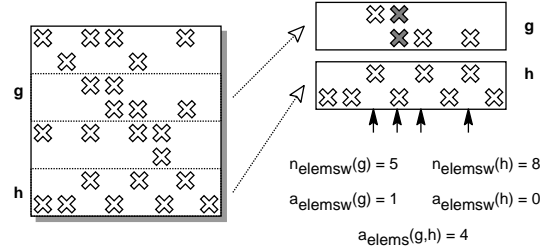


Figure 2. Example of the calculation of  $n_{\text{elemsw}}(g)$ ,  $a_{\text{elemsw}}(g)$  and  $a_{\text{elems}}(g, h)$ .

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

$$D = \sum_{i=0}^{n-2} d(i, i+1) \quad (2)$$

where  $n$  is the number of rows/columns of the sparse matrix. These definitions can be directly extended to columns. Note that these functions only provide results based on the locality evaluated on pairs of consecutive rows (or columns) of the sparse matrix. Nevertheless, 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* was 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  $d(i, j)$ , we can define the distance between windows of locality  $g$  and  $h$  as:

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

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  $w = 2$ .

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:

$$D_w = \sum_g d_w(g, g+1), \quad \forall g \mid 0 \leq g < \lceil n/w \rceil \quad (4)$$

These distances (Equations 3 and 4) are equivalent to distances measured over pairs of consecutive rows/columns of the matrix when the window size is  $w = 1$ .

The reordering technique 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 SpMV





code, we search for a permutation of windows of locality of the matrix that minimizes its total distance  $D_w$  (Equation 4). The problem of locality improvement is formulated as a classic NP-complete optimization problem, and it is solved as a graph problem using its analogy to the traveling salesman problem (TSP).

The problem is described using a 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  $a_{\text{elems}} = 0$ . Those values, according to the distance definitions, represent the worst cases regarding locality. So, without losing relevant information about locality, we can use an incomplete weighted graph where only values of  $a_{\text{elems}}$  different 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. 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 the *Chained Lin-Kernighan* heuristic proposed by Applegate et al. [2] was chosen. Note that, for practical purposes, if there is an isolated node without edges in the distance graph, the TSP heuristic will internally connect this node to the others through edges with a very high distance.

In order to select the window size ( $w$ ), two types of windows of locality are considered: fixed and variable [27]. For windows of fixed size 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 could be formed by consecutive rows (or columns) of the matrix which exhibit low locality according to our model. This is the reason why big fixed-size windows do not obtain, in general, good results.

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, whose 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 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. In order to decide if two consecutive rows  $i$  and  $j$  will be included within the same window a simple criterion must be fulfilled:  $d(i, j) < D/n$ , 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

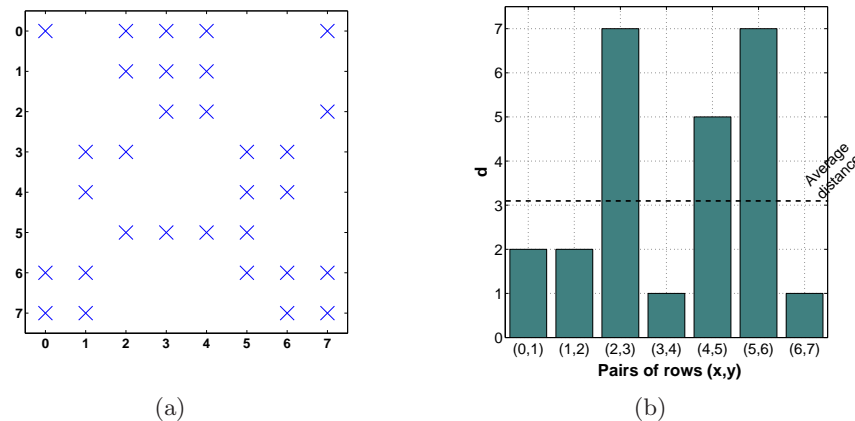


Figure 3. Example of the creation of windows of locality of variable size: (a) sparse matrix example and (b) distance histogram.

pattern is non-symmetric, the windows creation process must be applied considering rows and columns independently.

In our studies we conclude that windows of  $w = 1$  and  $w = variable$  are the best choices in terms of performance. For this reason we have focused on them in this work. However, big fixed-size windows can be a good solution due to the particularities of the sparse matrices in some applications such as those related to the simulation of semiconductor devices [25].

In the next sections the behavior of this reordering technique is analyzed when only a percentage of the nonzero elements of the matrix considered to estimate the locality. That is, when only a subset of the accesses to  $\mathbf{x}$  is available (see the SpMV code in Figure 1). In other words, the permutation vector is calculated using a *sampled matrix* generated from the original one. This vector is then applied to the original matrix in order to reorganize the data and, this way, to increase the locality in the execution of the SpMV.

#### 4. Performance Evaluation Using Randomly Sampled Matrices

In this section we have carried out a study in order to check if considering only a subset of the nonzero elements from the original matrices is enough for the locality optimization technique to obtain similar performance results with respect to the reorderings using complete information. With this purpose we have generated a set of sampled matrices from the original ones of our testbed. These matrices consist of a subset of the nonzeros of the original ones. In particular, sampled matrices contain 1%, 2%, 5%, 10%, 15% or 20% of the original nonzeros. These nonzeros are randomly selected using `random()`, which implies that each nonzero of the original matrix has equal chances of being sampled. Twenty sampled matrices were generated

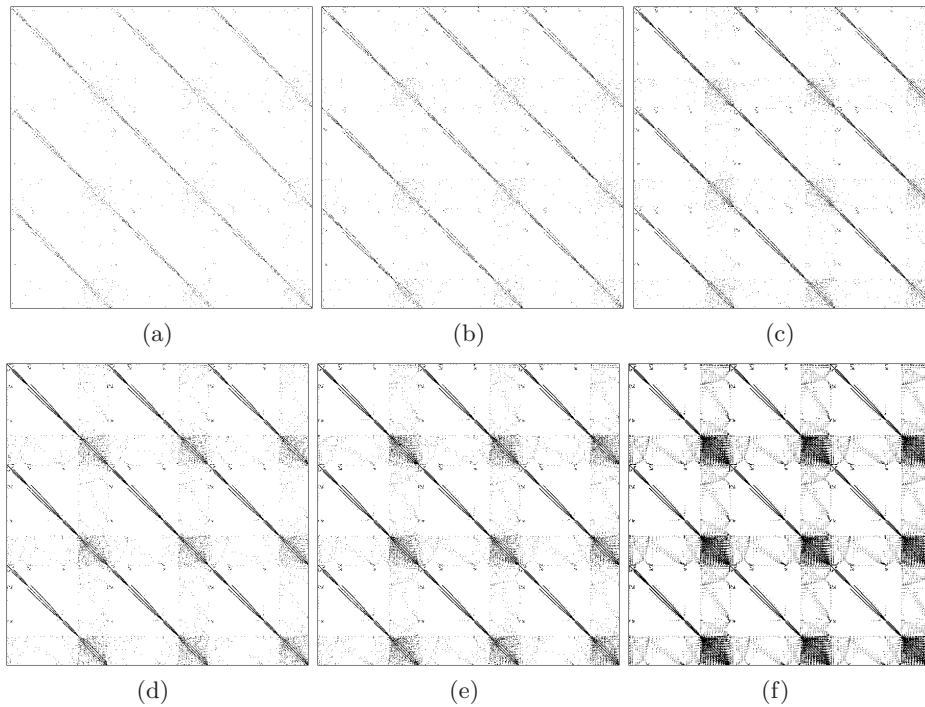


Figure 4. Examples of `nmos3` randomly sampled matrices. Matrices with 1% (a), 2% (b), 5% (c), 10% (d) and 20% (e) of the nonzeros with respect to the original matrix (f).

for each matrix and percentage, which means a total number of 1200 sampled matrices used in the tests. Figure 4 shows an example of the nonzeros pattern of some randomly sampled matrices generated from `nmos3` matrix.

Information provided by the sampled matrices has been used to generate a permutation vector that will be applied to the original matrix. This permutation vector is calculated by our reordering technique using the sampled matrices with windows of variable size and fixed size with  $w = 1$  (see Section 3.3). In this way, the reordering is performed to the original matrix considering only the information provided by a subset of its nonzeros.

In order to estimate the quality of the reorderings obtained using the sampled matrices, a comparison with respect to the original technique (that is, when the complete matrix is used to calculate the permutation vector) has been carried out. Figures 5, 6, 7 and 8 show the normalized SpMV performance in such a way that 1 is the result of the original technique. In this way, the normalized performance is calculated as the ratio between the performance of the considered reordering and the reorderings using complete information, both in GFlop/s. The figures show the average performance for each sampled matrices set and number of threads.

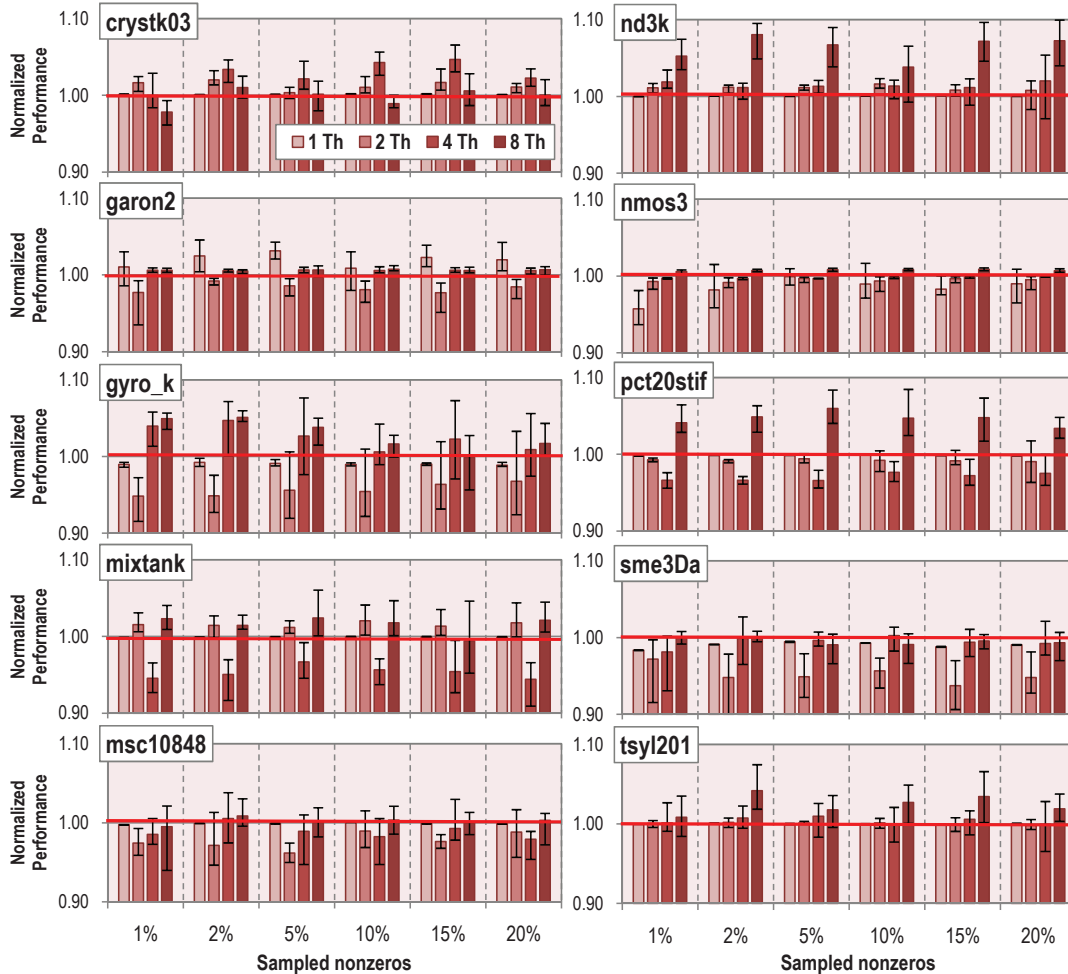


Figure 5. Normalized SpMV performance obtained by the reorderings generated using the locality optimization technique ( $w = 1$ ) and the information provided by the randomly sampled matrices on the Itanium2 platform.

They also include the maximum and the minimum performance values as error-bars. Fifty runs per matrix have been carried out to obtain these results. We must highlight that due to the random sampling process, the twenty sampled matrices of each subset can be very different. Given that these sampled matrices are used as input of the reordering technique, the output in each case may show a high variation, which will cause fluctuations in the observed performance within each subset.

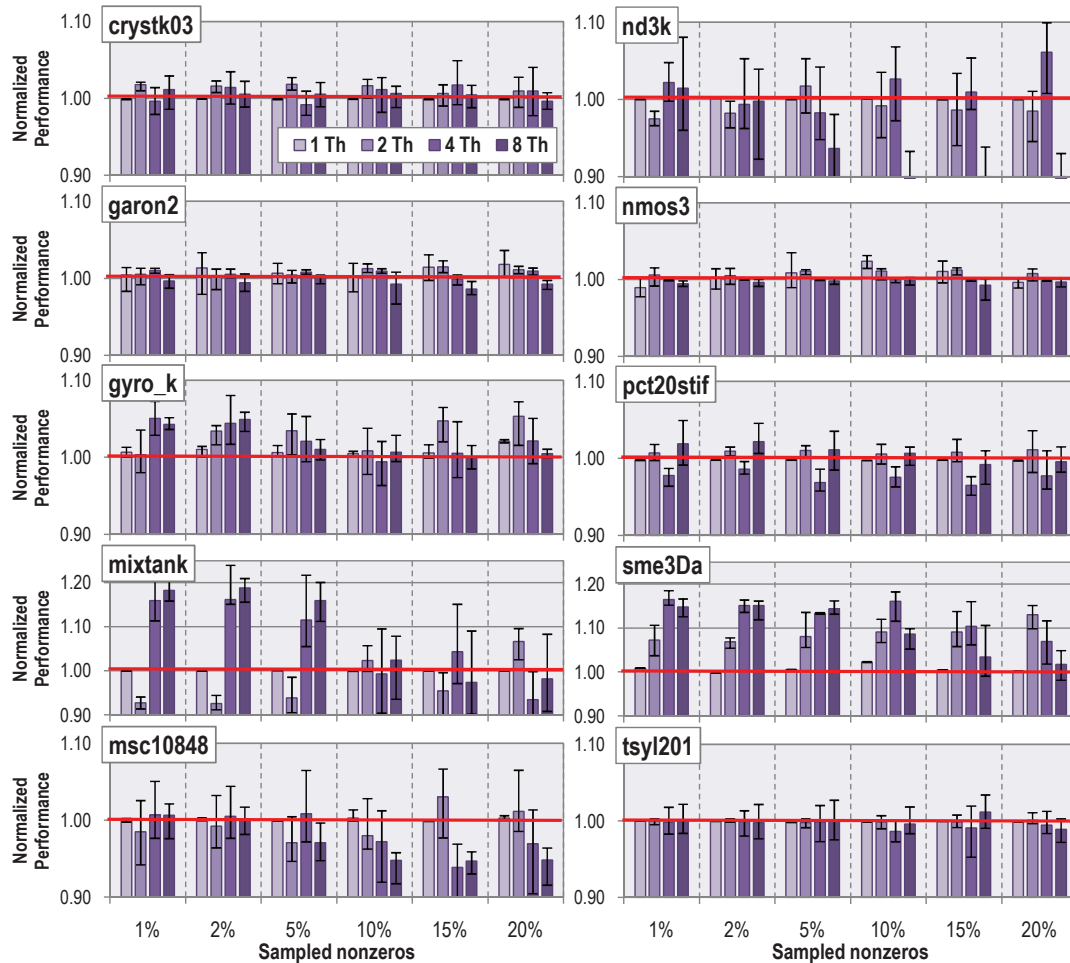


Figure 6. Normalized SpMV performance obtained by the reorderings generated using the locality optimization technique ( $w = \text{variable}$ ) and the information provided by the randomly sampled matrices on the Itanium2 platform.



First, we will focus on the results on the Itanium2 system when the reordering is performed using windows of fixed size,  $w = 1$  (Figure 5). At first sight we can conclude that reorderings using sampled and complete information show a similar behavior. We must highlight that this occurs in some cases even when only 1% of the nonzeros of the matrices are considered. It means that the locality model used by the reordering technique is able to characterize the accesses performed by the sparse matrix using only this information. In several cases the sampled nonzeros are not enough to compete with the original technique. This is the case of matrices `gyro_k` and `sme3Da` when using two threads, or `mixtank.new` when four threads are considered. However, the differences with respect to the reference is at the most about 5%. In the same way, there are some cases where reorderings using sampled information outperforms the original ones. For example, reorderings of `nd3k` and `pct20stif` when using eight threads.

The explanation of this behavior can be found in some details about the locality model and the reordering heuristic. The reordering technique uses as input the matrix to create the distance graph (see Section 3.3). Therefore, sampled matrices generate graphs that are different from the one produced by the original matrix. Due to the fact that sampled matrices have fewer nonzeros than original ones, the number of entry matches ( $a_{\text{elems}}$ ) will be smaller, so the graph will be reduced (small number of edges). Therefore, given that graphs are different and we are considering a heuristic strategy to minimize the total distance, the results will be different. Note that the TSP heuristic is limited to a fixed number of iterations searching for equilibrium between performance and overhead. In this way, in some cases, the reordering heuristic will find a better solution using a "sampled graph" than considering the original one because the problem to deal with is smaller.

Another observation is that performance results change depending on the considered number of threads. Note that reordered matrices are generated without taking into account the number of threads used to perform the SpMV (see Section 3.3). However, the sparse matrix is distributed among the threads to compute the SpMV in such a way that different computations are assigned to each thread. This distribution of computations depends on the number of threads, and different memory accesses are required by each thread to carry them out. Therefore, depending on the accesses performed by each thread the locality optimization technique will obtain different results. We must highlight that the performance of the parallel SpMV is determined by the slowest thread. Load balance can also influence the results for different number of threads.

Some of the observations regarding windows of fixed size agree with the behavior of the reorderings when using windows of variable size on the Itanium2 system (Figure 6). Note that considering only 1-2% of nonzeros is again enough to generate reorderings that show the same (or slightly different) performance in comparison with the original ones. However, the results show a higher variability with respect to the results in Figure 5. For example, considering few sampled nonzeros, noticeable improvements are achieved by the `mixtank.new` reorderings using four and eight threads, while there is some degradation with two threads. Irregular performance is also obtained by reorderings of matrices `nd3k`, `m3c10848` and `sme3Da`. Additionally to this, there are some cases in which reorderings based on sampled information always outperform those obtained by the original technique. This is the case of matrices `gyro_k` and `sme3Da`. We find the cause of this performance variability in the variable-size windows creation process of the locality optimization technique [27]. Note that the criterion in order to decide if two

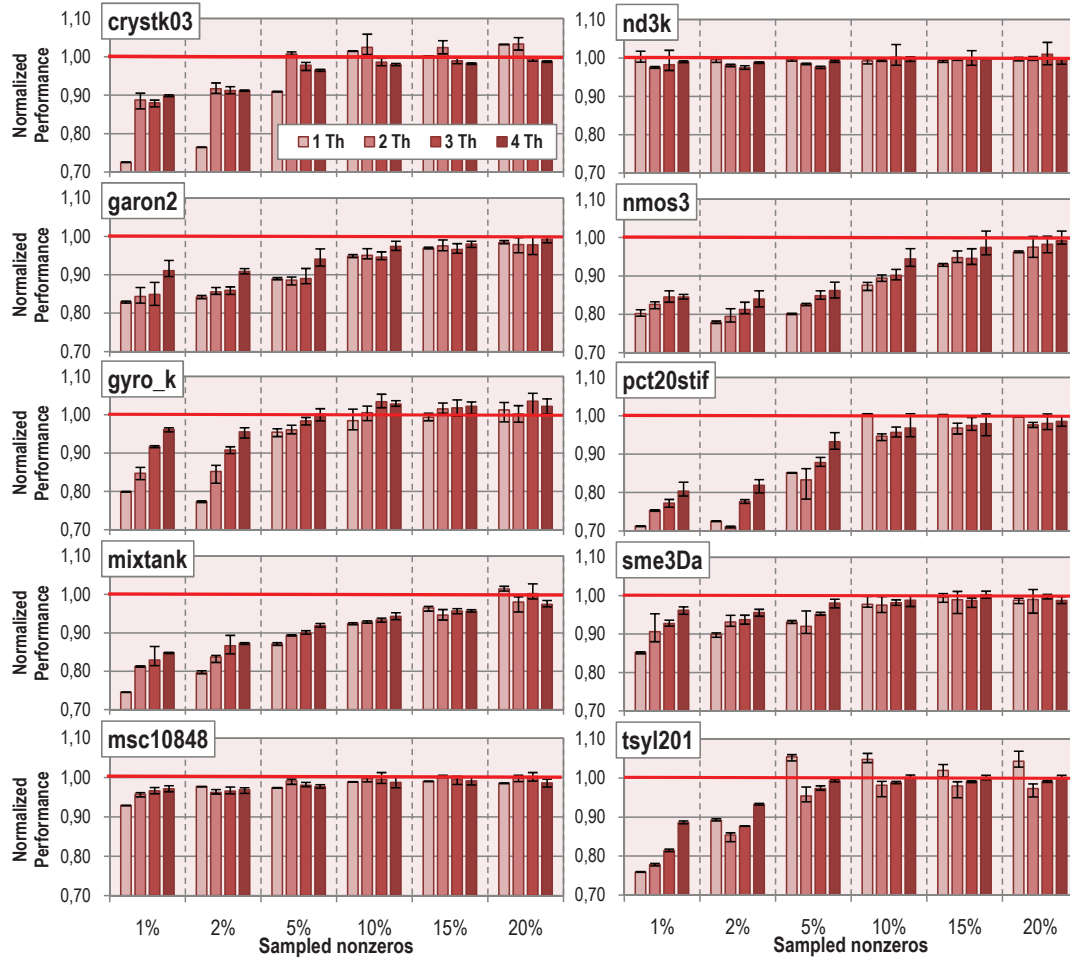


Figure 7. Normalized SpMV performance obtained by the reorderings generated using the locality optimization technique ( $w = 1$ ) and the information provided by the randomly sampled matrices on the Xeon platform.

consecutive rows/columns are within the same window depends on the locality estimation performed by the model (see Section 3.3). Therefore, different windows of locality (both in number and composition) are considered to calculate the permutation vector for each sampled matrix, which causes more variations in the performance results than in the case of using windows of fixed size.

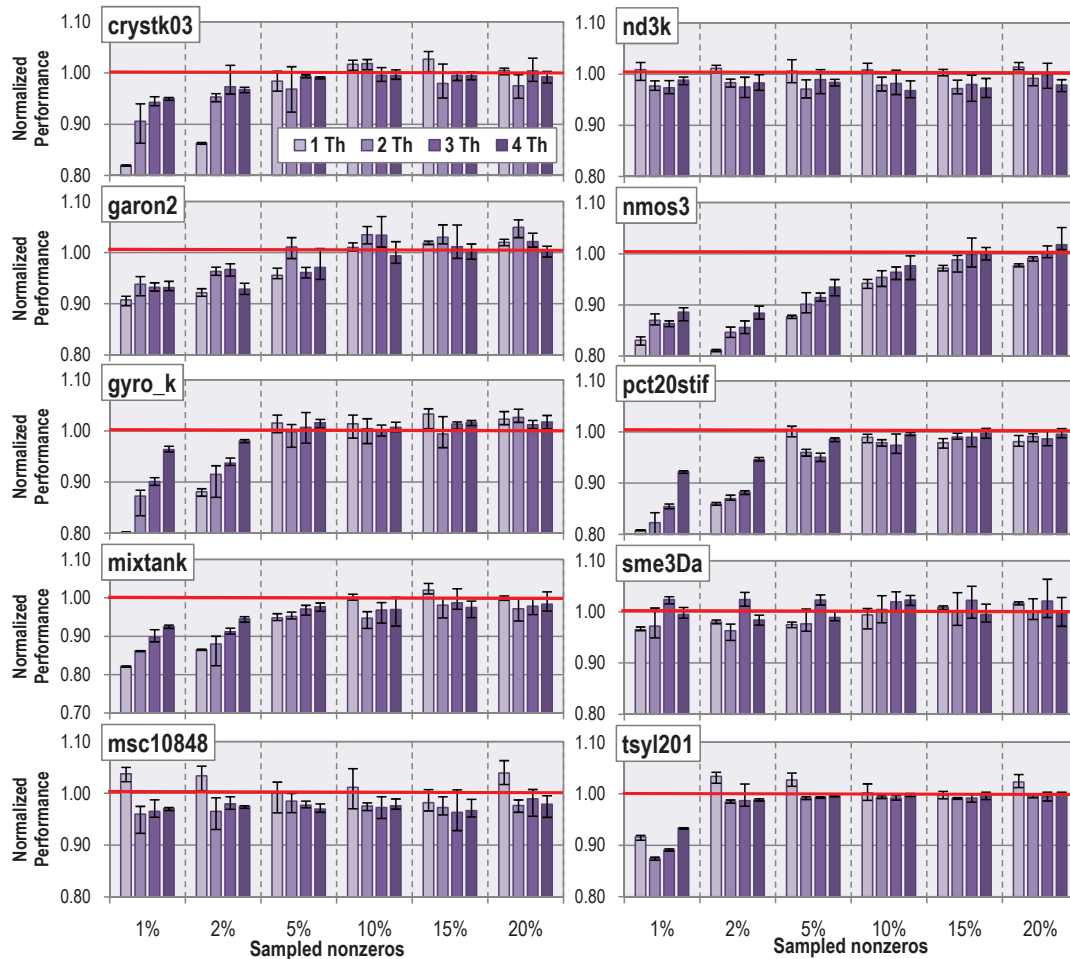


Figure 8. Normalized SpMV performance obtained by the reorderings generated using the locality optimization technique ( $w = variable$ ) and the information provided by the randomly sampled matrices on the Xeon platform.





Next we will focus on the performance evaluation on the Xeon processor system. Concerning the results using windows with  $w = 1$  (Figure 7), several conclusions can be made. First, unlike on Itanium2 system, the sampling percentage to achieve a similar performance with respect to the reorderings using complete information increases. It depends on the considered sparse matrix, ranging from 1% (matrix `nd3k`) to 20% (matrix `nmos3`). This behavior points out that this system penalizes more the small differences in the locality of the accesses than the Itanium2. This is caused by the different cache hierarchies (see Section 3.1). In particular, the Xeon processor has a smaller L3 cache and higher cache latencies. And second, performance follows the same trend in most of the cases, that is, it increases as the percentage of sampled nonzeros becomes higher.

When using windows of variable size on the Xeon system (Figure 8) higher sampling percentages are required to get closer to the original reorderings performance in comparison with the corresponding results on the Itanium2 (Figure 6). The behavior is similar to the observations for windows of fixed size detailed above. However, in most of the cases, 5% is enough for obtaining a performance near to 1. As on Itanium2, the results present more variability with respect to the fixed size case.

This study demonstrates that performing a data reordering to optimize the locality of the SpMV only considering a subset of the nonzeros of the sparse matrix is feasible. In the particular case of the Itanium2 system, a few number of nonzeros (typically 1-2%) is enough to obtain a similar performance with respect to the reorderings using complete information. On the Xeon system, the required percentage of sampled nonzeros increases, but even then, sampled information is enough for locality optimization. It has been tested using windows of fixed and variable sizes. However, important variations in the performance within each subset were observed when using randomly sampled matrices as input of the reordering technique. These fluctuations mean, in some cases, a variation higher than 10% between the maximum and the minimum performance.

## 5. Performance Evaluation Using Hardware Counters for Sampling

As second case of study, we will perform the sampling of the sparse matrices using the hardware counters provided by the Itanium2 processors. All modern processors include a Performance Monitoring Unit (PMU) implemented as a set of privileged registers. Typically, PMU configuration registers are programmed to measure certain events, and the results are collected by data registers which are usually counters. Many PMU models offer over hundred events which cover all aspects of the processor: pipeline, caches, TLBs, buses, interrupts, etc. In our case, Itanium2 PMU goes well beyond simply counting events, it can also precisely capture where they occur. For example, it can capture where cache and TLB misses occur as well as their latencies using the Event Address Registers (EARs) [17].

In order to access the EARs the *Perfmon2* [11, 12] monitoring interface has been used. *Perfmon2* performs an event-based sampling (EBS), that is, every time a given event occurs an EAR increments its value. After a number of events, a sample is collected and stored in a buffer. Every sample contains information such as the latency of the memory access, the memory address accessed, etc. When the buffer becomes full, an interruption is raised and

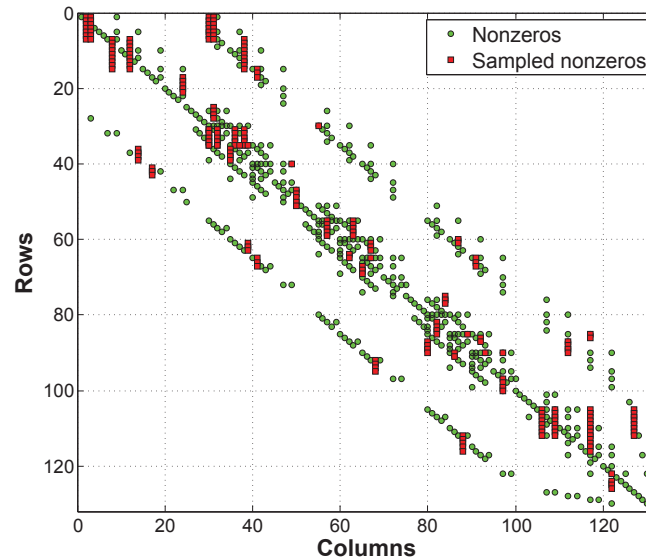


Figure 9. Example of sampled matrix by using the hardware counters.

the content of the buffer is available at user level. *Perfmon2* allows to monitor a particular memory address range.

We must highlight that floating-point operations on Itanium2 systems bypass the L1 cache in such a way that every access to a floating-point value generates a L1 cache miss [17]. Therefore, according to latencies in Table I, every access to a floating-point data has, at least, a 5 cycles latency (access to L2 cache). In this way, using “*cache misses with latencies higher than 4 cycles*” as event for sampling we assure that every access to array *x* and *y* is considered to be sampled. On the other hand, sampling information in this section was obtained at the lowest sampling rate.

A priori, by only getting the address of an access to array *x* we cannot determine, in an univocal way, the position of the accessed nonzero element on the matrix during the SpMV operation. This is because the counters only provide the address of the *x* element that misses the cache (this give us information about the exact column of the corresponding nonzero element of the matrix) and its latency. In this way, the identification of the row is not possible using this information. Therefore, in order to characterize the sparse matrix nonzeros by using the hardware counters we must find a way to obtain the row of the corresponding nonzero element of the matrix. With this purpose a methodology to obtain the position of an element in a matrix from the sampled information has been developed.

Accesses to arrays *x* and *y* are monitored at the same time. Some of the sampled events will be caused by accesses to *x*, whereas the others by the accesses to *y*. Note that accesses to *y*



are driven by loop  $i$  in the code of Figure 1. In this way, EARs provide a list of the sampled accessed elements. For example, considering seven sampled events, the result of reading the counters could be:  $y[0]$ ,  $x[23]$ ,  $y[1]$ ,  $y[2]$ ,  $x[12]$ ,  $x[19]$  and  $y[2]$ . As we have indicated above, accesses to  $x$  give us information about the exact column of the corresponding nonzero element of the matrix, while accesses to  $y$  provide information about the rows where the nonzero element can be placed. Note that with this measuring method we cannot obtain at the same time the row and column of an entry of the sparse matrix. However, the nonzero elements of the sparse matrix can be characterized according to the following features:

- Each sampled access to  $y[i]$  indicates that some nonzero element belonging to the row  $i$  of the matrix is being multiplied by one element of vector  $x$ .
- Each sampled access to  $x[j]$  indicates that a nonzero element of the matrix belonging to the  $j$  column has been accessed.
- We can state that the matrix has nonzero elements in the columns provided by the indices of the sampled  $x$  elements collected between two samples  $y[i]$  and  $y[i']$ . These nonzero elements are located in the rows within the interval  $[i, i']$ . From now on, these intervals are denoted as *uncertainty intervals*.
- As a particular case of the above property, if  $i=i'$  then the row is determined in an univocal way. Sampled nonzeros for which the row can be univocally determined will be called *univocal sampled entries* from now on.

Let's illustrate this methodology using the previous sampling example with seven samples. In this case, we can state that there are three nonzeros of the matrix in the positions  $([0,1],23)$ ,  $(2,12)$  and  $(2,19)$ . Note that for the first entry, we do not know exactly its row and only an interval of possible rows can be provided. In this case, the uncertainty interval is  $[0, 1]$ . The other two cases are univocal sampled entries. An example of the pattern characterization using this methodology with a real sparse matrix is shown in Figure 9 (in green circles the pattern of the matrix, and in red squares the sampled nonzeros). Note that uncertainty intervals are shown as a sequence of nonzeros along the same column in the sampled matrix. For example, intervals  $[0, 7]$  are in four columns, and  $[7, 16]$  in three.

Table III shows the results of applying our sampling methodology to the sparse matrices from the testbed. Only the univocal sampled nonzeros are considered. This information was collected after performing just one SpMV. The number and the percentage of univocal sampled entries with respect to the number of nonzero elements of the original sparse matrix are displayed. This percentage ranges from 2.9% (matrix `nmos3`) to 9.4% (matrix `nd3k`). Note that the percentage of sampled nonzeros will increase if a higher number of SpMV iterations is considered. The number of univocal sampled nonzeros per row is also shown in Table III. Figure 10 displays an example of sampled matrix obtained by using the hardware counters. In particular, the sampled matrix contains 7.3% of the nonzero elements of the original `sme3Da` matrix.

Next, a comparison between the reorderings obtained by the original technique and those obtained using the information provided by the hardware counters has been performed. Results are shown in Figure 11. SpMV performance is normalized with respect to the one obtained by the original technique. Reorderings using windows of fixed size with  $w = 1$  and variable size are analyzed. In most of the cases, reorderings guided by the information provided by the sampled matrices achieve very similar performance to those obtained by the original technique.

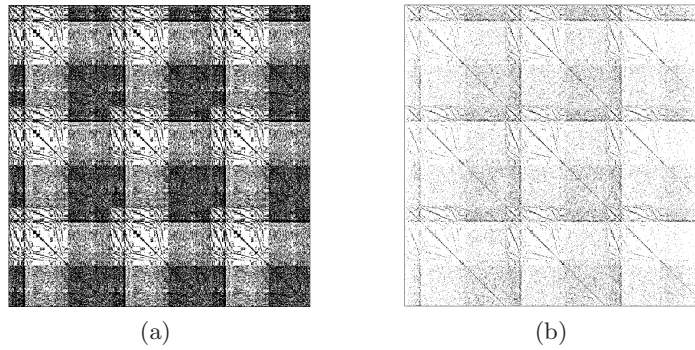


Figure 10. *sme3Da* original (a) and sampled matrix generated by using the hardware counters (b).

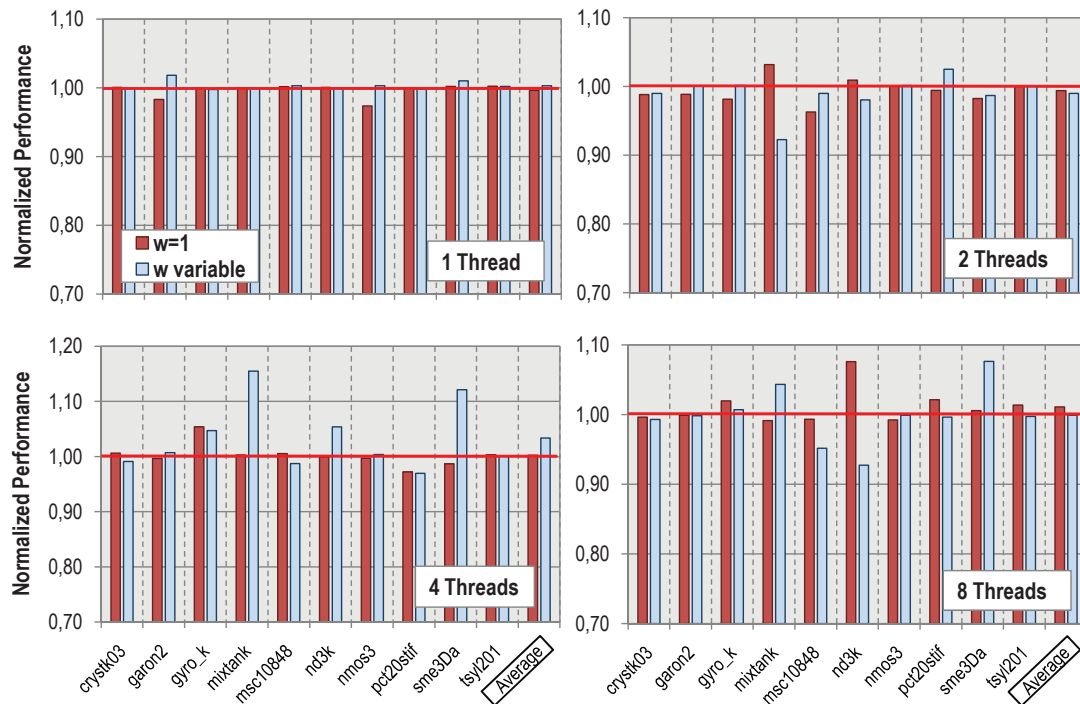


Figure 11. Normalized SpMV performance obtained by the reorderings generated using the locality optimization technique and the information provided by the hardware counters sampled matrices.



Table III. Characteristics of the sampled matrices generated by using the hardware counters.

Matrix	# sampled nonzeros (% w.r.t. original)	# sampled nonzeros/row
crystk03	128394 (7.3%)	5.2
garon2	17250 (4.4%)	1.3
gyro_k	70405 (6.9%)	4.1
mixtank_new	143226 (7.2%)	4.8
msc10848	101347 (8.2%)	9.3
nd3k	309116 (9.4%)	34.3
nmos3	11249 (2.9%)	0.6
pct20stif	176469 (6.5%)	3.4
sme3Da	63984 (7.3%)	5.1
tsyl201	204825 (8.3%)	9.9

A few exceptions can be found. For example, considering windows of fixed size, the original `pct20stif` reordering outperforms the sampled one when using four threads. The differences in the performance for these cases are, at most, about 3%. On the contrary, there are some sampling-based reorderings that achieve better performance with respect to the original ones. This is the case of matrix `nd3k` using eight threads, showing an improvement of 8%. A similar behavior is observed considering windows of variable size. Note that the magnitude of the performance variations for some matrices is slightly higher in comparison with the fixed size case. For example, an improvement of about 15% is achieved by the `mixtank_new` sampling-based reordering using four threads.

We have observed that reordered matrices generated using the information provided by different samplings of the same original matrix achieve very similar performance. Fluctuations in the SpMV performance for the reorderings of the same matrix are always lower than 2%. This is the reason why no maximum and minimum bars are displayed in the figure. We find the cause of this behavior in the event-based sampling (EBS) method using by *Perfmon2* library. For EBS, a sampling period is expressed as a number of occurrences of an event. According to [11], using a fixed sampling period may easily lead to biased results. This is the reason why *Perfmon2* uses a randomization of the sampling periods. In this way, hardware counters-based sampling performs a uniform EBS but not using a fixed sampling period.

According to the previous results we conclude that the sampled information provided by the hardware counters is enough for the locality optimization technique to generate quality reorderings.



Table IV. SpMV performance comparison between the original matrices and the reorderings obtained by the locality optimization technique (in GFlop/s). Reorderings were performed using the information provided by the hardware counters sampled matrices.

Matrix	1 Thread			2 Threads			4 Threads			8 Threads		
	Orig.	w=1	w=var.	Orig.	w=1	w=var.	Orig.	w=1	w=var.	Orig.	w=1	w=var.
crystk03	0.36	<b>0.38</b>	<b>0.37</b>	0.47	<b>0.50</b>	<b>0.49</b>	1.64	1.64	<b>1.70</b>	5.22	<b>5.32</b>	<b>5.31</b>
garon2	<b>0.53</b>	0.52	<b>0.53</b>	1.18	<b>1.21</b>	<b>1.21</b>	2.28	<b>2.38</b>	<b>2.39</b>	3.73	<b>4.66</b>	<b>4.66</b>
gyro_k	0.33	<b>0.34</b>	<b>0.34</b>	0.70	<b>0.79</b>	<b>0.76</b>	2.40	<b>2.47</b>	<b>2.57</b>	5.12	<b>5.31</b>	<b>5.36</b>
mixtank_new	0.33	<b>0.35</b>	<b>0.35</b>	<b>0.52</b>	0.48	0.48	1.03	<b>1.38</b>	<b>1.42</b>	2.29	<b>4.79</b>	<b>4.16</b>
msc10848	0.33	<b>0.35</b>	<b>0.35</b>	0.43	<b>0.59</b>	<b>0.60</b>	2.00	<b>2.54</b>	<b>2.44</b>	5.00	<b>6.08</b>	<b>5.68</b>
nd3k	0.33	<b>0.35</b>	<b>0.34</b>	<b>0.50</b>	0.47	0.46	0.77	<b>0.89</b>	<b>0.93</b>	2.42	<b>3.22</b>	<b>2.71</b>
nmos3	0.48	<b>0.49</b>	<b>0.50</b>	1.08	<b>1.10</b>	<b>1.09</b>	2.15	<b>2.18</b>	<b>2.18</b>	4.16	<b>4.26</b>	<b>4.27</b>
pct20stif	0.35	<b>0.37</b>	<b>0.37</b>	0.47	0.47	<b>0.48</b>	0.92	<b>0.99</b>	<b>0.99</b>	3.73	<b>3.76</b>	<b>3.74</b>
sme3Da	0.35	<b>0.37</b>	<b>0.37</b>	0.86	<b>0.91</b>	0.83	2.61	<b>2.68</b>	<b>2.67</b>	4.79	<b>5.64</b>	<b>5.25</b>
tsyl201	0.31	<b>0.32</b>	<b>0.32</b>	0.43	<b>0.44</b>	<b>0.44</b>	1.05	<b>1.06</b>	1.05	4.51	<b>4.54</b>	4.51
<i>Average</i>	<i>0.37</i>	<i>0.38</i>	<i>0.38</i>	<i>0.66</i>	<i>0.70</i>	<i>0.68</i>	<i>1.68</i>	<i>1.82</i>	<i>1.83</i>	<i>4.09</i>	<i>4.76</i>	<i>4.57</i>

### 5.1. Comparison with the non-reordered matrices

Until now we have only evaluated the SpMV performance with respect to the reorderings generated by the original technique. Therefore, a comparison with the original matrices (without reordering) is required in order to check if sampling-based reorderings obtain a better performance when the SpMV is executed. Results of this study considering reorderings performed using the information provided by the hardware counters are displayed in Table IV.

A first approach to the results points out that matrices obtained after applying the data reordering technique outperform the original ones. There are only a few cases that do not take profit from this reordering. SpMV performance improvements are up to 53%, which is the case of matrix `mixtank_new` running with eight threads. Note that, in this example, only 7.2% of the nonzeros of the matrix are considered to obtain the permutation vector (see Table III). On the other hand, as the number of threads increases, performance improvements caused by the locality optimization become more important. For example, considering windows of fixed size ( $w = 1$ ), the performance increases on average from 2.6% in the sequential case to 14.1% with eight threads. Finally, a slightly better behavior is observed for reorderings when using windows of fixed size instead of windows of variable size. This observation agrees with the conclusions of our previous works [27].

### 5.2. Reduction of the reordering technique overhead

One of the main drawbacks of the data reordering techniques is the reordering cost. It 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 [25]. As we show next, a consequence of using sampled information to calculate the permutation vector is an important reduction in the overhead introduced by the reordering technique.

Our reordering technique has two stages: the graph calculation and the generation of the permutation vector using the TSP heuristic. In both cases the reduction in the number of nodes of the graph decreases noticeably their overhead. If the reduction is in the number of

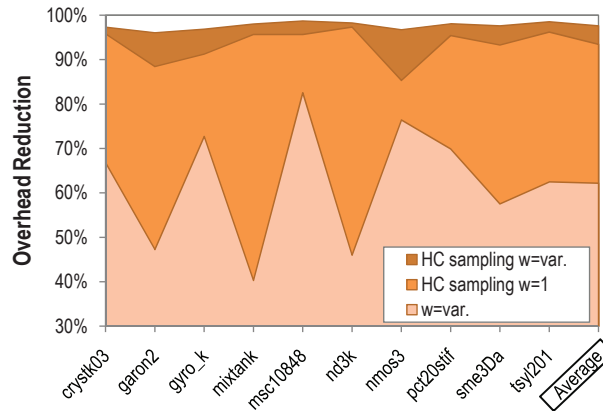


Figure 12. Overhead of the locality optimization technique using as reference the time required to perform the reordering using windows of fixed size  $w = 1$  and the original matrices (non-sampled).

edges the overhead also decreases but to lesser degree. Using sampled matrices we are reducing the number of edges and changing the edge weights in the graph. Note that there will be an edge between two nodes when at least there is an entry match (see Section 3.3). Given that sampled matrices have fewer nonzeros than the original ones, there is a high probability that edges between nodes with few entry matches in the original graph disappear in the graph generated using the sampled matrices. Note that these edge weights, according to our distance function, represent the worst cases regarding locality. Therefore, sampling reduces the cost of building the distance graph and its size. We must highlight that this distance graph is used as input of the TSP heuristic (Chained Lin-Kernighan algorithm). This heuristic is limited to a fixed number of iterations in such a way that most of the time is devoted to the graph calculation. TSP heuristic time dominates the overhead only when considering small matrices.

The analysis of the overhead is displayed in Figure 12. The plot shows the reduction in the reordering cost using as reference the time required by the technique when using windows of fixed size  $w = 1$  and the original non-sampled matrices.

When using windows of variable size and non-sampled matrices, the locality optimization technique reduces on average 62% the reference overhead. But this reduction is even more noticeable when sampled matrices are considered. In this case the overhead reduction on average reaches 93% and 98% depending on if fixed or variable size windows are used respectively. In a more precise way, the overhead expressed in terms of the number of SpMV operations and considering sampled matrices is on average 620 with  $w = 1$ , and 261 with  $w = \text{variable}$ . Note that the reference here is the computational time required to perform the SpMV operation on the original matrices.



---

## 6. Conclusions

In this work we have analyzed the possibility of increasing the locality of the sparse matrix-vector product (SpMV) when only a subset of the memory accesses performed is available in the optimization process. This is equivalent to consider only a subset of the nonzero elements of the matrix.

A data reordering technique previously developed by the authors [27] has been selected as locality optimization technique for the tests. It consists of reorganizing the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. The goal of this technique is to increase the grouping of nonzero elements in the sparse matrix pattern that characterizes the irregular accesses and, as a consequence, increasing the locality in the execution of the SpMV code. According to this, the technique must find the appropriate permutation of rows and columns of the original matrix for improving the locality using only the information provided by a subset of its nonzero elements. These nonzeros are obtained from the original matrices through a sampling process. In particular, two different sampling methods have been considered in this work: a random sampling and an event-based sampling (EBS) using hardware counters.

In the first case the nonzero elements of the sampled matrices used as input of the reordering technique are selected randomly from the original matrix in such a way that each nonzero has equal chances of being sampled. Tests have been performed on two systems consisting of different multicore processors: Itanium2 and Xeon (with Nehalem microarchitecture). Despite of fluctuations in the performance measurements caused by the randomly sampling method, results on both systems confirm that locality optimization guided only by sampled information is feasible. The main difference is the required percentage of sampled nonzeros to achieve similar results with respect to the reordering technique using complete information. While for Itanium2 system about 1-2% is enough, on the Xeon platform the percentage of nonzeros increases up to 10%.

Later, an event-based sampling process using hardware counters is considered. A new methodology to obtain the position of the nonzero elements of a sparse matrix from the sampled information provided by the hardware counters was introduced. The performance evaluation shows that reorderings generated using the information provided by these sampled matrices obtain very similar results with respect to the ones generated by the original technique. In particular, the percentage of sampled nonzeros ranges from 3% to 9%. As a consequence, using sampling-based reorderings leads to noticeable performance improvements with respect to the non-reordered matrices, reaching speedup values up to  $2.1\times$ . In addition, we have observed that there are very small differences in the SpMV performance achieved by reordered matrices generated using the information provided by different samplings of the same original matrix. Finally, note that the reduction in the amount of information managed by the optimization technique have an important impact on its computational cost.

Therefore, it has been demonstrated that using a subset of the memory accesses performed by the SpMV is enough for locality optimization, with the added benefit of an important reduction in the computational time required by the reordering technique.





---

## ACKNOWLEDGEMENTS

The authors also wish to thank CESGA (Galicia Supercomputing Center) for providing access to their supercomputing facilities.

## REFERENCES

1. P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.
2. D. Applegate and R. Bixby and V. Chvátal and W. Cook. *Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.
3. M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proc. of the Int. Conf. on Supercomputing*, pages 100–109, 2009.
4. B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the Intel Itanium 2 processor. In *Proc. of the ACM/IEEE conf. on Supercomputing*, page 58, 2004.
5. A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. *Int. Journal of High Performance Computing Applications*, 21(4):467–484, 2007.
6. J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O’Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proc. of Int. Symposium on Code Generation and Optimization*, pages 185–197, 2007.
7. Y. Choi, A. Knies, G. Vedaraman, and J. Williamson. Design and experience: Using the Intel Itanium2 processor performance monitoring unit to implement feedback optimizations. In *Proc. of EPIC2 Workshop*, 2002.
8. A. L. G. A. Coutinho, M. A. D. Martins, R. M. Sydenstricker, and R. N. Elias. Performance comparison of data-reordering algorithms for sparse matrix-vector multiplication in edge-based unstructured grid computations. *Int. Journal for Numerical Methods in Engineering*, 66(3):431–460, 2006.
9. E. Cuthill and J. McKee. *Several strategies for reducing the bandwidth of matrices*. Rose and Willoughby, 1972.
10. T. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997. <http://www.cise.ufl.edu/research/sparse/matrices>.
11. S. Eranian. *The Perfmon2 interface specification*, HP Labs, 2004.
12. S. Eranian. *Perfmon2: a standard performance monitoring interface for Linux*, 2008. <http://perfmon2.sourceforge.net/perfmon2-20080124.pdf>.
13. S. Eranian. What can performance counters do for memory subsystem analysis? In *Proc. of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 26–30, 2008.
14. B. B. Fraguera, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for blocked sparse algorithms. *Parallel Processing Letters*, 9(3):347–360, 1999.
15. E. J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proc. of the 10th SIAM Conf. on parallel processing for scientific computing*, 1999.
16. E. J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *Int. Journal of High Performance Computing Applications*, 18(1):135–158, 2004.
17. Intel. *Dual-Core Update to the Intel Itanium 2 Processor Reference Manual*, 2006. <http://download.intel.com/design/Itanium2/manuals/30806501.pdf>
18. Intel. *Turbo Boost Technology in Intel core microarchitecture (Nehalem) based processors*, 2008. <http://download.intel.com/design/processor/applnots/320354.pdf>
19. V. Karakasis, G. Goumas, and N. Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *Proc. of IEEE Int. Conf. on Computational Science and Engineering*, pages 247–256, 2009.
20. K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proc. of the Conference on Computing Frontiers*, pages 87–96, 2008.
21. J. Marathe, F. Mueller, and B. R. de Supinski. Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. *ACM Transactions on Architecture and Code Optimization*, 3(4):390–423, 2006.
22. D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.



23. 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*, pages 301–309, 1996.
24. 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.
25. J. C. Pichel, D. B. Heras, J. C. Cabaleiro, A. J. Garcia-Loureiro, and F. F. Rivera. Increasing the locality of iterative methods and its application to the simulation of semiconductor devices. *Int. Journal of High Performance Computing Applications*, 24(2):136–153, 2010.
26. 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.
27. J. C. Pichel, D. E. Singh, and J. Carretero. Reordering algorithms for increasing locality on multicore processors. In *Proc. of the IEEE Int. Conf. on High Performance Computing and Communications*, pages 123–130, 2008.
28. A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. of Supercomputing*, 1999.
29. M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *Int. Journal of High Performance Computing Applications*, 18(1):95–113, 2004.
30. M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel Distrib. Comput.*, 68(9):1186–1200, 2008.
31. S. Toledo. Improving memory–system performance of sparse matrix–vector multiplication. In *Proc. of the 8th SIAM Conf. on parallel processing for scientific computing*, March 1997.
32. R. Vuduc and H. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications, volume 3726 of Lecture Notes in Computer Science*, pages 807–816, 2005.
33. 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 the ACM/IEEE conf. on Supercomputing*, 2007.