Local Enumeration Techniques for Sparse Algorithms *

Gerardo Bandera *Pablo P. Trabado* *Emilio L. Zapata*

Computer Architecture Department. University of Malaga.
Campus of Teatinos, 29071 Malaga, Spain.
e-mail: {bandera, pablo, ezapata}@ac.uma.es
Phone: +34 5 213 27 89 Fax: +34 5 213 27 90

Abstract

Several methods have been proposed in the literature for the local enumeration of dense references for arrays distributed by the CYCLIC(k) data-distribution in High Performance Fortran. These methods deal only with loops without any irregular references. However, existing techniques are not enough when the code includes sparse references. In this work, some methods for enumeration of references are proposed and tested for some linear sparse algebra algorithms. We use the BRS(k) distribution for sparse matrices, which is a generalization of the dense CYCLIC(k) distribution. Efficiency evaluation for the proposed methods has been performed on different processors.

1. Introduction

When programming massively parallel machines using languages such as High Performance Fortran (HPF) the performance of the translated SPMD code is strongly dependent on the scheme used for the generation of local addresses from the data-parallel version of the code.

Plenty of research on this topic has been already done for dense computation and CYCLIC(k) distribution. Most of these methods use tables to store repetitive access patterns [3, 8]. Reeuwijk et al.[6] present local enumeration and storage compression schemes that use global-to-local routines instead of tables. In a more recent work [5] strategies to compute the local set of iterations for complex subscripts are developed. However, none of these methods are able to generate efficient local code for sparse applications as they do not consider the need of resolving indirections.

*The work described in this paper was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03 and the EU Brite-Euram project BE-1564.

This paper presents methods to improve the performance of codes with nested dense and sparse loops. A generalization of the block-cyclic distribution for sparse data-structures, called *generalized-BRS/BCS*, is also presented. This distribution outperforms the traditional CYCLIC(k) in memory allocation and alignment for sparse matrices [1].

The results presented in this paper are focused on the Sparse Matrix-Vector multiplication algorithm. Additional results have been also obtained for the Sparse Matrix-Matrix and the Sparse Matrix-Dense Matrix multiplication algorithms [2]. Our methods have been tested on several different processors.

The paper is organized as follows: Section 2 shows the local representation problem for sparse matrices using the generalized-BRS distribution. Section 3 presents the partitioning of dense and sparse loops using our proposed local enumeration methods. Experimental results are showed in section 4, and our conclusions are in section 5.

2. BRS Local Representation

The generalized-BRS distribution is an extension of the Block Row Scatter distribution (BRS), previously described in [9]. The BRS is a pseudo-regular version of the dense CYCLIC distribution, where the matrix elements are distributed in a round-robin fashion, with only non-null elements being stored, in compressed by rows format (CRS), by their owner processors. The generalized-BRS uses also the CRS format, but distributes the data using the dense block-cyclic (CYCLIC(k)) distribution. From now on, the term BRS will denote exclusively the generalized distribution.

Figure 1.a shows a sparse matrix distributed by BRS(3,3) on a 2x2 processor mesh; underlined elements belong to processor 0. Figure 1.b depicts the local CRS representation of these data.

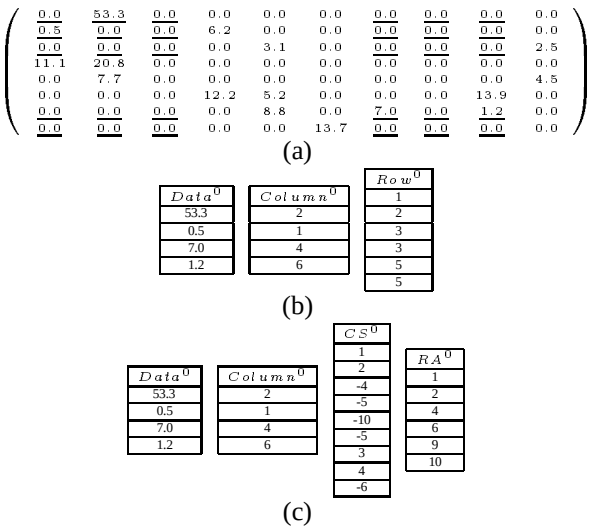

Figure 1. *(a) Sparse Matrix distributed by BRS(3,3) on a 2x2 mesh and two different local representation for Processor (0,0): (b) Local CRS, (c) Vector Quartet.*

Whenever just local data must be accessed, or when the local bounds in the compressed dimension for every processor can be computed at compile-time, the information contained in the CRS format is enough. However, when global accesses must be performed this representation leads to a heavy increase in communications, due to the loss of global information. Let us consider our example matrix of figure 1.a. Given a global row partially owned by processor 0, it can be seen that this representation has lost all information about the owner of non-nulls in that row not local to this processor. As a consequence, evaluation of the ownership of elements in a row will require communications, which will usually have a noticeable negative impact in performance.

If a memory overhead is acceptable, these communications can be avoided by reintroducing this lost global information into the local representation. This can be achieved by replacing the local row vector by two descriptors, which keep track of the owners of data belonging to every row owned by the processor, but not local to it. Let us suppose a processor (myR,myC), which owns a set of global rows given by $(i\%k_1) + k_1 \times (myR + P_1 \times (i/k_1))$, being i a local row, k_1 the number of rows per block and P_1 the number of processor rows in the mesh. Then:

1. *Column Set (CS):* For each data item in this subset of rows, this vector contains either an index pointing to the Data and Column local vectors, or a negative number whose absolute value is the global column number of this item.
2. *Row Auxiliary (RA):* Is a vector of pointers which works exactly as the original Row vector, but pointing CS instead of the Data and Column vectors.

These extended local representation will be called Vector Quartet. Figure 1.c shows this storage scheme for the data belonging to processor 0 in the sparse matrix of figure 1.a. Experimental results in [9] show that the performance increase achieved with these two new vectors widely offsets the memory overhead involved.

3. Address Generation Problem

In this section we deal with the problem of address generation for sparse loops in data-parallel applications. First, the conventional parallelization procedure is described, and an improvement to this scheme is introduced. In subsection 3.1 the use of tables to store access patterns is described. Subsection 3.2 describes the translation mechanism for full range accesses. Finally, in subsection 3.3 some additional optimizations are proposed.

To explain the different techniques employed in the parallelization of sparse loops, we use, without loss of generality, the Sparse Matrix-Vector multiplication code depicted in figure 2. This code contains two nested loops. The first one is a generic dense loop represented by the triplet (low1:high1:str1). The second loop is a generic compressed by rows sparse one, denoted by (ROW(I)+Loff2:ROW(I+1)-Roff2:str2).

```
        .....
        PARAMETER (N=1000,M=1000)
        .....
        REAL DATA(alpha) Y(N), X(M)
        INTEGER COLUMN(alpha), ROW(N+1)
!HPF$   REAL A(N,M), SPARSE(CRS(DATA,COLUMN,ROW)),
      * DYNAMIC
!HPF$   DISTRIBUTE (CYCLIC(k1),CYCLIC(k2)) :: A
!HPF$ ALIGN Y(:) WITH A(:,*)
!HPF$ ALIGN X(:) WITH A(*,:)
        .....
        FORALL 80 I=low1, high1, str1
          FORALL 60 J=ROW(I)+Loff2, ROW(I+1)-Roff2, str2
            Y(I)=Y(I)+DATA(J)*X(COLUMN(J))
60        END FORALL
80      END FORALL
        .....
```

Figure 2. Data-parallel code for MxV.

When the compiler translates this double loop, it uses different translation mechanisms for each loop:

- To partition the dense loop (loop 80), any of the dense mechanisms described in the literature can be used to

parallelize this loop. Our choice is a method based on the one defined by Reeuwijk et al. [6], which is the best performed and does not need any table. Hence, the original loop 80 is transformed into a pair of new loops: the outer loop, which represents the course, and the inner loop, which denotes the offset. Auxiliary functions are necessary to obtain the bounds of each one of these two new loops.

As a consequence of this partition, a global-to-local (g2l) routine must be inserted by the compiler into the inner loop (loop 70), to make possible access to the local index, which in this example is just the value of the local row.

- In the sparse dimension (loop 60), the compiler has to insert two new features in the local code, to guarantee that each processor will only access those elements locally allocated. These features are an IF-statement, and the descriptors CS and RA from the Vector Quartet representation.

The translated code is shown in figure 3.a. This would be the code obtained by any data-parallel compiler accepting sparse loops operations. Let us call **GLOBAL** to this enumeration technique, because the dense index is translated to local just before the sparse loop, where it is used.

An optimization of this code can be obtained by changing the position of the global-to-local routine, which does not need to be into the second dense loop. We will call **LOCAL** to this new method, in which this function is taken out of the inner dense loop (loop 70), and an additional variable containing the local index (in this example, the local row) is introduced. In every iteration this local index will be properly updated (adding the stride of the loop).

With this modification, the number of executions of the g2l routine is reduced. This reduction is specially significant in matrices with a high sparsity rate and when the size of the block of the BRS distribution is large (see section 4).

Neither GLOBAL nor LOCAL include any optimization in the sparse loop. However, to obtain good code performance, a method to efficiently generate the indirect addresses used in the sparse loop must be introduced. In the next subsections mechanisms for this address generation are presented.

3.1. Indices Tables

The tabular methods are based in the allocation of a new structure in the code that stores values which can be used to calculate the accessed indexes. For dense loops, these values can be either the repetitive access pattern which always can be found in a cyclically distributed array, or an explicit enumeration of the indexes. However, for loops accessing to sparse matrices with element compression, the concept of repetitive patterns does not exist, and an explicit enumeration of the accessed indexes must be stored.

It has been verified (see [6]) that the performance of those tabular methods storing repetitive access pattern is not better than the performance of the code generated by GLOBAL. So, only tabular methods explicitly storing the accessed indexes are considered in this paper.

For M_{sp}xV, where the outer dense loop only points the inner sparse loop, two different kind of tables may be used:

1. *Dense Table:* Only stores the references for the dense loop and does not introduce any transformation in the sparse one. The size of this table is the number of accessed elements of the first dimension of the matrix, that is, the number of local rows (nlr). In section 4 we will show that the code performance is not greatly improved whenever the sparsity rate is very small, due to the unsubstancial weight of the dense loops in the total execution time.
2. *Sparse Table:* Stores the references to the accessed data in compressed vectors. With this table, the internal IF-statement of the original translation is removed. The table size is the number of sparse elements accessed for every processor (**sp_index**) plus the number of accessed rows (**sp_bounds**). In average, this method provides an improvement of the performance when compared to the Dense Table method, or to LOCAL.

These two tables can be used simultaneously. This method gives the best performance, if the memory overhead is acceptable. The parallelization of our example using a combination of these tables is presented in figure 3.b.

It is important to note that if we are not able to fill in these tables at compile-time, a preprocessing stage will be needed. However, irregular codes usually require some kind of preliminary stages before the execution itself of the algorithm [7]. These stages can be used to gather the information needed by the table. Moreover, if an iterative algorithm is being carried out, we can fill these tables in the first iteration of the code.

3.2. Full Range Accesses

Frequently, applications with sparse matrices include full range accesses (in short, full accesses) to some dimension of the structures. However, current compiler technology uses the same translation mechanism for both generic and full access loops. But code performance can be improved by careful analysis and translation of this last kind of loops.

Let us consider again the M_{sp}xV code showed in figure 2. We will say that: there is a full access to the dense dimension when low1=1, high=N (# of rows) and str1=1;

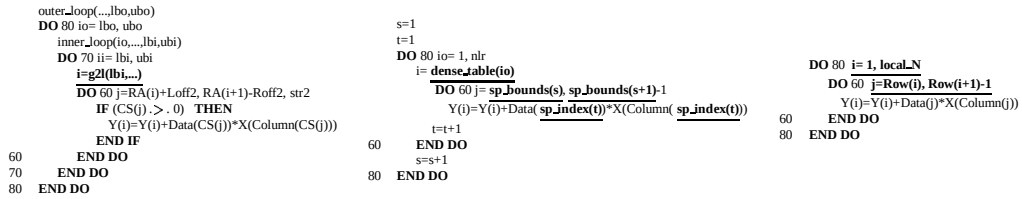

Figure 3. Parallelization of MxV using the different enumeration methods: (a) Global, (b) Tables, (c) Full Accesses.

there is a full access to the sparse dimension when Loff2=0, Roff2=1 and str2=1.

In the parallelization of dense loops, full accesses allow a significant reduction in the execution time because the g2l conversion routine can be omitted at all, and just a single loop is needed in the local code. Noticeable performance improvements can also be achieved for sparse loops with full accesses, because the additional control IF-statement can be removed, and use of the auxiliary vectors RA and CS is not necessary.

Figure 3.c shows the parallelization of the code of figure 2 when both loops contain full accesses.

3.3. Additional Optimizations

The previous enumeration methods can be improved with additional optimizations, which can be groupped in two categories:

- **Loop Interchange:** In some applications it is possible to regroup loops with related induction variables. This permutation can preserve some cache effects for connected data.
- **Indirections Grouping:** Avoid indirection accesses into the sparse loops by storing the values on scalars previously to the beginning of the loop execution. Update these additional variables as close as possible to the dense loop these indirections depend on.

4. Evaluation

Experimental results to validate the enumeration methods described so far have been performed on different processors with quite similar results in all of them. So, we will only present here timings for the DEC Alpha 21064. Logical processor configurations for these experiments were a 2D-array. Selected matrices from the Harwell-Boeing collection [4], together with a set of randomly generated sparse matrices, have been used for benchmarking.

The following tables contain results for the M_{sp}xV algorithm. Two evaluation parameters have been considered: the sparsity rate of the matrix and the size of the block in the BRS distribution. Results for Full Accesses were quite similar to those of the Tables, so they have been omitted by the sake of brevity. In this case, the Loop Interchanging enumeration optimization is not applicable. The different enumeration methods are referred to in the tables using their initials (i.e. DST=Dense & Sparse Tables).

Rate	PE	L	DT	ST	DST	IG
2%	2	1.5%	3.76%	21.4%	18.1%	11.7%
2%	8	4.5%	9.43%	34.7%	41.1%	15.5%
2%	32	12.8%	17.2%	61.1%	66.3%	21.3%
0.5%	2	10.1%	16.1%	14.3%	19.3%	20.9%
0.5%	8	16.6%	27.1%	43.8%	55.2%	24.6%
0.5%	32	21.9%	34.9%	55.2%	71.9%	28.1%
0.1%	2	28.9%	50.2%	35.5%	58.8%	34.6%
0.1%	8	35.3%	61.3%	48.8%	74.7%	39.4%
0.1%	32	40.0%	68.1%	56.3%	83.7%	42.2%

Table 1. Improvement percentage related to GLOBAL for different density rate matrices using a BRS(5,5) distribution.

Table 1 shows the impact of the sparsity rate on the enumeration methods. It can be seen that: for small sparsity rates, the main contribution to the total execution time comes from the sparse loop due to the high number of accesses to the compressed stored data. The methods which optimize the dense loop (L, DT) achieve a very low performance improvement, while the methods that optimize the sparse loop (ST, DST, IG) get a much higher reduction in the execution time. For high sparsity rates, the number of accesses into the sparse loop is very low, and their contribution to the total time is small compared with that of the dense loop. In this case, the methods optimizing only the sparse loop (ST) performs much worse than those optimizing the dense loop or both.

In a BRS(k_1,k_2), both k_1 and k_2 must be taken into account for a good workload balance, but k_2 is not meaningful for performance comparison of the enumeration schemes. Table 2 depicts the influence of k_1 for the Harwell-Boeing

sparse matrix BCSSTK29. This is a 13992x13992 matrix with a density rate of 0.16% (316740 non-nulls).

For small values k_1, there are no significative differences between GLOBAL and LOCAL, due to the small size of the inner dense loop. When k_1 grows, LOCAL greatly outperforms GLOBAL, because the number of executions of g2l decreases. The upper limit for this increase in performance is the time for full range dense access. At the same time, LOCAL scheme also approximates to Dense Table improvements with an increasing value of k_1. This last method obtains a better cache performance for small k_1 values.

k_1	PE	L	DT	ST	DST	IG
1	2	0.16%	33.1%	2.96%	34.2%	9.52%
1	8	0.72%	39.7%	32.5%	71.2%	6.50%
1	32	1.91%	53.7%	28.7%	79.9%	7.43%
5	2	10.4%	18.3%	14.7%	18.3%	21.8%
5	8	15.0%	25.0%	53.6%	62.7%	21.8%
5	32	22.2%	35.7%	57.4%	70.3%	28.8%
25	2	13.2%	13.8%	17.7%	15.1%	25.4%
25	8	18.3%	19.8%	59.8%	61.9%	26.2%
25	32	30.4%	33.1%	69.0%	72.1%	36.7%

Table 2. Improvement percentage related to GLOBAL for BRS(k_1,5) using BCSSTK29.

On the other side, the Sparse Table method is quite insensible to variations of k_1, but compares favourably to LOCAL for an increasing number of processors. In coherency with the results of table 1, Full Access optimizations have a similar behaviour to the table based methods. Finally, the last column of tables 1 and 2 show the slight improvement achieved by the Indirection Grouping optimization.

5. Conclusions

In this paper the problem of the generation of local indexes for SPMD codes from its data-parallel version when both dense and sparse structures are involved, is addressed. The adoption of compact representations for the sparse structures necessitates a massive use of indirections. Existing enumeration methods are initially focused in the computation of dense arrays distributed using CYCLIC(k), and are not able to optimize loops containing sparse references.

Several methods to improve the performance of codes including nested dense and sparse loops are presented. An extension of the scatter distribution for compressed formats, the generalized-BRS/BCS is introduced. When non-local accesses are required, two additional descriptors, RA and CS extend the CRS format. Optimizations to conventional loop parallelization mechanisms are described. These optimizations involve code modifications (carried out at compile-time), or use of additional indices tables (filled in at run-time).

These methods have been tested against varying values of the distribution parameter k and the sparsity rate. The results show that the optimization of dense loops only provides significant improvements for high sparsity rate matrices, while very dense sparse matrices benefit from optimizations in the sparse loops. Performance of methods optimizing the dense loops improve also with the decrease of k (except LOCAL, which is expected). Finally, if memory overheads are admisible, the tabular methods offer the best improvement in performance. Results show also good scalability for all these methods. We can conclude, then, that the quality and performance of SPMD codes could be significantly enhanced by the inclusion of these enumeration schemes into data-parallel compilers.

References

[1] G.Bandera, M.Ujaldon, M.A.Trenas, E.L.Zapata, *The Sparse Cyclic Distribution against its Dense Counterparts*, Proc. IPPS'97, pp. 638-642, Geneve, April 1997.

[2] G.Bandera, *Optimizing Data-Distributions and Enumeration Mechanisms for Applications containing Sparse References*, Internal Report, Computer Architecture Dept, Univ. of Malaga, 1998.

[3] S. Chatterjee, J.Gilbert, F.Long, R.Schreiberg, S.Teng, *Generating Local Addresses and Communication Sets for Data-Parallel Programs*, J. Parallel and Distributed Computing, no. 26, pp. 72-84, 1995.

[4] I.Duff, R.Grimes, J.Lewis, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, Research and Technology Division, Boeing Computer Services, Seattle, WA, 98124-0346, USA, 1992.

[5] J.Ramanujam, S.Dutta, A.Venkatachar, *Code Generation for Complex Subscripts in Data-Parallel Programs*, Proc. LCPC'97, Minneapolis, August 1997.

[6] K.Reeuwijk, W.Denissen, H.Sips, E.Paalvast, *An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems*, IEEE Trans. on Parallel and Distributed Systems, vol. 7, no. 9, pp. 897-914, September 1996.

[7] J.Wu, R.Das, J.Saltz, H.Berryman, *Distributed Memory Compiler for Sparse Problems*, IEEE Trans. on Computers, vol.44, no.6, pp.737-753, June 1995.

[8] A.Thirumalai, J.Ramanujam, *Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices*, Proc. LCPC'95, pp. 191-208, August 1995.

[9] M.Ujaldon, E.L.Zapata, S.Sharma, J.Saltz, *Parallelization Techniques for Sparse Matrix Applications*, J. Parallel and Distributed Computing, vol. 38, no. 2, pp. 256-266, November 1996.