Improving the Locality of the Sparse Matrix–Vector Product on Shared Memory Multiprocessors

J. C. Pichel D. B. Heras J. C. Cabaleiro F. F. Rivera
Dept. Electrónica e Computación
Universidade de Santiago de Compostela
15706 Santiago de Compostela. Spain.
jcpichel,dora,caba,fran@dec.usc.es

Abstract

In this paper we extend a model of locality and the subsequent process of locality improvement previously developed for the case of sparse algebra codes in monoprocessors to the case of NUMA shared memory multiprocessors (SMPs). In particular the product of a sparse matrix by a dense vector ($SpM \times V$) is studied. In the model, locality is stablished in run-time considering parameters that describe the structure of the sparse matrix involved in the computations. The problem of increasing the locality is formulated as a graph problem whose solution indicates some appropriate reordering of rows and columns of the sparse matrix. The reordering algorithms were tested for a broad set of matrices. We have also performed a comparison with other reordering algorithms. The results lead to general conclusions about improving SMP performance for other sparse algebra codes.

1. Introduction

Sparse matrix-vector multiplication ($SpM \times V$) is an important computational kernel used in scientific computation, computer graphics algorithms and many other applications. The primary reason for inefficiencies of sparse matrix codes such as the $SpM \times V$ is the poor data locality in the irregular accesses. Due to the indirect addressing, memory hierarchy is generally considered inefficient for irregular codes [13, 10]. Their unpredictable behaviour implies a lot of work in the development of special implementations of the codes and when tuning the performance of such codes. This fact is specially important on SMPs (NUMA Shared Memory Multiprocessors) because remote accesses, coherence and consistence mechanism are very costly [15].

A large number of algorithms for evaluating and optimizing data locality can be found in the literature. In the case of dense codes, most approaches are based on decreasing conflict misses by using *blocking*, *strip–mining* or other code restructuring techniques [16]. Some of these techniques have been applied to irregular codes as, for example, to different versions of the product of a sparse matrix by a dense matrix [8].

We are interested in applying a technique that is complementary to the restructuring techniques referenced above. Instead of changing the code, we perform a reorganization of data with the aim of increasing the grouping of elements in the pattern of the matrix. Therefore, the reordering could improve the effectiveness of a later restructuring technique. Our technique is effective for matrices with any structure. Similar approaches performing a reordering inside the data structures have been successfully applied in other papers [14, 10].

Some ordering techniques like *bandwidth reduction* algorithms, which derive from the *Cuthill-McKee* algorithm, and the *Minimum Degree* algorithm based heuristics as the *Approximate Minimum Degree* algorithm, among others [12, 1], are considered classical methods for dealing with the problem of improving the locality in the execution of a sparse code. These standard reordering algorithms do not ensure good results as oppossed to the reordering algorithms that we have developed [5].

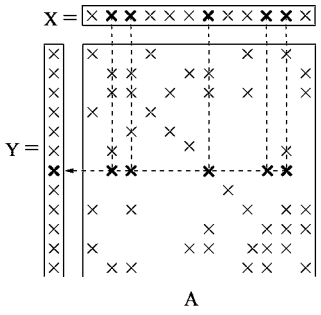
2. Data locality for the $SpM \times V$

In this work we use the standard Compressed-Sparse-Row format storage (CSR) for unstructured sparse matrices. DA, INDEX and PTR are the three vectors (data, column indices and row pointer) corresponding to this storage format. Based on this format the $SpM \times V$ product can be implemented as shown 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).

The starting point for our analysis on SMPs is to analyze the locality properties of the execution of the code in

```
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) Sequential algorithm



(b) Data accesses

Figure 1. Algorithm for the product of a sparse matrix by a vector.

Figure 1(a). We will consider [7] 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.

Given that each element of the matrix is multiplied only once, the accesses to DA, INDEX and PTR do not yield temporal locality. Each element of Y is used repeatedly a number of times and the elements are accessed consecutively so there is a possibility of having both spatial and temporal locality. The temporal locality is exploited by the use of the CPU registers in the code of Figure 1(a).

The locality is more unpredictable for vector X. For this case neither spatial locality nor temporal locality are ensured because there is a dependence on the pattern of the matrix which determines the elements of X accessed in the product. Therefore, the greatest potential for locality improvement is on vector X.

In the case of a SMP we can use the same code as the one presented in Figure 1(a). It is necessary only to add some directives to specify data partitioning. So each processor computes the product of only some consecutive rows of the original matrix. Assuming that the product is computed using the "owner-computes" rule, the locality properties of the code are similar to those for the case of a single processor.

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, the closer the grouping of entries in the matrix pattern, the greater the locality in the accesses described.

Given that each element of Y is updated by only one processor there is no sharing nor invalidations. So, the most costly accesses will be those caused by primary cache misses and mainly by secondary cache misses.

3. Modelling the locality

For the $SpM \times V$ operation as we have indicated in the previous section, as for other sparse algebra codes, the reuse achieved when the code is executed will depend on the locality properties and also on the characteristics of the memory hierarchy. And this holds for both single processor and multiprocessor computers.

To model the locality for a SMP we have used a model we have introduced [4] for some sparse codes executed on single processor computers. We are going to briefly outline the model. In this model the locality is measured over consecutive pairs of rows or columns. The model is based on two locality parameters: number of *entry matches* (a_{elems}) and number of *block matches* (a_{blocks}). The measurement of locality is based on 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 sparse matrix. The concept of entry matches can be extended to block matches in a straightforward way. The definition of both concepts can be extended for columns. For the case of the $SpM \times V$ operation, these parameters in terms of rows are related to the temporal locality of X. And when they are used in terms of columns, they are related to the spatial locality of X and Y.

Based on these two parameters we have defined a magnitude called *distance between rows x and y*, denoted by *d(x,y)*. It is used to measure the locality displayed by the irregular accesses performed by the sparse irregular code on these two rows. Two proposals for *d(x,y)* were introduced:

$$d_1(x,y) = max(n_{elems}(x), n_{elems}(y)) - a_{elems}(x,y)$$
$$d_2(x,y) = n_{blocks}(x) + n_{blocks}(y) - 2{*}a_{blocks}(x,y)$$

where $n_{elems}(x)$ is the number of elements in row x and $n_{blocks}(x)$ is the number of blocks of elements in row x. Each distance function defines a metric over the N rows or columns of the matrix [4].

For a given sparse matrix, two quantities, that are inversely proportional to the locality of the $SpM \times V$ for the whole sparse matrix, can be modelled by summing the dis-

tances between pairs of consecutive rows/columns in the order they are accessed:

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

For the case of a SMPs the locality must be high in each processor. Thus, the locality will be modelled for any processor separately by considering only the portion of the original sparse matrix that the processor needs for its computations.

4. Data locality improvement

For the parallel $SpM{\times}V$ operation it has been shown in Section 2 that some benefits, in terms of locality, can be obtained from grouping the entries of the submatrix corresponding to each processor. One way of achieving this objective is to search for the permutation matrices of rows and columns of the submatrix, that will produce an increase in the locality properties in the corresponding processor. These reorderings produce a decrease in the value of the quantities D_1 or D_2 (See Equation1) for the submatrix considered.

As we have explained in section 2, for our implementation of $SpM \times V$ in SMPs, it will be necessary to increase the locality of the accesses to primary and secondary caches in each processor. We formulate the problem of locality improvement as a NP–complete problem [9] whose solution is two permutation matrices. Given that we have measures for evaluating the adequateness of an ordering, we have opted for heuristic solutions based on graphs [11]. In such a way that solving our problem is equivalent to finding a path of minimum length that goes through all the nodes of a complete graph.

Until now we had proposed and extensively proven the two following solutions to the problem of locality on a single processor system:

- The first solution begins with the construction of a minimum–spanning tree of the complete graph using *Prim's algorithm*. The vertices of the tree are visited to establish an order of the nodes using a *depth–first search*.
- The second solution for solving our problem uses the greedy heuristic called *Nearest Neighbour algorithm* [11] to establish the order of rows and columns of the matrix.

Any of the two previous algorithms was proven with any of the two distance functions guiding the improvement process on single processor systems. Nevertheless, after a large number of experiments we concluded that the first algorithm (based on *Prim's algorithm*) with distance function d_1 and the second algorithm (based on the *Nearest Neighbour heuristic*) with distance function d_2 gave the best results for a variety of sparse matrices and operations studied.

Matrices	Name	N	N_Z
M1	*MSC10848*	10848	1229778
M2	*NC5*	19652	1499816
M3	*BCSSTK30*	23000	1608696
M4	*LI*	22695	1350309
M5	*Synthetic Matrix*	12000	1436806

Table 1. Scientific computing matrices used in the experiments.

The results obtained by these techniques present a large dependency on the pattern of the sparse matrix. For solving this graph problem of increasing the locality (and also guided by distance functions d_1 and d_2) we introduce a new heuristic. This new heuristic is based on the *Lin–Kernighan* algorithm that is one of the most successful tour-finding approaches. This is a local search heuristic that takes as an input a feasible but possibly suboptimal solution of the problem. It repeatedly tries to improve the solution modifying the previous one. The *Lin–Kernighan* heuristic can find improving exchanges involving many edges. In principle it can exchange almost all the edges in a single move. We use the *Chained Lin–Kernighan* implementation in *CONCORDE* proposed by Applegate *et al.* [2].

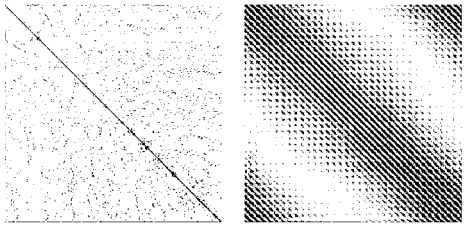
5. Reordering matrices for shared memory multiprocessors

In this section we compare the six results obtained using the three heuristic solutions and the two distance functions proposed in the previous section. P_i denotes the solutions obtained using *Prim's algorithm*, P_{in} corresponds to the solution using the *Nearest Neighbour algorithm*, and LK_i indicates that the solution is obtained by means of the *Lin–Kernighan algorithm*. In all cases i indicates the distance function employed for guiding the optimization process ($i = 1$ for d_1 and $i = 2$ for d_2).

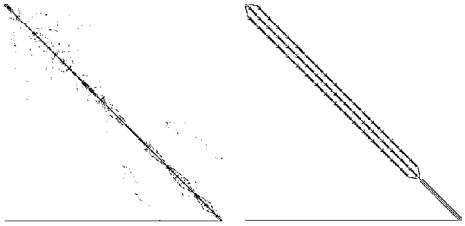
The process of reordering data is performed in two steps:

1. We perform a reordering over the whole sparse matrix to obtain a later partition of rows of the matrix among the processors achieving good load balancing.
2. A particular locality improvement is applied over the portion of the matrix accessed by each processor.

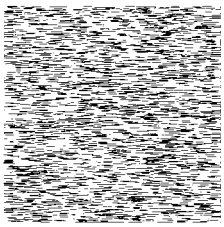
So, the initial matrix is divided into p submatrices (one per processor), and each one is reordered by columns independently, obtaining the final new reordered matrix.



(a) M1 (b) M2



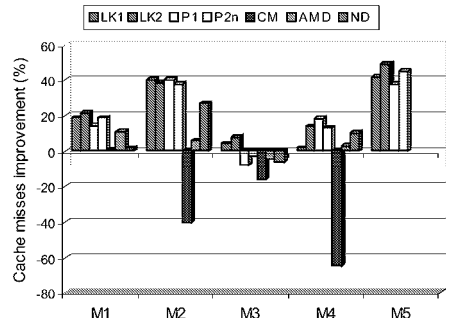
(c) M3 (d) M4



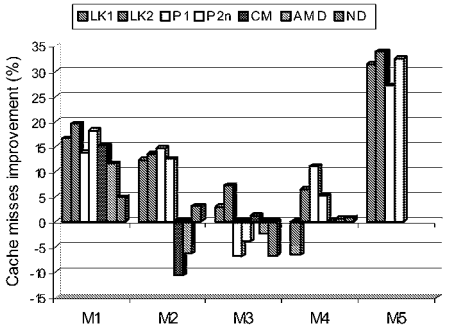
(e) M5

Figure 2. Test set matrices.

As a test set to validate our reorderings we have selected five square matrices from real problems that represent a variety of non-zero patterns [3], as shown in Figure 2. Table 1 summarizes the main characteristics of the matrices. N is the number of rows or columns of the matrix, and N_Z is the number of entries. *M3* presents the same pattern as the *BCSSTK30* matrix from the *Harwell–Boeing collection* [3], but we have selected only a square portion of 23000 rows from the original *BCSSTK30*. Throughout this section the number of cache misses were measured using a simulator that we developed based on trace driven simulation. We simulate two levels of cache, both using Least Recently Used (LRU) as replacement algorithm, considering a two-way associative cache configuration and without prefetching. The first level cache L1 is 32 KBytes and the line size is 4 words. L2 cache has a size of 256 KBytes and the line size is 8 words. We use this cache configuration because is similar to those of some real systems.



(a) L1 Cache



(b) L2 Cache

Figure 3. Comparison between different ordering techniques using three threads.

Next, we show the results achieved by our reorderings and compare them to some of the most representative standard reordering algorithms, the *Cuthill-McKee* (denoted as CM), the *Nested Dissection* (ND) and the *Approximate Minimum Degree* (AMD) algorithms. CM modifies the matrix pattern to obtain a reduced band. ND makes a partition on the adjacency graph searching for a group of nodes, so if we remove this nodes the graph is bisected. In the case of AMD, the objective is to find a symmetric permutation P of the original matrix which reduces the fill-in when the Cholesky factorization is performed. In Figure 3 we show the comparison with the standard algorithms simulating the parallel *SpM×V* and using three threads. The standard algorithms indicated above only can be applied to symmetric matrices. This is the reason because the results obtained through these algorithms for the unsymmetric matrix *M5* are not displayed. The results are expressed in terms of cache miss improvements for the most costly thread with respect to the original matrix. From the set of P_1, P_{1n}, P_2 and P_{2n} solutions, we only apply P_1 and P_{2n} that present the best results (as we explained in section 4). We also dis-

Matrices	Original	LK_1	LK_2
M1	0.59	**0.98**	**0.98**
M2	0.96	**0.98**	**0.99**
M3	0.85	**0.99**	**0.99**
M4	0.57	**0.75**	**0.77**
M5	0.99	**0.99**	**0.99**

Table 2. Load balance using three threads.

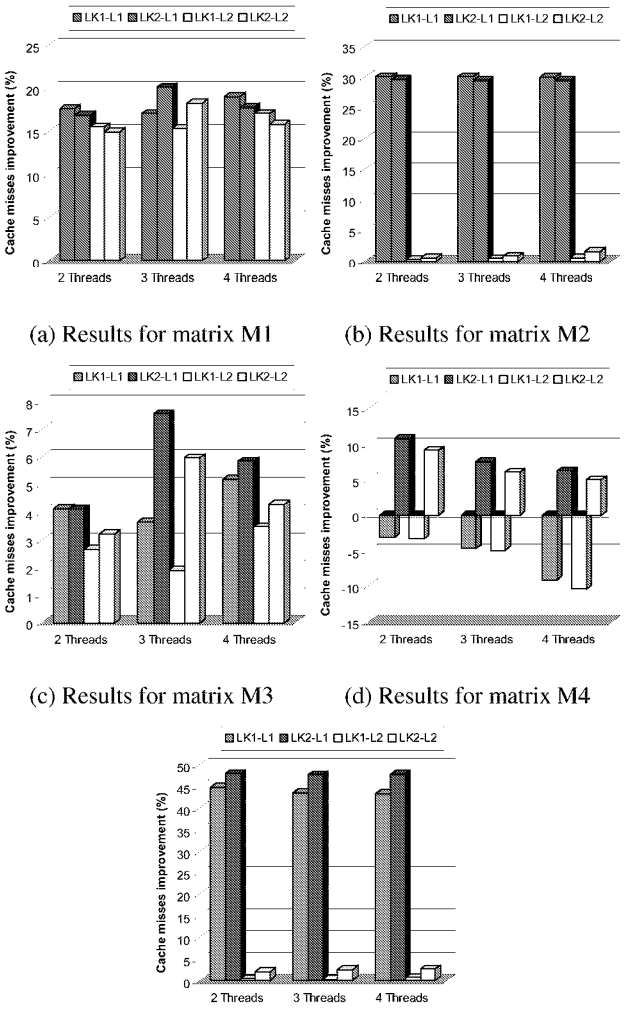
play the results for solutions LK_1 and LK_2. Note that the behaviour with CM, ND and AMD is very irregular. For example, using CM while for the original matrix *M1* the improvement is high in L2 cache, with the other matrices no locality improvement is obtained. Worsening with CM can be up to 64% and 27% for L1 and L2 caches respectively. Results for P_1 and P_{2n} are good for some matrices but very bad for others like the *M3* matrix.

It is specially relevant that LK_1 and LK_2 obtain good results in terms of cache miss improvements for most of the matrices included those highly structured as *M4* (see Figure 2(d)). In particular, LK_2 is the reordering technique that achieved the best and the most regular overall improvements. It obtains for all matrices improvements from 7% to 34% in L2, and from 8% to 48% in L1.

With the objective of evaluating the goodness of our locality improvement technique more exhaustively we also compare the load balance achieved using the matrices reordered using *Prim's algorithm*, the *Nearest Neighbour algorithm* and the *Lin–Kernighan algorithm*. We have observed that the load balance among processors for P_1 and P_{2n} and for the original matrices is very irregular and dependent on the matrix pattern. Nevertheless, with LK_1 and LK_2, the best solutions as explained in the previous paragraph, we achieve a good load balance among threads. In Table 2 we show these results in terms of the ratio between the number of entries assigned to the processor with the lowest load and the number of entries of the most loaded processor. The closer the value of this ratio to 1 the higher the load balance. For obtaining these results the *SpM*×*V* operation was performed using three threads. We can see how LK_1 and LK_2 obtain a better load balance for all the matrices than that obtained using the original matrices, specially for LK_2.

6. Validating the model on SMP

Our model has been validated on a *SGI Origin 200 system* [6]. The system provides four MIPS R10000 that operates at a clock frequency of 225 MHz. The MIPS R10000 is a four-way superscalar RISC CPU. The R10000 uses a two-level cache hierarchy: a 32 KBytes L1 data and instruction caches, and a unified L2 cache of 2 MB. For L1



(a) Results for matrix M1

(b) Results for matrix M2

(c) Results for matrix M3

(d) Results for matrix M4

(e) Results for matrix M5

Figure 4. Comparison between LK ordering techniques on a Origin 200 system for the test matrices.

the line size is 32 bytes, and 128 bytes for L2. Both caches are two-way set associative with LRU replacement policy. The R10000 provide hardware support for counting various types of events, such as cache misses, memory coherence operations, etc...

We show in Figure 4, the results obtained performing *SpM*×*V* with matrices reordered by applying the *LK heuristic* and considering two, three and four threads. LK_1–L1 and LK_2–L1 denote the results for the L1 cache and LK_1–L2 and LK_2–L2 those for the L2 cache. The results show that cache miss improvements are achieved in nearly all cases. In the Origin 200 system, as in the results presented in

the previous section and obtained through the use of a simulator, LK_2 obtains improvements even for the banded matrix *M4*. The best results are obtained for the original matrix *M1* which presents a structure with low clustering on entries over the pattern (see Figure 2(a)) and for which improvements in the number of cache misses are up to 20% and 18% for L1 and L2 respectively. The reason is that there are more opportunities for the *Lin–Kernighan* heuristic to obtain a good reordering if the initial matrix is unstructured. Another important observation is that the improvement percentage is always better in L1 cache. This behaviour is due to the fact that L1 cache is smaller than L2, therefore the number of conflict misses on L1 will be larger than on L2.

7. Conclusions

In this paper a methodology for characterizing and increasing the locality of sparse algebra codes on SMPs based on the definition of parameters associated with the order of data accesses is proposed. We have applied our methodology to the $SpM{\times}V$ which is one of the most important kernels in scientific applications.

The problem of locality improvement has been solved using an analogy to *The Travelling Salesman Problem* through the application of different heuristics such as *Prim's algorithm*, *Nearest Neighbour algorithm* and *Lin–Kernighan algorithm*. The evaluation of locality for guiding the improvement process is based on measurements using two parameters evaluated over the sparse matrix involved in the computations: *entry matches* and *block matches*. On applying this methodology, significant decreases in the number of cache misses for a representative set of matrices from real scientific applications have been obtained, specially using the *Lin–Kernighan* algorithm. We also conclude that our algorithms are competitive when comparing to standard ordering algorithms. Our algorithms also present more stability, i.e., they are more robust to changes in matrix structure. Our methodology was proven first in a simulator and then in a *Origin 200* SMP obtaining similar results in both cases.

As future work we will extend the locality improvement to other sparse algebra codes, such as transposition of a sparse matrix (SpM^T) or the product of a sparse matrix by a dense matrix ($SpM{\times}M$). For these codes we hope a larger benefit achieved by our techniques because they present a higher percentage of irregular accesses. In addition, we plan to combine our methodology with some classic locality improvement techniques such as register blocking or cache blocking.

Acknowledgments

This work was supported by the spanish Ministry of Science and Technology under project TIC2001-3694-C02-01.

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

[3] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection. Technical report, CERFACS, 1992.

[4] 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.

[5] 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.

[6] Silicon Graphics Inc. *Origin 2000 and Onyx2 Performance Tuning and Optimization Guide*.

[7] 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.

[8] 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.

[9] E. Petrank and D. Rawitz. The hardness of cache concious data placement. In *Proc. of th 29th annual ACM symposium on principles of programming languages*, January 2002.

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

[11] Gerhard Reinelt. *The Traveling Salesman. Computational Solutions for TSP applications*, volume 840 of *Lecture Notes in Computer Science*. Springer–Verlag, 1991.

[12] Yousef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1996.

[13] 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.

[14] 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.

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

[16] 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.