Iterative and Direct Sparse Solvers on Parallel Computers

R. Asenjo
G. Bandera
G.P. Trabado
O. Plata
E.L. Zapata

September 1996
Technical Report No: UMA-DAC-96/24

Published in:

Euroconference: Supercomputation in Nonlinear and Disordered Systems: Algorithms, Applications and Architectures,
San Lorenzo de El Escorial, Madrid, Spain, September 23-28, 1996, pp. 85-99

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

ITERATIVE AND DIRECT SPARSE SOLVERS ON PARALLEL COMPUTERS

R. ASENJO, G. BANDERA, G.P. TRABADO,
O. PLATA and E.L. ZAPATA
{asenjo,bandera,guille,oscar,ezapata}@ac.uma.es
Department of Computer Architecture, University of Málaga,
C. Tecnológico, PO Box 4114, E-29080 Málaga, Spain

Solving large sparse systems of linear equations is required for a wide range of numerical applications. This paper addresses the main issues raised during the parallelization of iterative and direct solvers for such systems in distributed memory multiprocessors. If no preconditioning is considered, iterative solvers are simple to parallelize, as the most time-consuming computational structures are matrix–vector products. Direct methods are much harder to parallelize, as new nonzero values may appear during computation and pivoting operations are usually accomplished due to numerical stability considerations. Suitable data structures and distributions for sparse solvers are discussed within the framework of a data-parallel environment, and experimentally evaluated and compared with existing solutions.

1 Introduction

Over the last decades, there have been major research efforts in developing efficient parallel numerical codes for distributed-memory multiprocessors, emerging the data-parallel paradigm as one of the most successful programming models. Recently introduced parallel languages, such as Vienna Fortran,[21] Fortran D[12] and High-Performance Fortran (HPF)[14,15] follow this approach.

All these languages had initially focused on regular computations, that is, well-structured codes that can be efficiently parallelized at compile time using simple data (and computation) mappings. However, the current language constructs lead to inneficiencies when they are applied to irregular codes, such as sparse computations, appearing in the majority of scientific and engineering applications (HPF 2.0 is an initial attempt to correct these shortcomings).

A wide range of these applications include the solution of large sparse systems of linear equations. There are two different approaches to solve such systems, direct and iterative methods. In direct methods,[9,13] the system is converted into an equivalent one whose solution is easier to determine by applying a number of elementary row and/or column operations to the coefficient matrix. A different approach is taken in iterative methods,[6,20] where successive approximations to obtain more accurate solutions are carried out.

The convergence rate of an iterative solver depends greatly on the spectrum of the coefficient matrix. Hence, these methods usually involve a second matrix

that transforms the coefficient matrix into one with a more favorable spectrum (preconditioning). Although iterative methods may have such convergence problems, they present a high degree of spatial locality, due to the existence of a sparse matrix-vector product as a basic kernel. This fact makes these algorithms very suitable for parallelization. A main issue in this process is to choose a sparse data distribution scheme which enhances locality and facilitates the location of nonlocal data.

Direct methods, on the other hand, exhibit different problems to those of iterative methods. Factorization of the coefficient matrix may produce new nonzero values (fill-in), so that data structures must consider the inclusion of new elements at runtime. Also, row and/or column permutations of the coefficient matrix are usually accomplished in order to assure numerical stability and reduce fill-in. All these characteristics make direct methods much harder to parallelize than iterative solvers (the exploitable parallelism is typically lower). However, direct methods are free of convergence problems and hence are essential for many cases in which iterative solvers may fail. A major aspect from the parallelization point of view is thus the data structure chosen to store the partitioned data in the local memories of the processors. Standard compressed formats used for representing sparse matrices are not the most suitable schemes to support efficiently the fill-in problem and pivoting operations.

The rest of the paper is organized as follows. In Section 2 we will discuss the computational aspects of two important kernels in iterative and direct schemes, the conjugate gradient (CG) method and the LU factorization. The discussion will be focused on sparse systems. In Section 3, parallel and data locality issues of CG and LU algorithms will be introduced enclosed in a data-parallel framework. The description will highlight the importance in performance terms of choosing a suitable data structure and distribution for the sparse matrices. An experimental performance analysis based on this space of options will be discussed in Section 4, and compared, in some cases, with the capabilities of an existing data-parallel compilation system. Finally, Section 5 presents some concluding remarks.

2 Sparse Matrix Algebra

2.1 Sparse Storage Schemes

A matrix is called sparse if only a relatively small number of its elements are nonzero. A range of schemes have been developed in order to store only the nonzero entries of a sparse matrix, this way obtaining considerable savings in terms of both memory and computation overhead.[6] In our work we have

```
k = 0; x = 0; r = b;
while (ρ_k > ε||b||_2^2 and k < k_max)
    ρ_k = r^T r
    k = k + 1
    if k = 1
        p = r
    else
        β = ρ_{k-1}/ρ_{k-2}
        p = r + βp
    endif
    q = Ap
    α = ρ_{k-1}/p^T q
    x = x + αp
    r = r − αq
endwhile
```

(a)

```
Initialize Ω and Π to the n-by-n identity matrix
Initialize R and C
do k = 1, n − 1
    Find pivot p = (μ, λ) (Markowitz criterion)
    Swap A(k, 1 : n) and A(μ, 1 : n)
    Swap A(1 : n, k) and A(1 : n, λ)
    Swap R(k) and R(μ)
    Swap C(k) and C(λ)
    Swap Ω(k, 1 : n) and Ω(μ, 1 : n)
    Swap Π(1 : n, k) and Π(1 : n, λ)
    A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k)
    do j = k + 1, n
        A(k + 1 : n, j) = A(k + 1 : n, j)
                          −A(k + 1 : n, k)/A(k, j)
    enddo
    Update R and C nonzero counts
enddo
```

(b)

Figure 1: Outline for the (a) Conjugate Gradient algorithm with no preconditioning, and the (b) right-looking LU factorization algorithm

considered the Compressed Row and Column Storages (CRS and CCS). The CRS format represents a sparse matrix `A` as a set of three vectors (`DATA`, `COL` and `ROW`). `DATA` stores the nonzero values of `A`, as they are traversed in a row-wise fashion, `COL` stores the column indices of the elements in `DATA`, and `ROW` stores the locations in `DATA` that start a row. By convention, we store in the position `n+1` of `ROW` (`n` is the number of rows of `A`) the number of nonzero elements of `A` plus one. The CCS format is identical to the CRS format except that the columns of `A` are traversed instead of the rows.

2.2 Conjugate Gradient Algorithm

The Conjugate Gradient (CG) method is the oldest and best known, but effective nonstationary method for the solution of symmetric positive definite systems,[6] $Ax = b$. Such class of methods differ from stationary methods in that the computation involve information that changes at each iteration.

CG belongs to the class of methods with short recurrences, that is, methods that maintain only a very limited number of search direction vectors. The method proceeds by generating vector sequences of iterates, residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. As Fig. 1 (a) shows, in every iteration of the method, two inner products are performed in order to compute update scalars (ρ and α). The

solution vector x is updated by a multiple (α) of the search direction vector p. Correspondingly the residual r is updated by the same multiple but of q, the resulting vector of the sparse matrix-vector product, Ap. Hence we can point out that only one sparse matrix-vector multiplication is required per iteration.

From the computational point of view it is important to analyze the convergence properties of the CG method. It has been observed that this method works well on matrices that are either well conditioned or have just a few distinct eigenvalues. It is thus usual to precondition a linear system so that the coefficient matrix assumes one of these forms. As an example, one of the most important preconditioning strategies involves computing an incomplete Cholesky factorization of the coefficient matrix.

2.3 LU Factorization Algorithm

The LU factorization is used for the conversion of a general system of linear equations to triangular form via Gauss transformations.[13] When it is applied to a n-by-n matrix A produces a couple of n-by-n matrices, L (lower triangular) and U (upper triangular), and the n-by-n permutation matrices Ω and Π, such that $\Omega A \Pi = LU$.

There are different strategies to deal with the sparse LU factorization.[8] The approach considered in this paper corresponds to the right-looking LU generic method. This strategy, also called submatrix-based method, performs a total of n iterations, as shown in Fig. 1 (b). In the k-th iteration a pivot is chosen, a column and a row permutations may be performed so that the pivot occupies the (k, k) position, and, finally, the submatrix defined by the pivot is updated (that is, elements $(k+1:n, k:n)$ of A).

As a consequence of this submatrix updating, fill-ins (that is, new nonzero entries) may occur in matrix A. This fact introduces complexities in the sparse storage structures and increments the memory and computation overheads. Typically, pivot elements are chosen in order to guarantee numerical stability and to maintain sparsity. Markowitz criterion[16] (used in Fig. 1 (b)) is a simple but effective one of such strategies. This heuristic is based in the minimization of the $(R_i - 1)(C_j - 1)$ factor, where R_i (C_j) counts for the number of nonzero elements in the i-th row (j-th column).

3 Parallelization of Sparse Solvers

3.1 Sparse Data Distributions

Current data-parallel languages, such as HPF, Vienna Fortran or Cray Craft, include the most useful and simple schemes for distributing data across the

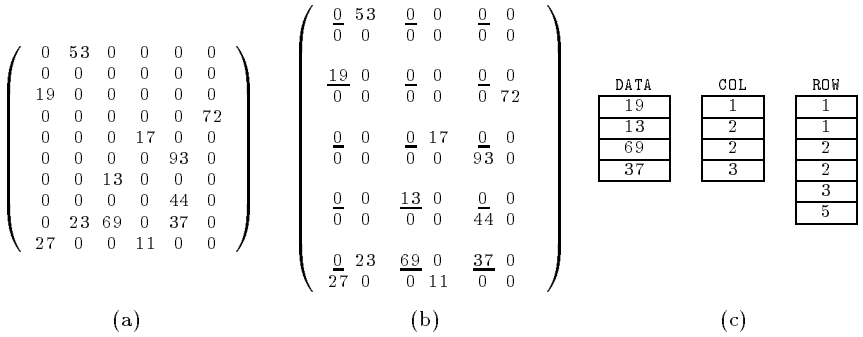


Figure 2: The BRS partitioning (b) of the sparse matrix (a) for a 2×2 processor mesh, where the data elements for processor 0 are underlined and represented compressed (CRS) in (c)

processors, specifically, block, cyclic and a combination of both. All these distributions constitute the basis for mapping dense data structures on the local memories of a distributed-memory multiprocessor.

In the case of sparse matrices, the use of compressed formats for storing them implies the appearing of array indirections in the code (irregular accesses to data). Our approach is to define sparse (pseudo-regular) data distributions as extensions to the classical block and cyclic regular distributions, such as MRD (Multiple Recursive Decomposition) and BRS (Block Row Scatter) or BCS (Block Column Scatter).[18,1]

MRD is a generalization of the Binary Recursive Decomposition [5] and it is intended for algorithms that exhibit strong spatial data locality. BRS (BCS), on the other hand, extends the application domain of the regular cyclic distribution. In this way, BRS (BCS) specifies a cyclic distribution of the sparse matrix represented by its CRS (CCS) compressed format. All the local sparse matrices are represented by using the same format (CRS o CCS) of the original one, as shown in Fig. 2.

An HPF-like description of the BRS distribution of a sparse matrix may be as follows,

```
!HPF$ SPARSE(CRS(DATA,COL,ROW)) :: A(N,N)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO MESH :: A
```

The `SPARSE` directive means that the sparse matrix `A` is actually represented in a CRS format, using the arrays `DATA`, `COL` and `ROW`. The BRS distribution is a cyclic distribution of the compressed representation of a sparse matrix. Stating `CYCLIC` in a `DISTRIBUTE` directive is understood by the compiler as applying a BRS distribution to the `DATA`, `COL` and `ROW` arrays declared

in the `SPARSE` statement. This way we have the benefits of a cyclic data distribution (load balancing and simple and limited communication patterns) applied to a sparse matrix independently of the compressed format used to represent it.

3.2 Parallelization Issues for the CG Method

The basic time-consuming computational structures of iterative methods, such as CG, are inner products, vector updates, matrix–vector products, and preconditioning. The inner products can be easily parallelized. Each processor computes the inner product of two segments of each vector (local inner products). Afterwards, the local inner products have to be sent to other processors in order to be reduced to the required global inner product. Vector updates, on the other hand, are trivially parallelizable, as each processor updates its own (local) segment of the vector.

The matrix–vector products are not so easily parallelized in many cases. It is usual that each processor has only a segment of the vector in its memory. Depending of the bandwidth of the sparse matrix we may need communications for other elements of the vector (a reduction operation may be needed), which may lead to communication problems. Due to the use of compressed formats to store sparse matrices (CRS or CCS, for instance), a crucial potential problem is to get the knowledge about the location of the vector elements (or partial products) that each processor may need to complete the matrix–vector product. It is thus very important to distribute the sparse data across the local memories in such a way that this location of vector elements is no longer a time-consuming operation.

BRS is an example of this kind of data distribution, as the sparse matrix is mapped on the local memories as a dense one. Hence the local matrices are sparse and are independently stored compressed in the corresponding local memory. This way, the elements of the vector can be easily aligned to the columns of the matrix just by using a cyclic distribution (by rows) and a partial replication (by columns) of the vector. This point will be elaborated in the next Section.

Finally, preconditioning is often the most problematic part in a parallel environment. Incomplete decompositions of the coefficient matrix form a popular class of preconditionings that usually introduces recurrence relations that are not easily parallelized. These problems have led to searches for other more simple preconditioners, such as diagonal scaling or polynomial preconditioning. In any case, the preconditioning parallelization problem will not be further considered in this paper.

3.3 Parallelization Issues for the LU Factorization

The right-looking LU solver presents two sources of parallelism we can exploit, as can be derived from the pseudo-code of Fig. 1 (b). The first source of parallelism corresponds to the updating of the reduced matrix (defined as the $(n-k)$-by-$(n-k)$ submatrix of A, such that $k \leq i, j \leq n$, in the k-th iteration), that is, the loops at the end of each iteration. The second one comes from the sparse nature of A. In many cases it is possible to merge several updates of range one in only one update process of multiple range (say, s), by modifying the Markowitz strategy in such a way that we search for a pivot set containing s compatible pivots, instead of only one pivot element (a couple of pivots, A_{ij} and A_{kl}, are said compatibles if $A_{il} = A_{kj} = 0$).

Two major issues have to be considered, fill-in and pivoting. It is known that for, for instance, symmetric positive definite matrices, a sparsity preserving ordering can be selected in advance of the numeric factorization, independent of the particular values of the nonzero entries (that is, only the patterns of the nonzeros matters). Based on this process, called symbolic factorization, the location of all fill elements can be anticipated prior to the numeric factorization, and thus an efficient static data structure can be set up in advance to accommodate them. Unfortunately, in general, we also have to take into account pivoting for numerical stability, which obviously require knowledge of the nonzero values, and would introduce a conflict with the symbolic factorization. Therefore, in order to deal with the general case, to accommodate fill-ins as they occur we need some dynamic data structure for storing the local sparse matrices. Naturally, this introduces significant overheads due to the adjusting of the dynamic structures. Additionally, the data structures should allow an efficient implementation of the pivoting operations, that is, row and column swapping. We have experimented with two-dimensional doubly linked lists, where each item in such structure stores not only the nonzero value of the local matrix entry and its corresponding row and column indices, but also pointers to the previous and next nonzero element in its row and column (four pointers in total).

Doubly linked lists make easy the insertion and delete operations and, hence, we can deal with the fill-in and pivoting problems in a efficient way. But linked lists have also severe drawbacks. The dynamic memory allocation for each new element is time-consuming, and the list traversing even more, as well as they consume a considerable amount of memory per element. Other major problems are the memory fragmentation due allocation/deallocation of items (fill-in), and the spatial data locality loss (cache inneficiency) during traversing rows and columns due to pivoting operations (pivoting does not

move data, only changes pointer references).

From the point of view of an efficient computation, packed vectors (for instance, CRS or CCS compressed formats) are a much more compact data structure for storing local sparse matrices than linked lists, and allow faster accesses by rows or columns to the matrix elements. But, regrettably, they do not support the complex pivoting operations and fill-ins that occur in the LU factorization. In order to deal with such problems we have to resort to auxiliary buffers and implement garbage collection operations, both with significant increases in terms of memory and computation overheads.

4 Experimental Evaluation

To demonstrate the need for defining special data storage structures and distributions to support sparse applications efficiently, we carried out a set of evaluation experiments comparing the current data-parallel software technology (specifically, block and cyclic regular data distributions) with those described previously.

All the experiments were conducted in a Cray T3D multiprocessor using the Craft data-parallel compilation system,[17], the SHMEM native shared-memory library,[3,4] and the optimized PVM message-passing library. In order to take into account the effects of the dimensions, sparsity rate and nonzero patterns in the input sparse matrix, we chose for the experiments a number of sparse matrices taken from the Harwell-Boeing collection,[10] such as shown in Table 1. BCSSTK29 and BCSSTK30 are large and very sparse matrices used in eigenvalue problems while LNS3937 is also a very sparse but quite small matrix taken from compressible fluid flow codes. Less sparse (and small) matrices are JPWH991, used in circuit physics modeling, and SHERMAN1, appearing in oil reservoir modeling codes. Finally, much less sparse matrices are PSMIGR1, containing population migration data, SHERMAN2, for oil reservoir modeling, and STEAM2, used in oil reservoir simulation.

4.1 Parallel Implementations Comparison for the CG Method

The most time-consuming computation included in the (no preconditioned) CG method corresponds to the sparse matrix–vector product executed in each iteration. This is also the most complex computation structure in CG in terms of parallelization. A major issue at this point is thus the data distribution scheme used to map the sparse matrix elements to the processor local memories, and the scheme adopted to align the vector elements to the matrix elements.

We made three parallel implementations for the CG method stressing these

Matrix	*Size*	*Nonzeros*	*Sparsity Rate*	*Features*
BCSSTK29	13992×13992	316740	0.16%	Large and sparse
BCSSTK30	28924×28924	1036208	0.12%	Very large and sparse
JPWH991	991×991	6027	0.61%	Small and quite sparse
LNS3937	3937×3937	25407	0.16%	Quite small and sparse
PSMIGR1	3140×3140	543162	5.51%	Quite small and dense
SHERMAN1	1000×1000	3750	0.37%	Small and quite sparse
SHERMAN2	1080×1080	23094	1.98%	Small and quite sparse
STEAM2	600×600	13760	3.82%	Small and quite dense

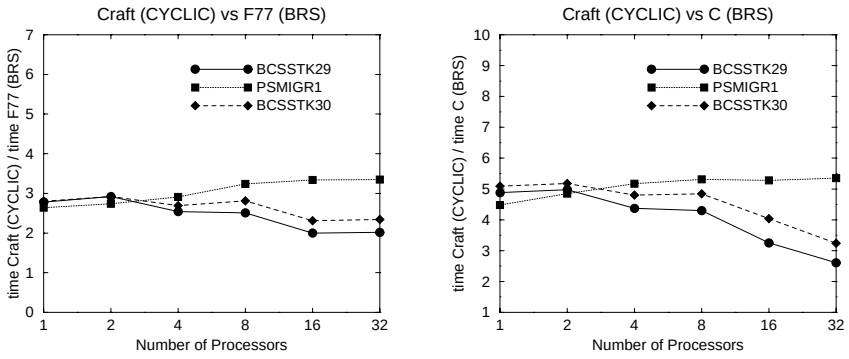
Table 1: Benchmark sparse matrices taken from the Harwell-Boeing suite

issues. The first version was coded using the Cray T3D Craft Fortran data-parallel environment. The sparse coefficient matrix (A in Fig. 1 (a)) is represented using the CRS format (that is, three vectors). All the three vectors of the compressed format as well as the rest of the vectors of the CG algorithm are mapped on the processors using the Craft cyclic data distribution scheme (`cdir$ shared V(:block(1))` for a generic V vector).

The other two parallel versions were written in standard C and Fortran77, respectively, using the BRS data distribution scheme for the sparse coefficient matrix. The rest of the (dense) vectors of the algorithm are distributed using a regular cyclic scheme. In order to minimize communication overhead we must align the p vector with the columns of A, and the q vector with its rows (see Fig. 1 (a)). This is accomplished arranging the set of processors as a mesh and distributing p cyclicly by columns of processors (and replicated across rows of processors), and q also cyclicly but by rows of processors (and replicated across columns of processors). This way all the partial matrix–vector products are completely local, and the needed reduction operations (to obtain global matrix–vector products) imply regular communications. In both cases, the Cray T3D SHMEM routines were used for efficient communications among the processors.

The Craft programming environment allows the programmer to distribute data across the processors using block and cyclic schemes. Here we report results derived from using only the second scheme due to two reasons. First, we are mainly interested in evaluating our BRS data distribution scheme, which is based on the cyclic scheme. Second, we have made implementations of the CG method using both mapping schemes, block and cyclic. For all tested sparse matrices (taken from Table 1) cyclic distribution yields a significant better performance than block.

A two–by–two comparison of the three above parallel codes for two representative sparse matrices is shown in Fig. 3. These results show that the Craft



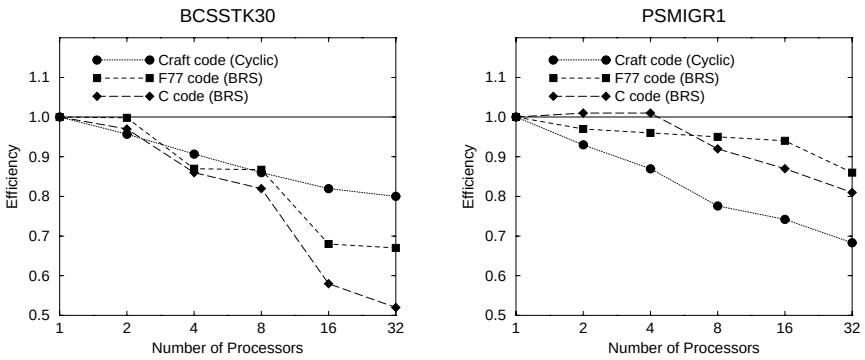
(a) (b)

Figure 3: Improvement over Craft (CYCLIC distribution) for the manual (a) Fortran and (b) C parallel codes of the CG algorithm using BRS distribution

(cyclic) code is more than two times slower than the Fortran77 (BRS) code, and more than four times slower than the C (BRS) code. In addition, the much better performance of the C compiler against the Fortran77 compiler on the Cray T3D is also stand out. The main reason for the low performance of the Craft code is that the compiler does not use the knowledge that the three vectors for the CRS format actually represent a sparse matrix. Hence there is no alignment between A and p and q, which produces complex communication patterns (in order to locate nonlocal data).

Fig. 4 presents, on the other hand, the parallel efficiency for the three codes using the same sparse matrices. From this figure we can emphasize the scalability properties of the codes. Observe that the fastest code (C program) is also the less scalable program (for BCSSTK30). The Craft compiler obtains the best scalable program due to its good behaviour during compilation, for instance, using intrinsic BLAS routines [7] for vector operations.

Finally, Fig. 5 shows the execution time per iteration of the three parallel codes, where the fraction of time consumed in the sparse matrix–vector product, and the other fraction of time consumed in the rest of (dense) operations are highlighted. It is important to note that the CG algorithm spend most of its time (more than 80 or 90%) executing matrix–vector products. So, the scheme used to distribute the sparse matrix across the processors, and the correct alignment of the dense vectors, constitute the key to obtain high-performance implementations of the CG method, as the above experimental results show.



(a) (b)

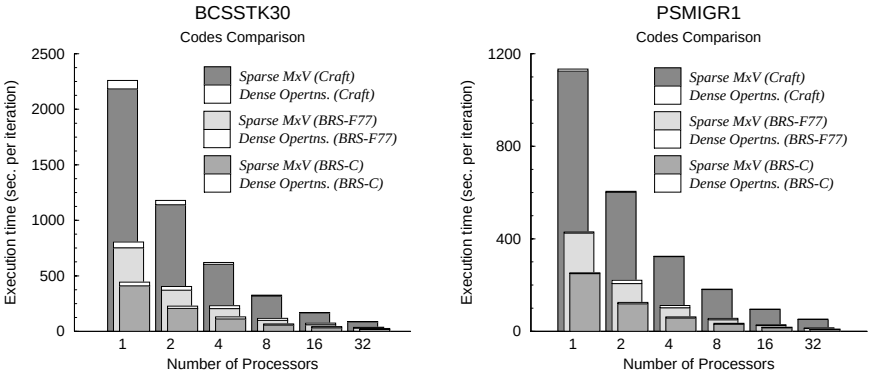
Figure 4: Efficiency for the Craft (CYCLIC distribution), Fortran and C (BRS distribution) parallel CG codes

4.2 Parallel Implementations Comparison for the LU Factorization

Unlike the CG method, the parallelization of the right-looking LU factorization must deal with the fill-in and pivoting problems, both difficult to solve in an efficient way. Now there are two important issues to be considered, the distribution of the sparse matrix elements across the processors and the data structures used to store the local sparse data.

In order to study the above computational aspects, two parallel implementations of the LU factorization algorithm were made, one of them based on doubly linked lists, and the other one based directly on a standard compressed format (also called packed vectors). Both codes, as in the CG solver, consider the processors arranged as a rectangular mesh. The linked list based version was coded in C and PVM routines were used for message-passing. In order to reduce the communication overhead, the low latency communication functions `pvm_fastsend` and `pvm_fastrecv` (non-standard PVM routines) were used for messages of length less than 256 bytes. The packed vectors based code, on the other hand, was coded in Fortran77 plus Cray T3D SHMEM routines to access nonlocal data.

The first parallel version maps the coefficient matrix on the processors using a BCS distribution, storing the corresponding sparse local matrices on a doubly linked list data structure.[2] Fig. 6 (a) reproduces the parallel execution times obtained for different mesh sizes and sparse matrices taken from Table 1. With the use of linked list to store the local matrix elements, the appearing of fill-ins and the pivoting operations (row and column swapping) are handled



(a) (b)

Figure 5: Complete iteration execution time for the Craft (CYCLIC distribution), Fortran and C (BRS distribution) parallel CG codes, using the (a) BCSSTK30 (quite sparse) and the (b) PSMIGR1 (rather dense) Harwell-Boeing matrices

easily in an efficient way.

In order to save memory overhead and increase cache performance we can replace linked lists by packed vectors, as the second parallel code do. Now the coefficient matrix is distributed following the BCS scheme and the sparse local matrices are stored in the implicit CCS format. In the right-looking LU fill-in affects to the whole reduced matrix, which may imply a major data movement process. To deal with this phenomenon we can resort to a garbage collection operation. We are also forced to minimize the number of row and column permutations during the factorization stage. To accomplish this, the stage where the appropiate row and column permutations (required for the selection of the pivot elements) are chosen is carried out separately from the factorization stage itself (updating of the reduced matrix). For the first of these stages the MA50AD routine[11] (included in the MA48 software package) is used in its original form (that is, it is executed not parallelized in only one processor before the factorization stage).

As the sparsity rate of the matrix decreases during factorization, a switch to a dense LU is advantageous at some point. The iteration beyond which a switch to a dense code takes place is decided in the pivoting stage. This dense code is based on Level 2 (or 3 if a block cyclic distribution is used) BLAS, and includes numerical partial pivoting in order to assure numerical stability. Once the sparse computations are finalized the reduced submatrix is scattered to a dense array (that is, the dense submatrix appers distributed in a regular cyclic manner). This way, the overhead of the switch process is negligible.

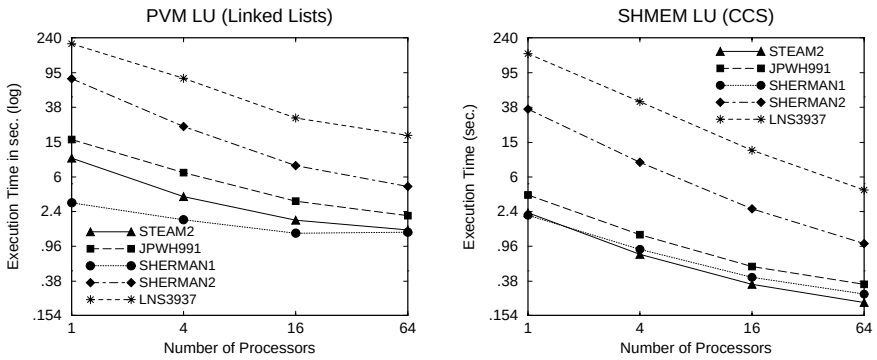


Figure 6: Parallel sparse LU execution times using (a) doubly linked lists and Cray T3D PVM, and (b) CCS distribution and Cray T3D SHMEM

The parallel execution times for the presently described implementation of the BCS-based LU factorization is showed in Fig. 6 (b). We can observe that this implementation is more scalable than the PVM one.

5 Conclusions

Current data-parallel language and compiler technology is not able to obtain outstanding performance from many scientific and engineering applications because they contain irregular code, mainly characterized by the presence of array indirections for data access.

Through this paper we have tried to show that it is needed and possible to design data distributions and structures specially suitable for parallelizing efficiently such programs. In particular, we have analyzed these two aspects in the framework of solving systems of linear equations using iterative and direct methods. We have showed that the use of pseudo-regular data distributions, such as BRS or BCS, allows us both to exploit data locality and to localize easily nonlocal data. This way communication overhead is limited and performance is improved.

In addition, we have discussed techniques to handle in an efficient way the fill-in and pivoting problems, appearing in some direct solvers. They are very hard to deal with because the use of compressed formats to store sparse matrices. We have implemented parallel versions of codes with such problems, such as LU, using dynamic data structures for storing the local sparse matrices, as doubly linked lists. Also the same problems were addressed considering

directly compressed formats (packed vectors) in order to save memory and to improve cache efficiency.

Current work on these techniques is aimed to implement these optimizations on a data-parallel compiler. Specifically, we are developing a run-time library, called DDLY [19] (Data Distribution Layer), with a set of routines to manage all this functionality, and to be called from the output code generated for such a compiler.

Acknowledgments

We gratefully thank Ian Duff and all members in the parallel algorithm team at CERFACS, Toulouse (France), for their kindly help and collaboration. We also thank the Ecole Polytechnique Federale de Lausanne, Switzerland, and the Edinburgh Parallel Computing Centre, UK, for giving us access to the Cray T3D multiprocessor.

This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03-01 and by the Human Capital and Mobility programme of the European Union under project ERB4050P1921660 and by the Training and Research on Advanced Computing Systems (TRACS) at the Edinburgh Parallel Computing Centre (EPCC).

References

1. R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata (1995), "Sparse Block and Cyclic Data Distributions for Matrix Computations", in *High Performance Computing: Technology, Methods and Applications*, J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik, eds., Elsevier Science B.V., The Netherlands, pp. 359–377.
2. R. Asenjo and E.L. Zapata (1995), "Sparse LU Factorization on the Cray T3D", *Int'l. Symp. on High Performance Computing and Networking (HPCN)*, Milan, Italy, pp. 690–696 (Springer-Verlag, LNCS 919).
3. R. Barriuso and A. Knies (1994), "SHMEM User's Guide for Fortran, Rev. 2.2", Cray Research, Inc.
4. R. Barriuso and A. Knies (1994), "SHMEM User's Guide for C, Rev. 2.2", Cray Research, Inc.
5. M.J. Berger and S.H. Bokhari (1987), "A Partitioning Strategy for NonUniform Problems on Multiprocessors", *IEEE Trans. on Computers*, 36 (5), 570–580.
6. R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst (1994), *Templates for*

the Solution of Linear Systems: Building Blocks for Iterative Methods, Siam Press.

7. "Basic Linear Algebra Subprograms: A Quick Reference Guide", Numerical Algorithms Gr. Ltd., Oak Ridge National Lab., Univ. of Tennessee.
8. J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst (1991), *Solving Linear Systems on Vector and Shared Memory Computers*, Siam Press.
9. I.S. Duff, A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, NY.
10. I.S. Duff, R.G. Grimes and J.G. Lewis (1992), "Users' Guide for the Harwell-Boeing Sparse Matrix Collection", Research and Technology Div., Boeing Computer Services, Seattle, WA.
11. I.S. Duff and J.K. Reid (1996), "The Design of MA48, A Code for the Direct Solution of Sparse Unsymmetric Linear Systems of Equations", *ACM Trans. on Mathematical Software*, 22(2), 187–226.
12. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng and M. Wu (1990), "Fortran D Language Specification", Tech. Report COMP TR90-141, Computer Science Dept., Rice University, TX.
13. G.H. Golub and C.F. van Loan (1991), *Matrix Computations*, The Johns Hopkins University Press, MD.
14. High Performance Fortran Forum (1993), "High Performance Fortran Language Specification, Ver. 1.0", *Scientific Programming*, 2 (1–2), 1–170.
15. High Performance Fortran Forum (1996), "High Performance Fortran Language Specification, Ver. 2.0".
16. H.M. Markowitz (1957), "The Elimination Form of the Inverse and its Application to Linear Programming", *Management Science*, 3, 255–269.
17. D.M. Pase, T. MacDonald and A. Meltzer (1994), "The CRAFT Fortran Programming Model", *Scientific Programming*, 3, 227–253.
18. L.F. Romero and E.L. Zapata (1995), "Data Distributions for Sparse Matrix Vector Multiplication", *Parallel Computing*, 21, 583–605.
19. G.P. Trabado and E.L. Zapata (1995), "Exploiting Locality on Parallel Sparse Matrix Computations", *3rd. Euromicro Worksh. on Parallel and Distributed Processing*, San Remo, Italy, pp. 2–9.
20. D.M. Young (1971), *Iterative Solution of Large Linear Systems*, Academic Press, NY.
21. H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald (1992), "Vienna Fortran – A Language Specification", Tech. Report ACPC-TR92-4, Austrian Center for Parallel Computation, Univ. of Vienna, Austria.