Sparse Matrix Block-Cyclic Redistribution *

Gerardo Bandera *Emilio L. Zapata*

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

Abstract

Run-time support for the CYCLIC(k) redistribution on the SPMD computation model is presently very relevant for the scientific community. This work is focused to the characterization of the sparse matrix redistribution and its associate problematic due to the use of compressed representations. Two main improvements about the buffering and the coordinates calculation modify the original algorithm. Our solutions contain a Collecting, a Communication and Mixing stage with different influence in the execution time depending on the sparsity of the matrix and the number of processors. Experimental results have been carried out on a Cray T3E for real matrices and different redistribution parameters.

1. Introduction

Data distribution plays a very important role in the performance of parallel applications on a distributed-memory machine. In this way, the redistribution is a very important problem for the scientific community. Related work about dynamic data-distribution modification is only focused on dense computation using BLOCK, CYCLIC or BLOCK-CYCLIC distributions. In this way, Gupta et al. [5] obtain expressions to calculate the send and receive processor sets and the data index sets for block and cyclic distributions. Kaushik et al. present in [6] an evaluation model for the calculation of the communication cost. The *strip mining* redistribution is described in [10] where parts of the array are redistributed in sequence. Thakur et al. [9] present an efficient algorithm for a generic redistribution based on GCD and LCM methods. In more recently approaches, Lim et al. [7] study the redistribution of a dense vector based on the notion of the *circulant matrix*, while in [4] an optimization of the message scheduling using coloring graphs is presented.

When sparse matrices are used different data structures and distributions can be used in order to improve the performance. In this paper, we present an analysis of the redistribution algorithm for sparse matrices stored in a compressed format. With this kind of format, the exact placement of a given matrix entry depends on the number of previous non-nulls, without any relationship from the coordinates, as happens with dense vectors. An optimized SPMD algorithm for the redistribution of sparse matrices will be presented in this work. Afterwards, some modifications of this code based on the selection of temporary buffers and coordinates calculation will be described.

Any algorithm which implies data movement contains three different stages: A collection of the information to move to the remaining processors, the data interchange and a final step where the received information is sorted. The use of a compressed format of representation in sparse matrices causes the necessity of storing and sending the coordinates of every data. It can produce an important overhead in the different phases of the algorithm, that will be avoided in our approaches.

The organization of the paper is as follows: Section 2 some important concepts about sparse matrices and data distributions will be presented. The original redistribution code and the additional optimizations are included in section 3. Section 4 contains the experimental part of the work, and finally, the conclusions.

2. Sparse Matrix Background

Many storage formats have been developed to represent sparse matrices in a way that avoids the large amount of memory and computation consumed by the elements of the

*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.

matrix. Solution schemes are often optimized to take advantage of the structure within the matrix. In this work we have considered the compressed by rows format due to its simplicity, although all the methods presented in section 3 can be also used for any other kind of storage. The CRS format represents the sparse matrix using three vectors to store the non-null values (*DA*), its columns number (*CO*), and pointers to the first entry of every row (*RO*). The representation format selection is very important because it defines the data access to the matrix entries.

To improve the performance of a parallel algorithm, its data-structures must be efficiently distributed across processors. In [1] we have demonstrated that regular distributions produce a poor performance for applications containing sparse references. Our alternative is the use of the *Scatter*, which is a generalization of the CYCLIC(k) distribution for sparse matrices. It is called *BRS* when the CRS format is used. In this way, in section 3 we have characterized the redistribution process from a BRS(r) distribution to a BRS(s), for any r and s parameters.

Figure 1 includes an example of a 8×10 matrix with 16 entries where the BRS(2) $\rightarrow$ BRS(5) redistribution on a 2-processors mesh has been carried out. We can observe in 1.b the local submatrix of the processor-0 before the redistribution, while the resulting submatrix is shown in figure 1.c. From this example we can deduce that the sparse redistribution algorithm is not as simple as the dense one. Compressed representations produce a suitable difference on the destination processors data ubication, because the exact position of a given entry is not known until processor receives all the elements from the remaining processors. For example, the data 7.7 with global coordinates *(5,2)* is stored in the fourth cell of the local Data vector on processor-0 for the source distribution CYCLIC(2). The ubication of this data on the resulting local vector for the destination distribution CYCLIC(5) can not be calculated until the owning processor (i.e, processor-0) will know the number of the arriving entries situated on its previous locations. It causes an important overhead in the performance due to the necessity of sorting the receiving entries instead of the direct placement on the destination vector.

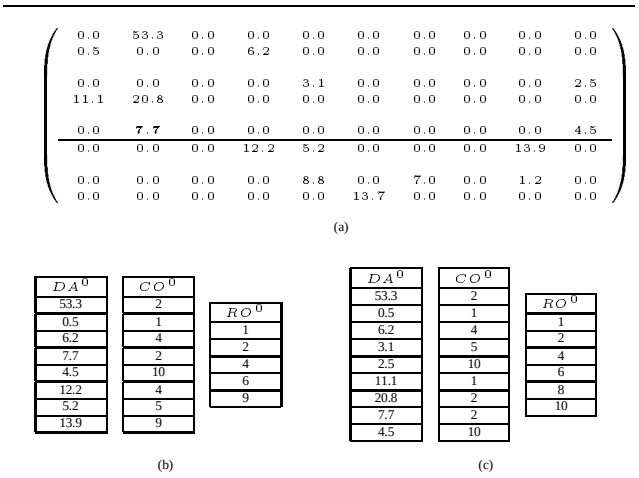

Figure 1. *(a) CYCLIC(2,*) to CYCLIC(5,*) redistribution on a 2-processor mesh; Processor-0 submatrices for the source (b) and target (c) distributions.*

3. Redistribution Algorithms

The parallel code for a vector redistribution needs the calculation of the destination processor to send each local entry and its remote placement. As we have commented previously, while in the dense case these issues can be computed from the size of the matrix, the number of processors and the redistribution parameters, with sparse matrices these calculations can not be done beforehand. The precise placement of every non-null will be computed at runtime, when all the entries have been received. This problem is caused for the compressed representation, in which coordinates are usually replaced by local pointers. At the same time, while in dense computation a vector redistribution only implies the sending of a single value for each local element, a sparse data movement also necessitates the coordinates communication. With this additional information, the receiving processor will be able to reconstruct its new local compressed data-structure.

Algorithms implying data movements across processors have the following three typical phases of computation: first, the indices collection *(COLL)*, where sending information is packing; second, the communication *(COMM)*, for the information interchange; and finally, the placement of the receving data on the new local structure *(MIX)*. In the *COLL* stage, processors store their entries values in a different buffer for every the destination. Additionaly, the coordinates of each sending data will be stored in these temporary buffers, in order to make possible the reconstruction of the new local submatrices in the destination processor. The *COMM* phase is very similar to the dense one, but involving a bigger quantity of sending items, due to the inclusion of the coordinates. Finally, in the *MIX* step, while in the dense redistribution each entry is directly situated in its final location on the resulting vector, sparse matrices require the management of the coordinates information received from the source processor.

The simplest but optimized algorithm for the CYCLIC(r) to CYCLIC(s) redistribution over the rows of the sparse matrix is shown in figure 2. The matrix is called A and it is stored within vectors *Data*,*Column* and *Row* in a CRS format. In this code, loop i iterates over the local rows on every processor, while loop j does it over the compressed elements of every row. To calculate the destination of a given

entry we have used a run-time routine (*DDLY_OWNER*), although a coordinates translation is required in order to obtain the global index from the local one (*DDLY_L2G*). Both run-time routines will be described in subsection 3.2. Although a single vector buffer could be enough for our purposes, for the sake of clarity we have used a different vector for each sending element. An specific counter for every destination processor is also used (*scount*). Similar routines are also required for the MIX stage, where the receiving entries are placed in the final vectors.

```
!!! COLLECTING Phase !!!
    scount[1:Q]=1
    DO i=1, n
        g=DDLY_L2G(i,r,Q,my_pid)
        p=DDLY_OWNER(g,s,Q)
        DO j=Row[i], Row[i+1]-1
            sDATAbuffer[p][scount[p]]=Data[j]
            sCOLUMNbuffer[p][scount[p]]=Column[j]
            sROWbuffer[p][scount[p]]=g
            scount[p]++
        END DO
    END DO

!!! COMMUNICATION Phase !!!
    scount[ ] → rcount[ ]
    sDATAbuffer[ ] → rDATAbuffer[ ]
    sCOLUMNbuffer[ ] → rCOLUMNbuffer[ ]
    sROWbuffer[ ] → rROWbuffer[ ]

!!! MIXING Phase !!!
    count[1:Q]=1
    alpha=1
    Row[0]=1
    DO i=1, n'
        g=DDLY_L2G(i,s,Q,my_pid)
        p=DDLY_OWNER(g,r,Q)
        WHILE (rROWbuffer[p][count[p]]=g .AND. count[p] < rcount[p])
            Data[alpha]=rDATAbuffer[p][count[p]]
            Column[alpha]=rCOLUMNbuffer[p][count[p]]
            count[p]++
            alpha++
        END WHILE
        Row[i+1]=alpha
    END DO
```

Figure 2. *Parallel sparse redistribution code.*

In the next two subsections we have included some modifications of the previous code to enhance the redistribution performance. Additional optimizations will be also presented in section 4.

3.1. Temporary Buffers

The sparsity of a matrix produces a bit of "ignorance" about the memory occupation of each local vector on the compressed representation format. It implies the utilization of an *estimation* procedure. If the parallelization of a given application requires the use of additional storage, this estimation has to be even more exactly.

As we have commented previously, the redistribution algorithm necessitates temporary storage to allocate sending/receiving buffers. To decrement the negative impact on the performance produced by an excessive memory allocation, a careful buffering selection has to be done. In the former redistribution algorithm (figure 2), we have presented a sending buffer which is composed by three vectors per processor, storing every sending entry value and its coordinates. The size of this buffer is three times the number of non-nulls sent to a given processor.

As the sparse format selected produces a particular way of entries visiting (e.g, sorted by rows), the memory allocation of the previous algorithm can be then reduced. The best alternative is the compression of a coordinates vectors in a similar way than the compressed representation of a sparse matrix. Our solution will be called **Histogram buffer** because it stores the number of entries with the same coordinate number in every cell of the compressed vector. With this modification, the memory reservation will be reduced for every destination processor. It will be two times the number of sending non-null plus the size of the histogram, which is about the number of local rows on the destination. This alternative is expected to optimize the performance previously obtained even more for very dense matrices, when the number of elements per row is high.

3.2. Coordinates Calculus Optimization

Both computation of local data coordinates and their inclusion in the sending buffer implies the utilization of some conversion routines in the algorithm. These functions are useful to calculate the destination processors and the global coordinate from a given local one. Although similar routines can be used for the second dimension of the matrix, in this paper we will only show the support for the rows conversion. These routines are included in a run-time library called Data Distribution Layer (DDLY)[8], which also includes other routines useful for irregular applications. Thus, the definition of the routines presented in figure 2 is the following:

- For obtaining the **global index (*g*)** from a given local one (*l*) the following three parameters are needed: *k*: size of the cyclic block, *Q*: the number of processors and *my_pid*: local number of the owner's processor. The definition of the routine $DDLY\_L2G(l, k, Q, my\_pid)$ is the following:

$$g = (l\%k) + (k * (my\_pid + (Q * \frac{l}{k}))) \qquad (1)$$

- To calculate the **owner processor (*p*)** for a given global index (*g*) two parameters are also necessary: the size of the cyclic block *k* and the number of processors *Q*. Then, we can define the $DDLY\_OWNER(g, k, Q)$ routine as:

$$p = (\frac{g + k}{k} - 1)\%Q \qquad (2)$$

Note that in previous formulas $g \in [0, N-1]$, $l \in [0, n-1]$ and $my\_pid \in [0, Q-1]$, where N is the number of matrix rows and n is the number of local rows on the source processor.

The run-time routines previously presented are useful for every necessary local-to-global (l2g) conversion, but the performance of this coordinates processing can be enhanced with the following two improvements:

1. The l2g coordinates translation can be carried out only once every k-consecutive elements on a CYCLIC(k) distributed vector. The remaining k-1 global values are obtained by incrementing this conversion. This optimization is due to the proper definition of the block-cyclic distribution, where k-consecutive entries of the global matrix are stored in the same processor.

2. If the first improvement is used, we can also simplified the routine of the l2g routine of the equation 1 using the following formula:

$$g = k * (my\_pid + (Q * BN)) \tag{3}$$

where BN is number of the local k-size block ($BN \in [0, \frac{n}{K}]$). The above equation reduces the complexity of the *DDLY_L2G* routine, and it is obtaining due to the verification of the following properties:

- $(BN * k)\%k = 0$
- $\frac{BN*k}{k} = BN$

Previous optimizations can be included in the redistribution algorithm. It is expected that the resulting code produces the most important performance enhancement for very sparse matrices, when the number of l2g conversions is high in comparison with the number of iterations of the sparse loop.

4. Experimental Results

To validate the different optimizations described in the previous section we have used some redistribution parameters (*r* and *s*) and different matrices coming from a random matrix generator and from a real collection. The results are very similar for these two kind of matrices, so we will only show the results with real matrices.

We run our codes on a Cray T3E up to 64 processors with SHMEM routines [3] for the communications. Though these routines need explicit barrier synchronizations to avoid cache coherence problems, they produce the best performance on the T3E. The *cc* compiler with the $-O2$ optimization turned on was also used. The evaluation is based in the performance comparison of the original algorithm of the redistribution (figure 2) versus the code obtained from the two enhancements described in subsections 3.1 and 3.2.

The figure 3 shows the total redistribution time (logarithmic scale) for the three algorithms presented in this paper, using two matrices: **PS1** $(N = 3140, \rho = 5.51\%)$ and **B30** $(N = 28924, \rho = 0.12\%)$. We can observe that the performance of the coordinates optimization over the Histogram buffer code (named *Coordinates*) is about 30% of the original. This improvement is maintained for a high number of processors, except for very sparse matrices where the enhancement is over 20%, due to the small percentage of entries per dimension. For very sparse matrices and a large number of processors, while the Histogram Buffer produces a cache performance decreasing, the coordinates optimization achieves important improvements. In the other hand, when the number of entries to communicate per dimension grows (i.e, dense matrices), the buffering optimization benefits very much the execution time, while the coordinates optimization produces a slight delay.

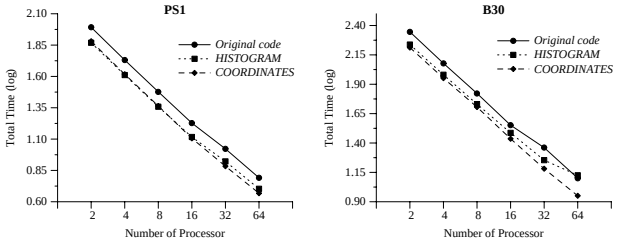

Figure 3. *Redistribution time for the original code versus the Histogram and Coordinates improvements.*

Figure 4 shows the speedup of the best redistribution code with the GWN matrix $(N = 7320, \rho = 0.6\%)$. Jointly with the execution time, this figure also includes the different stages of the redistribution. In spite of the massive data movement presented for this algorithm, a remarkable speedup (32 on 64 processors) is obtained on the code. This result is produced for the influence of the different stages of the code on the total time of the application: while COMM produces an expected value of 10 (on 64 processors), speedups of the COLL and MIX stages are very nice and similar, because they only include indices calculations (approx. 60).

Additional considerations to optimize the parallel code has been taking into account in the evaluation: First, the inclusion of the redistribution algorithm of a sparse loops modification to reduce its indices computation (named *Indirections Grouping* [2]) produces a 10% of decrement to the execution time. In the other hand, when the redistribution parameters has a multiplicity relationship the algo-

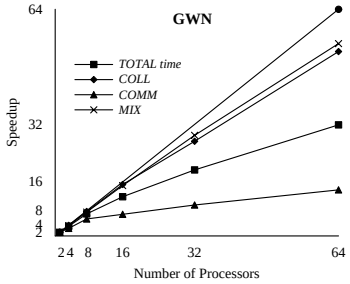

Figure 4. *Speedup of the redistribution stages.*

rithm can be simplified, although it only increment the performance in a 5%. Then, the division of a generic redistribution into two separate stages using a GCD or a LCM scheme [9] does not enhance the performance. Finally, as memory contention (produced when two or more processors are trying to send information to the same destination) usually appears in any massive communication stage, in this work we have included a simple strategy to reduce its impact. In this way, the scientific community is trying to reduce the impact of this situation on the performance by studying the redistribution parameters at compile time, in order to compute previously the number of elements to communicate to each destination. This scheme has not applicability with sparse matrices, because the "ignorance" caused for this kind of arrays promotes the utilization of a costly run-time procedure. The solution used in this work has been the use of a cyclic permutation of the receiving processors for every source.

5. Conclusions

In this article we have presented a characterization of the CYCLIC(k) redistribution of a sparse matrix, which requires a different management than dense structures due to the use of compressed representations. Any parallel application with data movement contains three main stages on the code: an indices *Collecting*, the *Communication* and the receiving data *Mixing*. These stages contain a coordinates processing for each local entry.

An initial but improved redistribution algorithm has been presented. It has been optimized in two different ways: first, using a different temporary storing (called *Histogram* buffer) to reduce the memory allocation for the sending information; and second, reducing the number of local-to-global *coordinates* conversions and their complexity.

Experimental results validate the utility of our redistribution alternatives, where every stage of the code plays a different role on the performance, depending on the number of processors involved and the matrix sparsity. While the collecting and mixing stages produces the most important impact on the performance with a small number of processors and with very dense matrices, the communication time is increased with the number of processors. In the other hand, the use of an optimal buffer produces a better improvement with dense matrices, while the coordinates optimizations achieves an enhancement for very sparse matrices. All the methods present a nice speedup in spite of the redistribution algorithm is based on a massive data movement between processors.

Additional improvements are also obtained by using some optimizations on loops accessing to the compressed dimension and when the redistribution parameters possesses a multiplicity relationship. Memory contention can not be efficiently managed at compile-time, though we have reduced it by using a PUT/GET protocol and a cyclic permutation of the destinations.

References

[1] G. Bandera, M. Ujaldon, M.A. Trenas, E.L. Zapata, *The Sparse Cyclic Distribution against its Dense Counterparts*, Proc. of the 11^{th} Int'l Parallel Processing Symposium, Geneve (Switzerland), pp. 638-642, April 1997.

[2] G. Bandera, P.P. Trabado, E.L. Zapata, *Local Enumeration Techniques for Sparse Algorithms*, Proc. of the 12^{th} Int'l Parallel Processing Symposium, Orlando (Florida), pp. 52-56, March-April 1998.

[3] R. Barriuso, A. Knies, *SHMEM User's Guide for C (Revision 2.2)*, Cray Research Inc., August 1994.

[4] F. Desprez, J. Dongarra, C. Randriamaro, Y. Robert, *Scheduling Block-Cyclic Array Redistribution*, IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 2, February 1998.

[5] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C. Huang, P. Sadayappan, *On the Generation of Efficient Data Communication for Distributed Memory Machines*, Proc. Int'l Computing Symposium, pp.504-513, 1992.

[6] S. Kaushik, C. Huang, R. Johnson, P. Sadayappan, *An approach to Communication-Efficient Data Redistribution*, Proc. Eighth ACM Int'l Conf. Supercomputing, July 1994.

[7] Y.W. Lim, N. Park, V.K. Prasanna, *Efficient Algorithms for Multi-dimensional Block-Cyclic Redistribution of Arrays*, Proc. Int'l Conf. Parallel Processing, pp.234-241, 1997.

[8] G.P. Trabado, O. Plata, E.L. Zapata, *HPF Directives and Run-Time Support for Molecular Dynamics Problems*, Seventh Workshop on Compilers for Parallel Computers, Linkoping (Sweden), pp. 43-56, June-July, 1998.

[9] R. Thakur, A. Choudhary, J. Ramanujam, *Efficient Algorithms for Array Redistribution*, IEEE Trans. on Parallel and Distributed Systems, Vol. 7, No. 6, pp. 587-593, 1996.

[10] A. Wakatani, M. Wolfe, *A New Approach to Array Redistribution: Strip Mining Redistribution*, Proc. Parallel Architectures and Languages Europe (PARLE), pp. 323-335, July 1994.