Performance optimization of irregular codes based on the combination of reordering and blocking techniques

J. C. Pichel D. B. Heras J. C. Cabaleiro F. F. Rivera

Dept. Electrónica e Computación
Universidade de Santiago de Compostela
15782 Santiago de Compostela. Spain.
Tel.: +34 981563100 ext.: 13564, Fax: +34 981528012
jcpichel,dora,caba,fran@dec.usc.es

September 6, 2005

Abstract

The combination of techniques based on reordering data with classic code restructuring techniques for increasing the locality in the execution of sparse algebra codes is studied in this paper. The reordering techniques are based on, first modeling the locality in run-time, and then applying a heuristic for increasing it. After this, a code restructuring technique specially tuned for sparse algebra codes called register blocking is applied. The product of a sparse matrix by a dense vector ($SpM\times V$) is the code studied on different monoprocessors and distributed memory multiprocessors. The combination of both techniques was tested for a broad set of matrices from real problems and known repositories. The results expressed in terms of execution time show that an adequate reordering of the data improves the efficiency of applying register blocking, therefore, reducing the execution time for the sparse algebra code considered.

Keywords: irregular codes, data locality, reordering, blocking, multiprocessors.

1 Introduction

The hierarchical arrangement of memory in current computer architectures, called memory hierarchy, 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 the execution of a code [15]. This fact is specially important for the case of irregular codes [20] because the irregular accesses tend to present low spatial and temporal locality.

The fact above can be applied to monoprocessors, SMPs and also to distributed memory multiprocessors like some clusters of PCs. In the case of monoprocessors, the most costly level of the memory hierarchy is typically the main memory. So the main issue is to reduce the number of cache misses. In the case of distributed memory multiprocessors a lack of locality implies a movement of data between local and remote memories. And these are the most costly accesses in such computers. High locality in the accesses generated by each processor empowers the data reuse and consequently reduces the global execution time. Another important aspect for achieving high performance in such architectures is an adequate load balance.

A large number of algorithms for evaluating and optimizing data locality can be found in the literature. In the case of regular codes operating on dense matrices, most approaches for increasing locality are based on decreasing conflict cache misses by using *blocking*, *strip–mining* or other code restructuring techniques [23]. There are a variety of static models for selecting memory hierarchy transformations and parameters such as tile sizes for regular codes [23, 12, 3].

In the case of irregular codes, the techniques for increasing the locality can be mainly divided into two categories: restructuring code techniques and reordering data 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 [14, 5].

In previous works we have developed a model to characterize the locality and a procedure for increasing it based on reorganizing the data instead of changing or restructuring the code. The objective is increasing the grouping of elements in the pattern of the sparse matrix involved in the irregular code. As we demonstrated for monoprocessors and SMPs [16], 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 [21, 17] although in a more reduced context.

The potential of the locality improvement model, described in this paper, is on increasing the grouping of elements on the pattern of the matrix which allows to reach two objectives. First, our model can be applied for increasing the reuse of data in any level of the memory hierarchy, including cache memory and registers. Second, the grouping of entries over the pattern of the

matrix makes more efficient the subsequent application of any restructuring technique like, for example, blocking, given that it generates a structure of entries in blocks.

So, we have two groups of techniques that are complementary and can be successfully applied together as some authors have previously suggested. In [22] and [10] a combination of reordering algorithms and register blocking in different orders have been applied to the sparse matrix vector product. Good results were achieved only for some scientific matrices and only for monoprocessor machines. As far as we know, any author has systematically studied the combination of data reordering and code restructuring techniques for sparse matrix codes in distributed memory multiprocessors.

Along this paper we analyze, for several architectures, the performance of the reordering technique that we propose. We also prove that the combination of such reordering technique and register blocking is profitable in terms of execution time. The sparse matrix-vector product ($SpM\times V$) is an important computational kernel used in scientific computation, signal and image processing, document retrieval, and many other applications. This is the main reason why we have chosen it as representative example of irregular codes.

This paper is organized as follows. Section 2 presents a brief summary of the locality model, and the reordering data technique for improving the locality is detailed. In Section 3 a description of the main characteristics of the matrix benchmark suite and the different hardware platforms used are shown. The application of the reordering technique is detailed in Section 4. In Section 5 the results of the combination of the reordering technique and register blocking are shown. Section 6 presents a summary of the results obtained on applying our methodology, and finally, the main conclusions of this work are explained.

2 Locality modeling and improvement based on reordering data

In this section we briefly describe the locality improvement technique that we have developed and applied to a variety of sparse algebra codes and computer architectures in previous works. The locality model was detailed and applied to the $SpM\times V$ operation in [7]. A process to deal with the locality improvement problem applied to the same sparse algebra code was described and validated for monoprocessors in [6]. The application of the locality model for improving the locality was studied in [9] for the product of a sparse matrix by a dense matrix, and in [8] for the transposition of a sparse matrix. The model previously developed for monoprocessors was also extended to the case of NUMA shared memory multiprocessors in [16].

2.1 The locality model

From now on we consider 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 [13]. Our locality model is based on the evaluation of the data locality for the sparse algebra code considered. The model is general enough to include any sparse algebra code that can take profit of a clustering of entries in the pattern of the matrix. And, as we will explain below, the locality model initially developed for the case of monoprocessors can be easily extended to the case of distributed memory multiprocessors.

The locality model have been introduced in [7] for some sparse codes executed on single processor computers. In this model the locality is measured over consecutive pairs of rows or columns. It is based on evaluating two locality parameters: the number of *entry matches* (a_{elems}) and the number of *block matches* (a_{blocks}). The locality is modeled from these two parameters. The number of entry matches between any two rows of the sparse matrix is defined as the number of times that there are two nonzero elements in the same column of the matrix. The concept of entry matches can be extended to block matches in a straightforward way by considering instead of single entries, pieces of consecutive positions in a row of the matrix which include at least one entry. The definition of both concepts can directly be extended to columns.

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

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

where max_{elems} is the maximum number of elements in any row of the matrix, and $n_{elems}(x)$ is the number of elements in row x. These distance functions define metrics over the N rows or columns of the matrix [7].

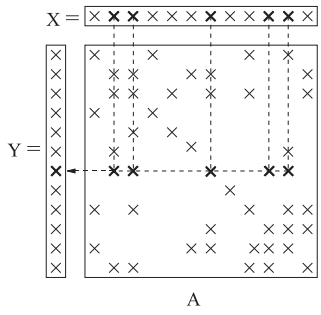
For a given sparse matrix two quantities, that are inversely proportional to the locality of the data for the whole sparse matrix, can be defined by summing the distances between pairs of consecutive rows or columns in the order they are accessed:

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

And this could be applied to any sparse algebra code whose locality is determined by the level of grouping of the entries in the pattern of the matrix.

```
DO I = 1, N
    REG = 0.0
    DO J = PTR(I), PTR(I+1)-1
        REG = REG + DA(J)* X(Index(J))
    ENDDO
    Y(I) = REG
ENDDO
```

(a)

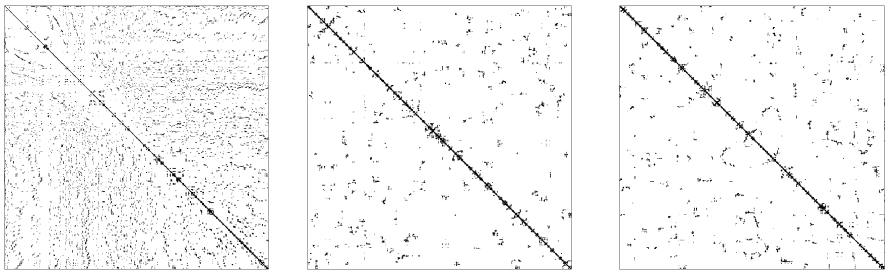


(b)

Figure 1: Algorithm for the product of a sparse matrix by a vector: (a) sequential algorithm and (b) data access.

This is, in particular, true for the $SpM\times V$ code. Let consider for this operation that the matrix is stored using the standard Compressed-Sparse-Row format (CRS) for unstructured sparse matrices [2]. DA, INDEX and PTR are the three vectors (data, column indices and row pointer) corresponding to this storage format. Using this format and considering that X is the vector by which the product is performed and Y the result vector, the product could be implemented as displayed in Figure 1(a). The data accesses required by this code to perform the product of a row of the matrix by the vector are displayed in Figure 1(b). According to the code in Figure 1(a), the sparse matrix is accessed in a row major order and a closer grouping of elements in a particular row of the matrix will lead to accesses to nearer elements of vector X, improving the spatial locality in the accesses to that vector. A closer grouping of nonzero elements between two or more consecutive rows of the matrix will produce an increase in the temporal locality achieved in the accesses to vector X. So, in general, the closer grouping of entries in the matrix pattern, the greater the locality in the accesses. In the case of distributed memory multiprocessors the same code displayed in Figure 1(a) can be used. It is only necessary to add some directives to specify which data are assigned to each processor. The code has to be tuned to establish the group of consecutive rows that must be multiplied in each processor. The rows have to be equally partitioned among the available processors. Given that they will operate over disjoint pieces of the original matrix, the problem of locality inside each processor is similar to the one of the monoprocessor case.

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



(a) (b) (c)

Figure 2: Example of the patterns of matrix *13*: (a) original, (b) reordered using d_1 and (c) reordered using d_2.

any level of the memory hierarchy. This issue is studied in the next section.

2.2 Locality improvement based on reordering data

We have outlined the relevance of searching for an increase of the locality in the execution of irregular codes. We propose to deal with this problem a heuristic technique consisting in modifying the pattern of the sparse matrix and evaluating the goodness of the modification performed by using the locality model described in the above section.

We formulate this problem of locality improvement as a classic NP-complete optimization problem, and we solve it as a graph problem using an analogy to the Traveling Salesman Problem (TSP). The problem is described using a graph where each node represents a row/column of the sparse matrix, and each edge has an associated weight that reflects the distance between rows/columns according to the description of locality given previously. The solution of the problem consists of two permutation vectors that indicate the appropriate order of rows or columns minimizing the value of the total distance defined by Expression 1. We have compared different heuristics for solving this problem, and finally we have chosen the *Lin-Kernighan* heuristic (LK). This is a local search heuristic that takes as an input a feasible but possibly suboptimal solution of the problem and repeatedly tries to improve it modifying the previous one. We use the *Chained Lin–Kernighan* implementation in *CONCORDE* proposed by Applegate *et al.* [1].

A solution of the locality problem implies, as we have indicated before, changes in the pattern of the sparse matrix. As an example, Figure 2 shows the change in the appearance of the pattern of matrix *13* of our test set (see Section 3) when the reordering process is applied. Note that the reordered matrices present a higher level of clustering of their entries than the original one. In the case of multiprocessors the situation is slightly different because we have different processors operating on disjoint parts of the initial matrix and performing a part of the $SpM\times V$. For this case we propose a locality

	Matrix	N	N_Z	Application Area
1	*bayer02*	13935	63679	Chemical process
2	*bcspwr10*	5300	21842	Power Networks
3	*crystk02*	13965	968583	FEM Crystal
4	*ex11*	16614	1096948	3D Steady flow
5	*goodwin*	7320	324784	Fluid mechanics
6	*jpwh991*	991	6027	Circuit physics
7	*lhr10*	10672	232633	Light hydrocarbon
8	*zenios*	2873	27191	Air traffic control
9	*mcfe*	765	24382	Astrophysics
10	*memplus*	17758	126150	Circuit Simulation
11	*bcsstk30c*	23000	1608696	Structural engineering
12	*li*	22695	1350309	3D Finite element
13	*msc10848*	10848	1229778	Structural engineering
14	*nc5*	19652	1499816	N-Body simulation
15	*syn12000a*	12000	1436806	Synthetic matrix

Table 1: Matrix benchmark suite.

improvement process divided into two steps:

- First, a reordering among rows over the whole sparse matrix is performed to obtain a partition of rows of the matrix among the processors.
- Second, a particular reordering among columns is performed over the portion of the matrix assigned to each processor.

So, once the whole sparse matrix is reordered, it is divided into as many submatrices as the number of processors, and then each submatrix is reordered by columns. By means of this stage the locality is optimized within each processor separately.

3 Matrix benchmark suite and hardware platforms

As a test set to evaluate the techniques for locality improvement we have selected fifteen square sparse matrices from different real problems that represent a variety of non-zero patterns. Table 1 summarizes the main characteristics of the matrices. N is the number of rows or columns, and N_Z is the number of entries. We have selected these matrices from the University of Florida Sparse Matrix Collection (UFL) [4], except matrices *11*, *14* and *15*. Matrix *11* corresponds to a square portion of 23000 rows from the original *bcsstk30* matrix (provided by the UFL collection). Matrix *14* was generated from an N-body simulation [18]. Finally, matrix *15* is a synthetic matrix generated randomly. According with the structure of the patterns, we can group the matrices in three different sets. In the first set those matrices that present a large number of small fixed-size dense blocks on their patterns are included. In general these are very banded matrices. For example,

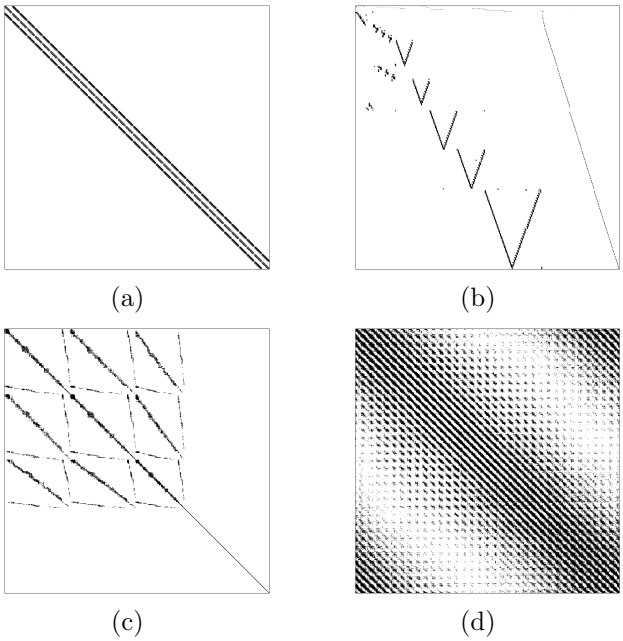


Figure 3: Examples of matrix patterns: Matrices (a) *3*, (b) *7*, (c) *8* and (d) *14*.

Finite Element Method (FEM) matrices such as matrix *3*. Its pattern is shown in Figure 3(a). Matrices *4*, *5* and *12* also belong to this set. The second set includes matrices with non-symmetric irregular patterns such as matrix *7* shown in Figure 3(b). Matrices *1* and *15* present the same kind of pattern. Finally, symmetric and nonsymmetric structured matrices belong to the third set. For example matrices *8* and *14*, shown in Figures 3(c) and 3(d) respectively. Matrices *2*, *6*, *9*, *10*, *11* and *13* are also included in this group. For performing the experiments, we have selected four different hardware platforms based on the following processors: *Mips R10000*, *Sun UltraSparcII*, *Sun UltraSparcIII*, and *Intel Pentium III*. Table 2 summarizes the main characteristics of these processors. The clock speed, number and size of the floating-point registers, the cache configuration and latencies of each processor are summarized in the table. Floating-point registers in the *UltraSparc* platforms are arranged in a way that some of them overlap, that is, are aliased [19]. Secondary level caches are external on the four platforms, and cache latencies in the table correspond to the worst cases.

4 Application of the reordering technique

In this section the application of the locality improvement technique to the $SpM\times V$ operation, described in Section 2, for the matrices and machines

	Mips R10000	**Sun UltraSparcII**	**Sun UltraSparcIII**	**Intel PentiumIII**
Clock rate	225 MHz	300 MHz	900 MHz	500 MHZ
FP Registers	32	32/32/16	32/32/16	8/8
FP Registers size	64 bits	32/64/128 bits	32/64/128 bits	80/128 bits
L1 data cache size	32 KB	16 KB	64 KB	16 KB
L1 line size	32 Bytes	32 Bytes	32 Bytes	32 Bytes
L1 latency	3 cycles	3 cycles	2 cycles	3 cycles
L2 cache size	2 MB	2 MB	8 MB	512 KB
L2 line size	128 Bytes	64 Bytes	512 Bytes	32 Bytes
L2 latency	12 cycles	9 cycles	14 cycles	10 cycles

Table 2: Summary of the main characteristics of the evaluation platforms.

detailed in Section 3 is studied. A preliminary study for a more reduced set of matrices and considering SMP architectures was described in our paper [16]. In particular, we evaluated the improvements achieved in terms of the reduction in the number of cache misses. Nevertheless, a locality improvement implies an increase in the reuse achieved at any level of the memory hierarchy: cache memory, registers, ... In this paper from now on, the results are expressed in terms of the quotient of the execution time of a solution respect to the execution time of a situation considered as reference. The results are shown in terms of execution time because, in fact, it is the most relevant magnitude for evaluating the performance.

4.1 Reordering on monoprocessor platforms

In particular in Figure 4 we show the execution time improvements achieved when executing the $SpM\times V$ code on a reordered matrix with respect to the execution time for the same code considering the original matrix. Each graph represents the results obtained for each one of the four monoprocessor architectures considered: *MIPS R10000*, *UltraSparcII*, *UltraSparcIII* and *PentiumIII*. The matrices in Table 1 are represented in the X-axis. The line in each graph indicates a value of 1 for the improvement in the execution time. This means that the execution time for the reordered matrix is the same than for the original matrix. For each matrix we show the results obtained when the reordering process is guided by the distance definitions d_1 and d_2.

The main conclusion from these results is that for most of the matrices some improvement is achieved, although they are generally small. Nevertheless even in the worst cases for which a worsening is obtained, this is under 5%, while the improvements for some cases achieve 18%. Note that for any matrix, the reuse of data achieved for the different architectures depends on the particular characteristics of the memory hierarchy: number and length of the registers, cache size, replacement algorithm,... It is also important to note that the results for the different matrices are quite different depending on the size and pattern of the matrix. For all the architectures only for

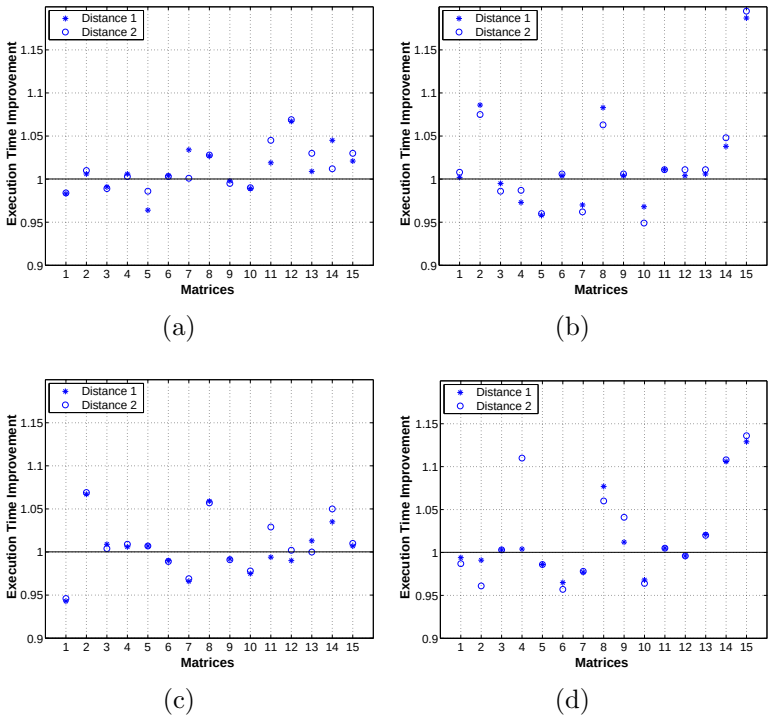


Figure 4: Execution time improvement obtained using LK compared to original matrix: (a) *Mips R10000*, (b) *UltraSparcII*, (c) *UltraSparcIII* and (d) *PentiumIII*.

matrix *10* no improvements are achieved. The reason is that this matrix presents a structure in narrow bands, with high initial locality and therefore with a poor margin for locality increase.

4.2 Reordering on multiprocessor platforms

We have also applied the reordering technique to the case of distributed memory multiprocessors. In particular to four different multiprocessors each of them built based on the microprocessors described in Table 2. The reordering technique for improving the locality in the case of multiprocessors was described in Section 2.2 and consists basically in performing an adequate reordering of the rows of the original matrix and after it, a different reordering of columns over only the portion of the matrix assigned to each processor. The results in terms of execution time improvements are represented in Figure 5 in four graphs corresponding to the cases of having a multiprocessor of one, two, three and four *Mips R10000* processors respectively. In each case the execution time improvements refer to the one achieved when executing the parallel $SpM\times V$ code over a reordered matrix with respect to the execution time for the same code but with the original

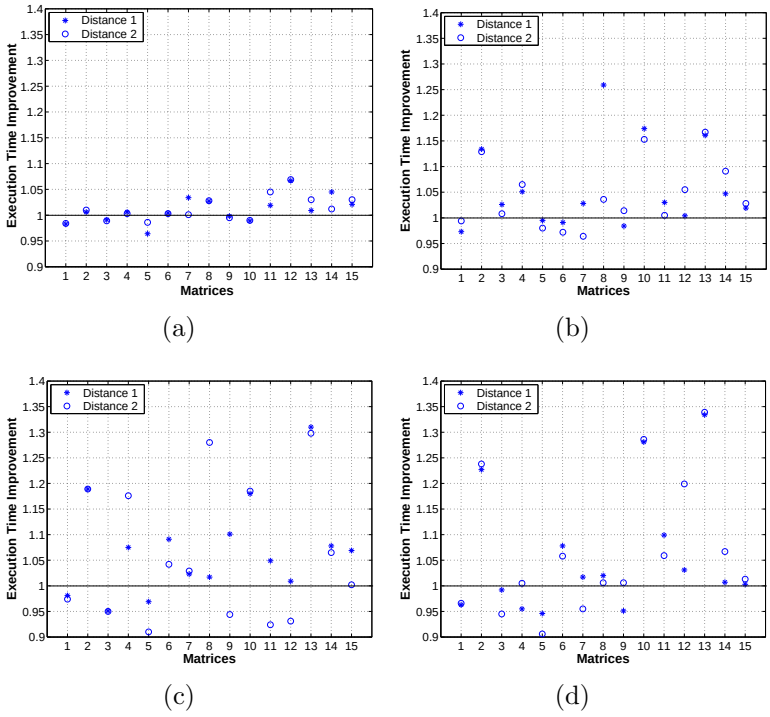


Figure 5: Execution time improvement obtained using LK compared to original matrix on a *MIPS R10000* multiprocessor using (a) 1, (b) 2, (c) 3 and (d) 4 processors.

matrix. When more than one processor is considered the execution time is the time spent by the most costly processor. Note that the results follow the same trend for the four cases. Nevertheless, the magnitude of the execution time improvements is greater when more processors are considered. The results considering the four different multiprocessors and three processors per each one are displayed in Figure 6. The behavior in terms of magnitude of the execution time improvements is similar for the different multiprocessors. The differences are due to the different characteristics of the memory hierarchy present in each computer. Note that, for instance, improvements up to 1.67 for the *UltraSparcII* platform were obtained.

5 Combining reordering and register blocking techniques

We are interested in combining the reordering technique that we have developed and explained in previous sections with code restructuring algorithms for optimizing the data locality.

Blocking is one of the most well-known code restructuring techniques and

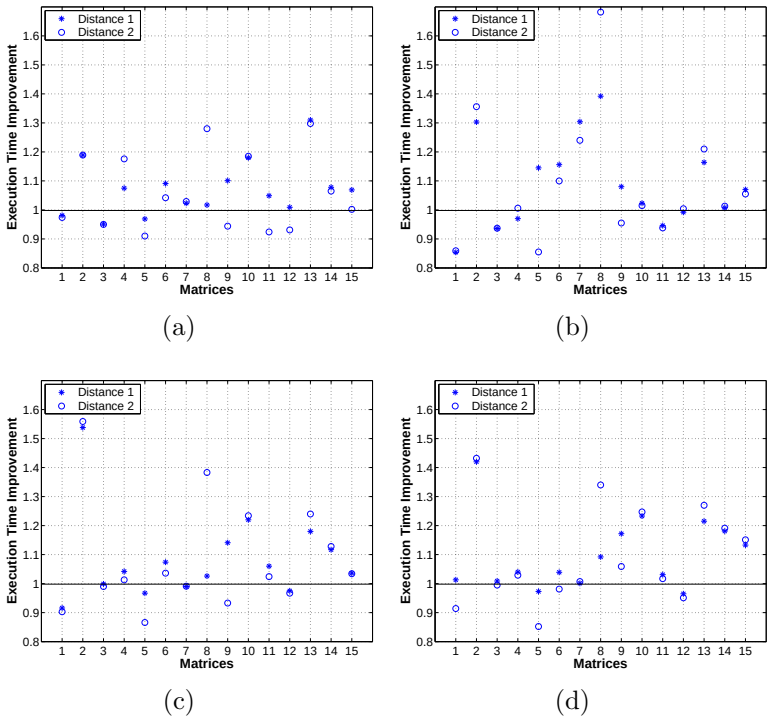


Figure 6: Execution time improvement obtained using LK compared to original matrix using three processors: (a) *Mips R10000*, (b) *UltraSparcII*, (c) *UltraSparcIII* and (d) *PentiumIII*.

it can be applied to different levels of memory hierarchy such as physical memory, caches and registers [23]. Blocking reshapes an iteration space over a data domain by partitioning it into pieces that fit into the selected level of hierarchy. In this way, all the computations on each piece can be completed before moving to the next one. Blocking rearranges the order of the computations, so that multiple references to a data element occur in inner loops while such element is still resident in the corresponding level of hierarchy.

In particular *register blocking* tries to eliminate loads and stores by reusing values stored in registers. We use the SPARSITY register blocking implementation [11]. While the idea of blocking for dense matrix operations is well known, the sparse matrix transformation is quite different. Register blocking reorganizes the sparse matrix into a set of fixed-size dense blocks by finding a small block size that fits into the target machines's register set. A reasonably small number of extra zeros elements are stored to compose dense blocks. These extra zero values increase the number of floating-point operations because they are involved as a part of the blocks in the sparse matrix computation. So, the matrix is stored as a sequence of small dense

blocks, and the computations are reorganized to compute each block before moving on to the next.

For selecting an appropriate register block size SPARSITY proposes an heuristic based on two components. First, an approximation for the performance rate of a matrix with a given block size. And second, an approximation for the amount of unnecessary computations that would be performed. This number depends on the ratio of computations before and after the addition of the extra zeros. These two components differ in the amount of information they require: the first only needs the target machine, whereas the second only needs the structure of the matrix. The register block size is, therefore, dependent on the nonzero structure of the sparse matrix. One of the main drawbacks of the register blocking technique is that it only tends to be effective on matrices that contain a large number of small fixed-size dense sub-blocks on their patterns. For example, Finite Element Method matrices (FEM). But it does not obtain good results with other types of matrices in the sense that the execution time of the blocked code is higher than the corresponding time for the original code.

Our proposal is effective for matrices with any structure. It performs a previous reorganization of the data. This reordering usually improves the effectiveness of applying a later restructuring technique such as register blocking. Other approaches that combine reordering and code restructuring techniques were presented for other authors in [17, 10].

5.1 Blocking and data reordering on monoprocessor platforms

In this section the application of register blocking to previously reordered matrices on a monoprocessor system is analyzed. As we have mentioned above, we expect that register blocking improves its performance when combining with reordering due to the closer grouping of entries that it causes in the matrix. And, this way, the creation of blocks in the pattern of the matrix is favored. Figure 7 shows the execution time improvement when combining reordering and register blocking respect to the execution time of the original sequential code with the original matrix on the four platforms introduced in Section 3. For the matrices labeled with an asterisk the register block size selected is 1×1, i.e., no register blocking is performed at all. Note that register blocking can be applied to each reordered matrix at least in one platform, except to matrix *15* which appears with asterisk in all the graphs. The best results are obtained for the *UltraSparcIII* platform. It is specially relevant the execution time improvement for FEM matrices such as matrix *3*, where even improvements of 3.44 are reached. After reordering these matrices (see Section 3), they also present dense sub-blocks in their patterns like the original ones. This fact makes specially efficient the application of register blocking. Figure 8 shows the original matrices to

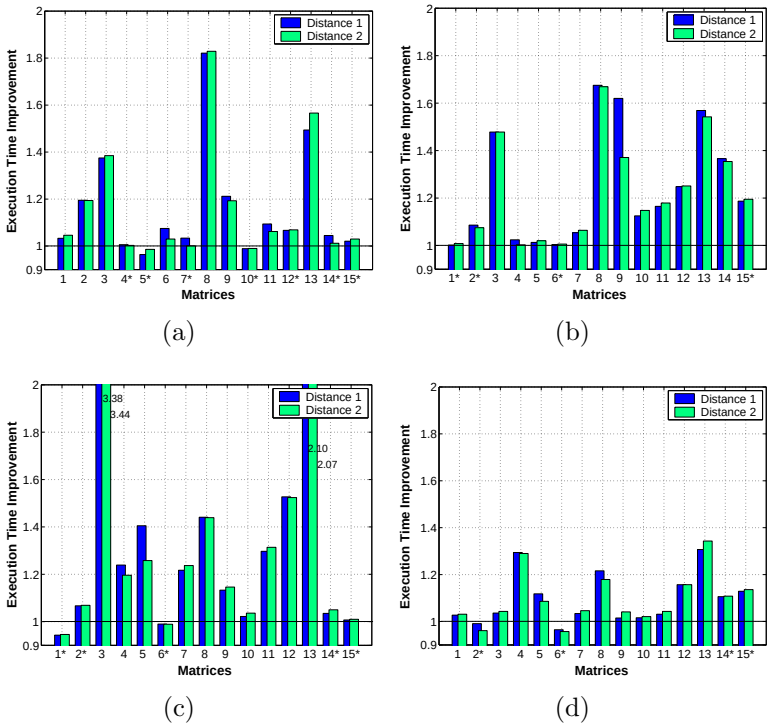


Figure 7: Execution time improvement obtained using LK+Blocking compared to original matrix: (a) *Mips R10000*, (b) *UltraSparcII*, (c) *UltraSparcIII* and (d) *PentiumIII*.

Matrix	d_1	d_2
3	**1.02**	**1.02**
4	0.99	0.99
13	**1.01**	**1.05**

(a)

Matrix	d_1	d_2
3	0.99	0.99
4	**1.02**	**1.01**
9	**1.31**	**1.11**
10	0.95	0.95
13	**1.02**	**1.01**

(b)

Matrix	d_1	d_2
3	0.99	**1.01**
4	**1.14**	**1.10**
7	**1.09**	**1.11**
11	**1.17**	**1.18**
13	**1.01**	**1.01**

(c)

Matrix	d_1	d_2
3	**1.03**	**1.04**
4	**1.02**	**1.02**

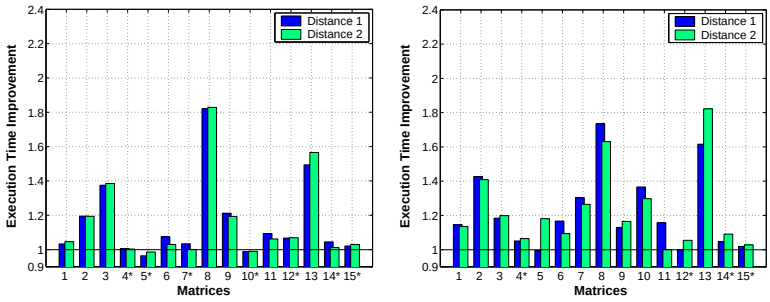
(d)

Figure 8: Execution time improvement of blocked code on reordered matrices with respect to blocked code on original matrices on the four hardware platforms for distances d_1 and d_2: (a) *Mips R10000*, (b) *UltraSparcII*, (c) *UltraSparcIII* and (d) *PentiumIII*.

which register blocking can be applied with success without performing a previous reordering. Register blocking on original matrices only is effective for a reduced number of them. In particular, for matrices *3* and 4 on all the platforms. The execution time improvement of the blocked code for the reordered matrices with respect to the blocked code for the original matrices is also shown. As we have shown in Figure 7, the effectiveness of register blocking increases remarkably when using reordered matrices. Comparing the execution time respect to the blocked code for the original matrices, the results show that, except for a few cases, an improvement is achieved. For instance, this improvement is up to 1.31 for matrix *9* on the *UltraSparcII* platform.

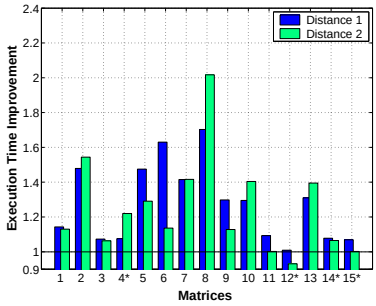
5.2 Blocking and data reordering on multiprocessor platforms

In this section we show the results of combining reordering and register blocking on four different multiprocessors. The sparse matrix-vector product was executed using 1, 2, 3 and 4 processors. In the multiprocessor case each processor only computes the product of some consecutive rows of the matrix. As we have commented we divide the initial matrix into p submatrices (being p the number of processors), and then each submatrix is reorganized independently for increasing the locality. A reorganization of the data to increase the locality within each submatrix is performed individually. This way register blocking can be applied to each submatrix with a different block size, and it could happen for some submatrices that the application of register blocking would be inefficient. In the figures of this section, those reordered matrices for which register blocking is inefficient for at least one of its submatrices are labeled with an asterisk. As an example, Figure 9 shows the execution time improvement of the blocked code on reordered matrices with respect to the non-blocked code on original matrices on a multiprocessor system based on *Mips R10000*. Note that, generally, the improvements grow when the number of processors increases. On this platform we achieve improvements up to 2.2 using 4 processors. As in Figure 8, Figure 10 shows the original matrices to which register blocking can be applied directly, and the execution time improvement of the blocked code for the reordered matrices respect to the blocked code for the original matrices using 1, 2, 3 and 4 processors. The behavior using several processors is similar to the monoprocessor case shown in Figure 10(a). That is, register blocking is only effective for the original matrices if they present blocks in their structure. Anyway, as we point out in Figure 10, the combination of the two techniques produces better results than applying only register blocking. Execution time improvements applying reordering and register blocking with respect to the application of only register blocking on the original matrices are always achieved when the product is performed using

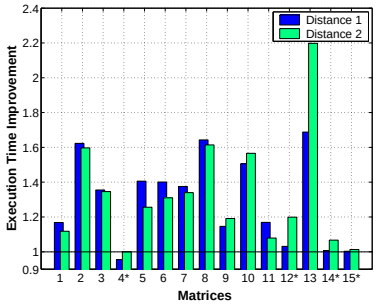


(a)

(b)



(c)



(d)

Figure 9: Execution time improvements for the blocked code on reordered matrices compared to the non-blocked code on original matrices on a *MIPS R10000* multiprocessor using (a) 1, (b) 2, (c) 3 and (d) 4 processors.

Matrix	d_1	d_2
3	**1.02**	**1.02**
4	0.99	0.99
13	**1.01**	**1.05**

(a)

Matrix	d_1	d_2
3	**1.03**	**1.05**
13	**1.01**	**1.14**

(b)

Matrix	d_1	d_2
1	**1.14**	**1.13**
3	**1.03**	**1.01**
4	**1.01**	**1.14**
13	**1.02**	**1.09**

(c)

Matrix	d_1	d_2
1	**1.16**	**1.11**
3	**1.06**	**1.05**
13	**1.12**	**1.46**

(d)

Figure 10: Execution time improvement of blocked code on reordered matrices with respect to blocked code on original matrices on a MIPS R10000 multiprocessor for distances d_1 and d_2 using (a) 1, (b) 2, (c) 3 and (d) 4 processors.

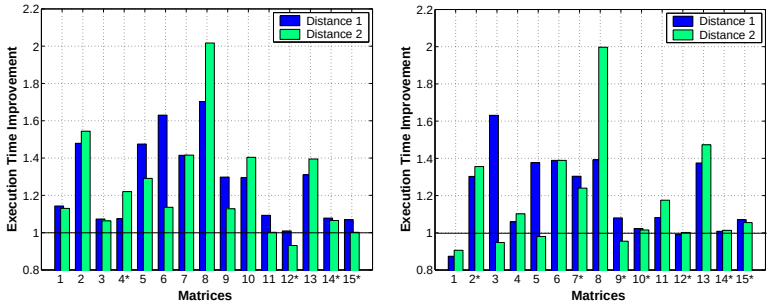


Figure 11: Execution time improvements obtained using LK+Blocking compared to original matrix using three processors on the four multiprocessor platforms: (a) *Mips R10000*, (b) *UltraSparcII*, (c) *UltraSparcIII* and (d) *PentiumIII*.

more than one processor. These improvements are up to 1.46 using 4 processors. Finally, Figure 11 shows that the results are qualitatively similar for all the platforms. In this case the results are shown for three processors. In general, the execution time improvements are higher than those obtained for the monoprocessors. Comparing the results between multiprocessor platforms it can be observed that the best overall improvements are achieved using an *UltraSparcIII*, as in the sequential case. The lowest improvements are obtained on the *Pentium III* cluster, and even in this case they are up to 1.54.

6 Summary of results

In this section we summarize for all the platforms the results obtained using only the reordering technique, or the combination of register blocking and reordering when it is possible. Table 3 shows, as example, the results for the monoprocessor case in terms of the percentage of execution time improvement. The improvements obtained for the two distance functions (d_1 and

Matrix	d_1 (%)	d_2 (%)	**Global (%)**
1	0.2	0.8	0.5
2	7.8	6.9	7.4
3	45.0	45.5	45.3
4	12.3	11	11.7
5	11.1	8.0	9.6
6	0.8	-0.4	0.2
7	7.8	8.0	7.9
8	35.0	34.6	34.8
9	19.7	15.8	17.8
10	4.2	4.7	4.4
11	11.3	12.7	12.0
12	20.0	20.0	20.0
13	38.2	38.8	38.5
14	12.1	11.6	11.9
15	7.9	8.5	8.2
Average	15.6	15.1	15.3

Table 3: Summary of the best results in terms of percentage of execution time improvement for the monoprocessor case on all the hardware platforms.

d_2) are very similar. From this table we can conclude that the best results are obtained for matrices that belong to matrix groups 1 and 3 described in Section 3. Group 1 includes matrices that present small dense blocks on their patterns. Matrices *3*, *4*, *5* and *12* belong to this group. In this case the average execution time improvement is up to 45.3%. The structured matrices of the third group also obtain very good results (matrices *2*, *6*, *8*, *9*, *10*, *11*, *13* and *14*). Finally, matrices belonging to the second group (with irregular patterns), also achieve improvements. Matrices *1*, *7* and *15* are in this group. The total average percentage of improvement is about 15% for both distance functions. As we have commented in the previous section, the execution time improvements increase when the number of processors implied in the product is higher. For example, the total average improvement performing the sparse matrix-vector product using four processors is about 20%.

7 Conclusions

The combination of different techniques for increasing the locality in the execution of irregular codes is studied in this paper. The techniques are data reordering and code restructuring ones. We have chosen the sparse matrix-vector product ($SpM\times V$) as a representative kernel of the irregular codes whose locality properties depend on the clustering of entries on the pattern of the sparse matrix. The architectures considered are monoprocessors and distributed memory multiprocessors.

A model of locality and a procedure for increasing it were presented. Our

proposal is based on performing a reorganization of the data with the aim of increasing the grouping of elements in the pattern of the matrix, and this way, improving the locality. Along this paper we analyze, for several architectures, the performance of the reordering technique that we propose. On applying it, significant decreases in the execution times for a representative set of matrices from real scientific problems have been obtained.

After the reordering, a code restructuring technique specially tuned for sparse algebra codes called register blocking is applied. Register blocking reduces the number of loads and stores by reusing values that are in registers.

The combination of both types of techniques is profitable in terms of execution time. This is tested for a broad set of matrices and for different monoprocessors and distributed memory multiprocessors. The results also show that an adequate reordering of the data increases the possibility of successfully applying register blocking, and therefore, of reducing the execution time for the sparse algebra code considered.

Acknowledgements

This work was supported by the Spanish Ministry of Science and Technology (MCYT) under project TIC2001-3694-C02-01.

References

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

[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 2001 Conference 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] B. B. Fraguela, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for blocked sparse algorithms. *Parallel Processing Letters*, 9(3):347–360, September 1999.

[6] D. B. Heras, V. Blanco, J. C. Cabaleiro, and F. F. Rivera. Modelling and improving locality for the sparse matrix-vector product on cache

memories. *Future Generation Computer Systems. Special Issue on High Performance Numerical Methods and Application*, 18(1):55–67, 2001.

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

[8] D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Analysis and improvement of data locality for the transposition of a sparse matrix. In F. J. Peters G. R. Joubert, A. Murli and M. Vanneschi, editors, *Proceedings of the ParCo2001. Parallel Computing: Advances and Current Issues*, pages 457–464. Imperial College Press, 2002.

[9] D. B. Heras, F. F. Rivera, and J. C. Cabaleiro. Analytical description of locality for the product of a sparse matrix by a dense matrix. In *Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 178–184, 2002.

[10] 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*, March 1999.

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

[12] 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.

[13] 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*, October 1996.

[14] 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.

[15] D. A. Patterson and J. L. Hennesy. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers, 1996.

[16] 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 muliprocessors. In *Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP2004*, pages 66–71. IEEE Computer, 2004.

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

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

[19] Corporate SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., 1994.

[20] 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.

[21] 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.

[22] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[23] 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.