Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-in

J. Tourino
R. Doallo
R. Asenjo
O. Plata
E.L. Zapata

December 1996
Technical Report No: UMA-DAC-96/30

Published in:

Sixth Workshop on Compilers for Parallel Computers
Aachen, Germany, December 11-13, 1996, pp. 151-168

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

Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In [*]

J. Touriño R. Doallo

Dept. Electronics and Systems
University of La Coruña, Spain
email: {juan,doallo}@udc.es

R. Asenjo O. Plata E.L. Zapata

Dept. Computer Architecture
University of Málaga, Spain
email: {asenjo,oscar,ezapata}@ac.uma.es

Abstract

This paper addresses the problem of the parallelization of sparse direct methods for the solution of linear systems in distributed memory multiprocessors. Sparse direct solvers include pivoting operations and suffer from fill-in, problems that turn the efficient parallelization into a challenging task. We present some data structures to store the sparse matrices that permit to deal in a efficient way with both problems. These data structures have been evaluated on a Cray T3D, implementing, in particular, LU and QR factorizations as examples of direct solvers. Any of the data representations considered enforces the handling of indirections for data accesses, pointer referencing and dynamic data creation. All of these elements go beyond current data-parallel compilation technology. Our solution is to propose new extensions to HPF that permit to deal with these codes, and to support part of the new capabilities on a runtime library at the compiler level.

1 Introduction

The solution of systems of linear equations, $Ax = b$, where A is a large sparse matrix, plays a basic role in many fields of science and engineering. There are two different approaches to solve such systems, direct and iterative methods. In direct methods [10][15][18], 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 matrix A. A different approach is taken in iterative methods [4][28][18], where the number of operations required is not known in advance.

In this paper we will focus on two of the most important direct methods, LU and QR factorizations [15]. LU factorization is used for the conversion of a general system of linear equations to triangular form via Gauss transformations. The QR decomposition has various other applications in linear algebra, such as solving least squares problems, eigenvalue

[*]The work described in this paper 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)

problems, coordinate transformations and projections problems. All this kind of computations appears in many scientific areas, such as fluid dynamics, structural analysis, circuit simulation, device simulation and quantum chemistry, among many others.

Over the last decades, there have been major research efforts in developing efficient parallel numerical codes, emerging the data-parallel paradigm as one of the most successful programming models to reach the above objective. As a result, during the last few years, a number of high-level data-parallel languages have been designed, such as Vienna-Fortran [29], Fortran D [14], High-Performance Fortran (HPF) [16] and Craft [20].

All these languages had initially focused in regular computations, that is, well-structured codes that can be efficiently parallelized at compile-time using simple data distributions. However, the current constructs included in these languages lead to a low efficiency when they are applied to irregular codes, such as sparse computations, appearing in the majority of real scientific and engineering applications. In order to help solving this problem we have developed and extensively tested a number of pseudo-regular data distributions, designed as natural extensions of the regular data distributions [1] [2] [3] [7] [22] [25] [26] [27]. The aim of these distributions is their simplicity to be incorporated to a data-parallel language and be used by a programmer, together with their effectiveness to obtain high efficiencies from the parallelization of irregular codes. Other important aspect that may influence the parallel efficiency is how the partitioned data is stored in the local memories of the processors. In this paper we will discuss all these related issues in the scene of sparse direct methods, specifically LU and QR decompositions. Special attention will be given to the efficient solution of the fill-in problem and the pivoting operation.

The rest of the paper is organized as follows. In Section 2 we discuss the sparse direct methods, in particular the LU and QR factorizations. In Section 3 we describe and discuss the data structures we have designed to implement efficiently the factorization codes. The pseudo-regular data distributions we propose, specifically for the efficient parallelization of sparse direct methods are introduced in Section 4. The parallelization strategy of such direct methods, LU and QR factorizations, are presented in Section 5, together with some experimental results comparing different data structures and distributions. Based on the experimental results obtained, we propose in Section 6 new extensions to the HPF data-parallel language for solving efficiently the main issues which appear during the computation of sparse direct methods. Finally, Section 7 presents concluding remarks.

2 Sparse Direct Methods

2.1 QR Factorization

The QR factorization of a m-by-n matrix A is given by $A = QR$, where Q is m-by-n orthogonal matrix (that is $Q^T = Q^{-1}$) and R is a n-by-n upper triangular matrix. The QR computation can be arranged in several ways, such as methods based on Householder reflections and Givens rotations. The Gram-Schmidt orthogonalization process and, particularly, the more numerical stable variant called Modified Gram-Schmidt (MGS) is the method considered here.

The MGS algorithm is a rearrangement of the Classical Gram-Schmidt algorithm with better numerical properties. Figure 1 shows an in-place algorithm that includes column pivoting in order to deal with rank deficiency problems (rank$(A) < n$) and to provide numerical stability. Basically, the MGS algorithm is an iterative procedure of up to n iterations. In each iteration k a pivot column is identified and swapped for the current column k in both

```
rank = $n$;
do $j = 1, n$
    norm($j$) = $\sum_{i=1}^{m} A(i,j) * A(i,j)$
enddo
do $k = 1, n$
    Find $p$ with $k \leq p \leq n$ so norm($p$) = $\max_{k \leq j \leq n}$ norm($j$)
    if (norm($p$) $< \phi$)
        rank = $k - 1$; break
    else
        swap $A(1:m,k)$ and $A(1:m,p)$
        swap $R(1:n,k)$ and $R(1:n,p)$
        swap norm($k$) and norm($p$)
    endif
    $R(k,k) = \sqrt{\mathtt{norm}(k)}$
    $A(1:m,k) = A(1:m,k)/R(k,k)$
    do $j = k+1, n$
        $R(k,j) = \sum_{i=1}^{m} A(i,k)A(i,j)$
        norm($j$) = norm($j$) $- R(k,j)R(k,j)$
        $A(1:m,j) = A(1:m,j) - A(1:m,k)R(k,j)$
    enddo
enddo
```

Figure 1: Modified Gram-Schmidt (MGS) algorithm

matrices Q (that is A) and R. Afterwards all the j columns, $k \leq j \leq n$, of Q are updated and the k row of R is computed. Once the algorithm has finished, what we really obtain is a $A\Pi = QR$ factorization due to the pivoting operation carried out.

Considering that A is a sparse matrix, we have taken special actions during the pivoting operation in order to reduce the fill-in problem in the MGS algorithm and to ensure numerical stability. A simple strategy is based on the selection of a column with the maximum norm (the one considered in Figure 1), but we have also experimented with a more elaborated pivoting criterion, where columns with few nonzero elements are the only eligible columns to be the pivot [7]. This way the fill-in in R and Q is reduced.

Figure 2 presents, in a graphical way, the data accesses (dependences) and flows for the three main operations in the MGS code: pivoting (column swapping), columns (Q) and row (R) updating and fill-in. An efficient parallel implementation of the first two operations requires fast accesses to data by columns. This fact strongly determines the data structures we should use for storing the sparse matrices, as well as the method chosen to distribute these matrices among the local memories of the multiprocessor. Moreover we should consider some dynamic mechanism at the data structure level in order to deal with the fill-in efficiently. All these issues will be discussed in the next Section.

2.2 LU Factorization

The LU factorization of 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 LU factorization of generic

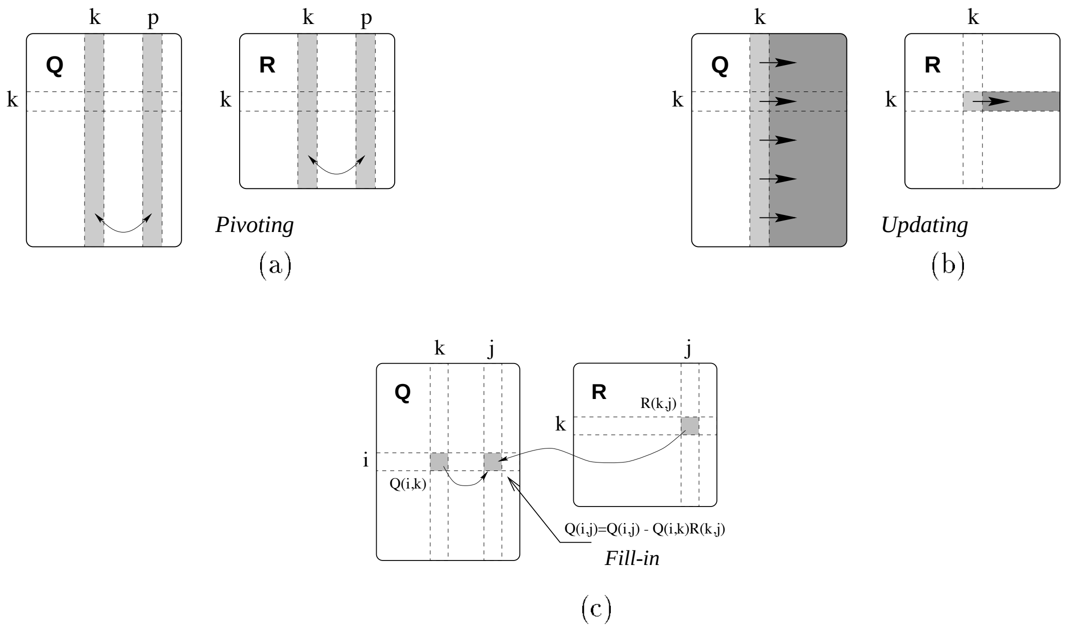


Figure 2: Pivot (a), update (b) and fill-in operations (c) of matrices Q and R in MGS

sparse matrices [8]. Some of the options are based on supernodal or multifrontal approaches, in which arithmetic is performed on dense submatrices [9] (level 2 or 3 BLAS can be used). A third approach, the one considered in this paper, may consist in selecting some generic methods, such as left-looking or right-looking LU.

The left-looking strategy, also known as column-based (row-based) method and implemented for instance in the MA48 routine [11], updates in the k-th iteration the k-th column (of L and U) starting from the $k-1$ previously updated columns. This is accomplished by two different steps for each iteration, a symbolic factorization and a numerical factorization. In the first step, the nonzero structure of the current column is predicted. The second step performs the arithmetic operations strictly needed, using the information gathered by the symbolic factorization. The right-looking LU, also called submatrix-based method, performs a total of n iterations. 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 $(k+1:n, k:n)$).

Any of the above methods can complete the resolution of the linear system of equations following four stages: *reordering*, *analyze*, *factorize* and *solve*. The first step realigns the matrix with the aim of reducing the complexity of the subsequent processing stages. The *analyze* stage chooses the appropriate row and column permutations (Ω and Π), required for the selection of the pivot elements. These pivots must be chosen such that the sparsity rate is preserved (applying, for example, the Markowitz criterion [19]), and the numerical stability is guaranteed (choosing those pivots greater than a certain threshold value). Afterwards, in the *factorize* stage, the factorization $\Omega A \Pi = LU$ is performed. This may be the most consuming-time stage, since it performs the update operations for floating-point numbers. Finally, in the *solve* stage, the above factorization is use to solve the equation $Ax = b$ or $A^T x = b$.

From a parallel implementation point of view, the left-looking organization does not let us to exploit as much parallelism as in the right-looking strategy [2], as in the first one a column is

```
Initialize $\Omega$ and $\Pi$ to the $n$-by-$n$ identity matrix
Initialize $R$ and $C$
do $k = 1, n-1$
    Find pivot $p = (\mu, \lambda)$ (Markowitz criterion)
    Swap $A(k, 1:n)$ and $A(\mu, 1:n)$
    Swap $A(1:n, k)$ and $A(1:n, \lambda)$
    Swap $R(k)$ and $R(\mu)$
    Swap $C(k)$ and $C(\lambda)$
    Swap $\Omega(k, 1:n)$ and $\Omega(\mu, 1:n)$
    Swap $\Pi(1:n, k)$ and $\Pi(1:n, \lambda)$
    $A(k+1:n, k) = A(k+1:n, k)/A(k,k)$
    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
```

Figure 3: Outline for the right-looking LU factorization algorithm, where the *analyze* and *factorize* stages appear joined together

updated in each iteration, while in the second one a complete submatrix is updated. Moreover, an additional overhead arises from the parallelization of the depth-first search in the symbolic factorization.

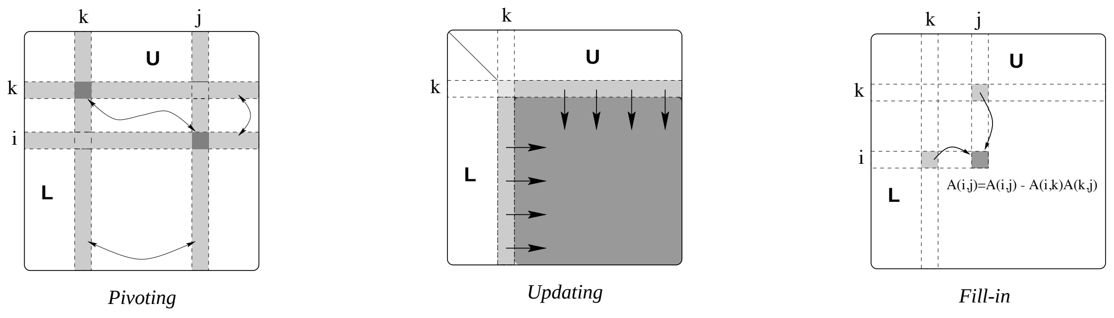
The submatrix-based (right-looking) approach presents two sources of parallelism we can exploit, as can be derived from the the pseudo-code of Figure 3. The algorithm performs a number of iterations, each of them involving a pivot search in the reduced matrix (defined as the $(n-k)$-by-$(n-k)$ submatrix of A, such that $k \leq i, j \leq n$, in the k-th iteration), followed by a row and a column swapping, and an update of range one in the same reduced matrix. In this code the Markowitz's heuristic was chosen for finding the pivot element, which is based on the minimization of the $M_{ij} = (R_i - 1)(C_j - 1)$ parameter, where R_i (C_j) counts for the number of nonzero elements in the i-th row (j-th column).

The first source of parallelism in the above code corresponds to the loops in charge of updating of the reduced matrix. The second source of parallelism 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 (and independent) if $A_{il} = A_{kj} = 0$.

The data flows and accesses for the right-looking LU code is shown in Figure 4, for the pivoting (row/column swapping) and sub-matrix updating operations, as well as fill-in. Observe that for an efficient parallel LU algorithm fast accesses to data by both rows and columns are required. Therefore we need to use more complex data structures than for the MGS algorithm.

3 Data Structures for Sparse Direct Methods

Direct methods, specifically LU and QR factorizations, decompose the original sparse matrix A by using simple row and/or column operations. When implementing these methods on a distributed-memory multiprocessor we should distribute the data across the local memories in such a way that workload balance and limited communication overhead is assured. Such



(a) (b) (c)

Figure 4: Pivot (a), update (b) and fill-in operations (c) of matrices L and U in LU

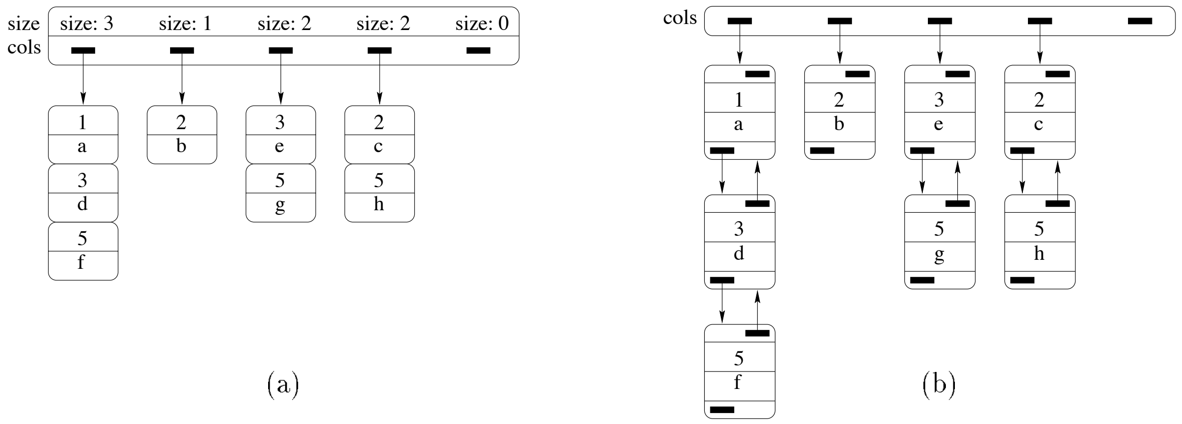
data distributions are discussed in the next Section.

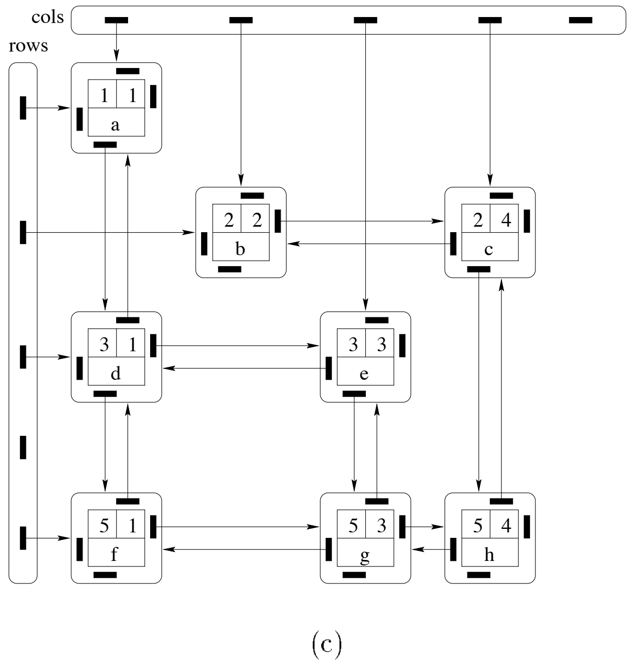
Once the local sparse matrices are obtained, we can select a number of data structures to store them. Typically, in order to save memory (and computations), zero elements of the sparse matrices are not stored. There are many methods for storing the nonzero elements of the matrices [4]. Here we will only discuss 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.

We can simply take some sort of packed vector format [10] (such as CRS or CCS), or use some other more complex and flexible data structure for storing the local sparse matrices. We have experimented with linked lists, pure CRS and CCS compressed formats and some mixed structure, depending on the type of data accesses we have to deal with.

In a MGS factorization of a sparse matrix only efficient accesses by matrix columns are needed (see Figure 2). This fact implies large memory and computation savings because we can use simple packed vectors (CCS, for instance) or one-dimensional doubly linked lists to store the local sparse matrices. As we can see in Figure 5 (b) each linked list represents one column of the local sparse matrix where its nonzero elements are arranged in growing order of the row index. Each item of the list stores the row index, the matrix element and two pointers. A simplification of the linked list is showed in Figure 5 (a), where the columns are stored as packed vectors and they are referenced by means of an array of pointers. The packed vectors do not have pointers inside and, therefore, this mixed structure requires only almost half as much memory space as the doubly linked list (considering the fact that in the C compiler of the Cray T3D, for example, the `int` as well as the `double` types take up 8 bytes both). Note that the data structure shown in Figure 5 (a) is a variant of the CCS compressed format, where the `DATA` and `ROW` vectors are joined together and the `COL` vector is represented as an array of pointers, with the size of the corresponding column associated with each one.

In a LU decomposition, on the other hand, we require a data structure such as a two-dimensional doubly linked list (see Figure 5 (c)) in order to make efficient data accesses both by rows and by columns (see Figure 4). Each item in such a dynamic structure stores not only the value and the local row and column indices, but also pointers to the previous and next nonzero element in its row and column (four pointers in total).





$$\begin{pmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & c & 0 \\ d & 0 & e & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ f & 0 & g & h & 0 \end{pmatrix}$$

(d)

Figure 5: Packed vectors and linked lists as efficient data structures for direct methods: (a) Packed vectors; (b) one-dimensional doubly linked list; (c) two-dimensional doubly linked list; (d) local sparse matrix

From the point of view of an efficient computation, packed vectors are very compact and allow fast accesses by rows and columns to the matrix elements (but not both at the same time). Linked lists are specially useful when more flexible accesses to the matrix elements (by rows and columns simultaneously) are needed. Additionally, there are two critical issues to be taken into account when factorizing a sparse matrix using a direct method, pivoting operations and fill-in.

Doubly linked lists make easy the insertion and deletion operations and, hence, we can deal with the fill-in and pivoting problems in a efficient way. In the case of using packed vectors, the fill-in problem is more difficult to solve. For a MGS code, for instance, we have followed this procedure: an auxiliary buffer (which is also a packed vector) long enough to store one column is allocated. During the updating process of a column, each nonzero entry, a previously existing one or a new one (fill-in), is stored in the auxiliary buffer, instead of in the original packed vector. This way, the new elements are just added to the buffer, but the zeroing of existing entries are discarded. After finishing the column updating process,

Data Structure	Algorithm	In place	Pivoting	Fill-in	Mem. Fragment.
Linked lists	All	yes	yes	yes	high
Packed vectors (w/ pointers)	All but Givens	no	yes	yes	low
CCS/CRS	All but Givens	no	no	yes	no

Table 1: Properties for the different data structures

the buffer contains the new packed column. Hence, this auxiliary buffer is just reallocated to a memory block of its exact size, becoming the new column of the sparse matrix, while the memory space of the old column is freed. Packed vectors, such as CRS and CCS, have also the inconvenience of not allowing the pivoting operation (column/row swapping) in a efficient way. This is the reason of using some mixed data structure, such as the one presented in Figure 5 (a). Column pivoting is implemented just interchanging pointer values.

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 more space memory than packed vectors. But one major problem is the memory fragmentation due to allocation/deallocation of items, and spatial data locality loss (cache inefficiency) during traversing rows and columns due pivoting operations (pivoting does not move data, only changes pointer references). For these reasons we have made an effort to develop an efficient parallel MGS and right-looking LU factorization algorithms avoiding linked lists.

Table 1 summarizes the discussed properties of the described data structures from the point of view of their behaviour when using in parallel sparse direct methods codes. Linked lists entry in this table corresponds to the structures shown in Figure 5 (b) and (c), whereas packed vectors (w/ pointers) corresponds to that of Figure 5 (a). The last entry is the standard CCS and CRS compressed formats. All these structures were implemented for the parallel right-looking LU and QR factorization, this last one using the MGS algorithm, Householder reflections and Givens rotations (the parallel implementation of the last two methods are described in [13] and [24]).

In any case, the use of data structures such as packed vectors implies the appearing of code segments like

```
DO i = a, b
  vu(row(i)) = vp(i)
ENDDO
```

and

```
init = c
DO i = a, b
  IF (vu(i) .NE. 0.0) THEN
    vp(init) = vu(i)
    row(init) = i
    init = init + 1
  ENDIF
ENDDO
```

The first piece of code is used to convert a packed vector to an unpacked one, and has an assignment statement with indirections at the left-hand side. The second construct is the reverse operation, used to pack the elements of a sparse vector. In this case we have a loop containing an induction variable. In general, the current data-parallel compilers (such as the T3D-Craft compiler) fail when compiling these kind of constructs. In the first case because the contents of `row()` is unkown until runtime and the compiler simply assume dependencies across iterations. In the second piece of code, the increment of the induction variable is included into a conditional statement.

$$\left(\begin{array}{cccccccc}
0 & 53 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\
0 & 0 & 0 & 17 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 13 & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
0 & 23 & 69 & 0 & 37 & 0 & 0 & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{array}\right)$$

(a)

$$\left(\begin{array}{cccccccc}
\underline{0} & 53 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\[6pt]
\underline{19} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\[6pt]
\underline{0} & 0 & \underline{0} & 17 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\[6pt]
\underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{13} & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\[6pt]
\underline{0} & 23 & \underline{69} & 0 & \underline{37} & 0 & \underline{0} & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{array}\right)$$

(b)

DATA	COL	ROW
19	1	1
13	4	1
69	2	2
37	3	2
		3
		5

(c)

Figure 6: (a) A sparse matrix. (b) The BRS partitioning of the sparse matrix for a 2×2 processor mesh, where the data elements for processor 0 are underlined. (c) The compressed local submatrix for processor 0

4 Pseudo-Regular Sparse Data Distributions

Current data-parallel languages, such as HPF, Vienna Fortran or Craft, include the most useful and simple schemes for distributing data across the processors, specifically block, cyclic and a combination of both. All these distributions allow us to parallelize in a efficient way most Fortran codes with regular accesses to data. However, this is not true for applications with irregular patterns for accessing data. These programs contain array indirections that produce not well-balanced parallel codes and/or with complex communication patterns when using regular data distributions.

Consider, for instance, applications that process data organized as sparse matrices. The use of compressed formats for storing sparse matrices implies the appearing of array indirections in the code. Our approach to deal with this kind of data accesses is to define pseudo-regular data distributions as extensions of the classical block and cyclic regular distributions, such as MRD (Multiple Recursive Decomposition) and BRS (Block Row Scatter) or BCS (Block Column Scatter) [22][1]. Let us concentrate on the last two data distributions, as they are extensions of the regular cyclic distribution, one of the most successful data distribution for matrix computations.

In our current situation, a sparse matrix is represented by a set of vectors (arrays), depending on the compressed format considered (CRS or CCS, for example). Instead of decomposing these arrays separately, as in commonly done, we follow the approach of considering the sparse matrix as a dense one, mapping this dense matrix on the processors using some standard data distribution and, finally, representing the local sparse matrices using the adopted compressed format. In this way, BRS (BCS) uses a cyclic mapping of the matrix represented by its CRS (CCS) format, as shown in figure 6.

An HPF-like description of the BRS data distribution 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

Distribution	SHL400	JPWH991	SHERMAN1	MAHINDAS	ORANI678	SHERMAN5
BCS	6	59	27	21	134	191
CHAOS	52	615	243	179	1228	1685

Table 2: Execution times (in sec.) for MGS using BCS and CHAOS

Matrix	Origin	$m \times n$	#elem(A)	% #elem(A)
SHL400	Linear programming problems	663×663	1712	0.39%
JPWH991	Circuit physics modeling	991×991	6027	0.61%
MAHINDAS	Economic modeling	1258×1258	7682	0.49%
ORANI678	Economic modeling	2529×2529	90158	1.41%
WELL1850	Least squares problems in surveying	1850×712	8758	0.66%
LNS3937	Compressible fluid flow	3937×3937	25407	0.16%
ORSREG1	Oil reservoir simulation	2205×2205	14133	0.29%
STEAM2	Oil reservoir simulation	600×600	13760	3.82%
SHERMAN1	Oil reservoir modeling	1000×1000	3750	0.37%
SHERMAN2	Oil reservoir modeling	1080×1080	23094	1.98%
SHERMAN5	Oil reservoir modeling	3312×3312	20793	0.19%

Table 3: Harwell-Boeing test matrices

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.

As a test to compare our parallel solutions with others using the standard *dense* data distributions, we made an experiment implementing the sparse algorithms using such distributions and the CHAOS runtime library [21], in order to deal with the irregular data accesses. We have inserted routines from the CHAOS library [23] to rebalance the load (and data) of the parallel MGS algorithm. Basically, through these routines, we have generated a translation table which assigns the global indices of matrix A to the different processors by following an irregular model. This table is distributed across the processors and is used by the routine `localize` to translate the global indices into local indices within each processor. It also generates a communication schedule which is used to gather the off-processor data which are needed during computation, and to scatter back local copies after computation. For the parallel algorithm MGS, table 2 shows a comparison of the execution times using BCS pseudoregular data distribution as opposed to the irregular distribution used in the CHAOS routines. The execution times have been taken in a cluster of 16 workstations Sun SPARCstation 4 with 85-MHz microSPARC-II processors in a PVM message-passing environment. The test sparse matrices were taken from the Harwell-Boeing suite and are described in table 3.

The CHAOS approach has a large number of communications and high memory overhead, as a consequence of accessing a large distributed data addressing table. This results in high execution times. BCS (and BRS) distribution, on the other hand, is adequate for sparse matrix problems (in particular, the MGS algorithm) because it exploits the data and computations locality and minimizes the communications. It does not require neither additional storage nor communications for addressing nonlocal data, as all the processors know where data are allocated.

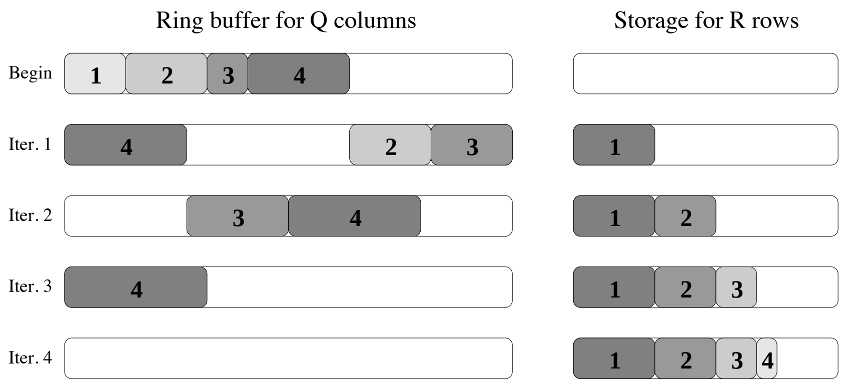


Figure 7: MGS-pv-SHMEM data structures, using BCS, for A and Q, BRS, for R, and packed vectors for storing the local matrices

5 Evaluating Data Structures and Distributions

5.1 Parallel Sparse MGS

We have designed three different parallel implementations of the MGS algorithm of the Figure 1. The parallelization procedure we have used is extensively explained in [7]. Here we will focus in the performance evaluation of the parallel MGS algorithm in terms of the discussed data structures and distributions. All the experimental results were obtained using a Cray T3D multiprocessor [6].

A first parallel implementation is a simplification of the considered MGS algorithm. The pivoting process is discarded and the upper triangular R matrix is the only one calculated (and the only needed in many codes that incorporate this kind of factorization). In this parallel version all the local sparse matrices are stored using packed vectors. Specifically, the initial A matrix is stored by packed columns (CCS compressed format) in a buffer. As R is calculated row by row, each one is stored as a packed vector (CRS format) in other buffer. As we do not intend to preserve Q, the buffer associated to A (and Q in the in-place algorithm) operates as a ring buffer, saving this way local memory in the processors.

Figure 7 shows an example for a A matrix with four columns. Each filled block represents a column for A (and Q) and a row for R. As we can see, in each k iteration, the last $n-(k+1)$ columns of Q are updated, and the k row of R is calculated. Whenever the end of the ring buffer storing Q is reached, the new columns are stored from the beginning of such buffer. Note that we have to dimension the ring buffer to a size long enough to preserve, during the k iteration, the k column of Q until the updating of the last column of Q (see Figure 1).

This parallel algorithm was coded in Fortran 77 and the Cray T3D SHMEM [5] native shared-memory library was used for communication. With these routines we can minimize the communication overhead at the expense of a very careful programming due to possible synchronization and cache coherence problems. It only has three communication operations per iteration (say, k), a reduction for obtaining the norm of the k column, a broadcast of the normalized k column to all processor columns of the mesh, and a reduction for calculating the dot products (see Figure 1). The rest of computations are completely local. This reduced number of communications comes from the absence of the pivoting operation and the resolution of the least squares problem. Figure 8 presents the parallel execution times for several processor meshes and Harwell-Boeing sparse matrices. We can see that, in general, the parallel efficiency is near optimal (in some cases we have superlinearity). A very low

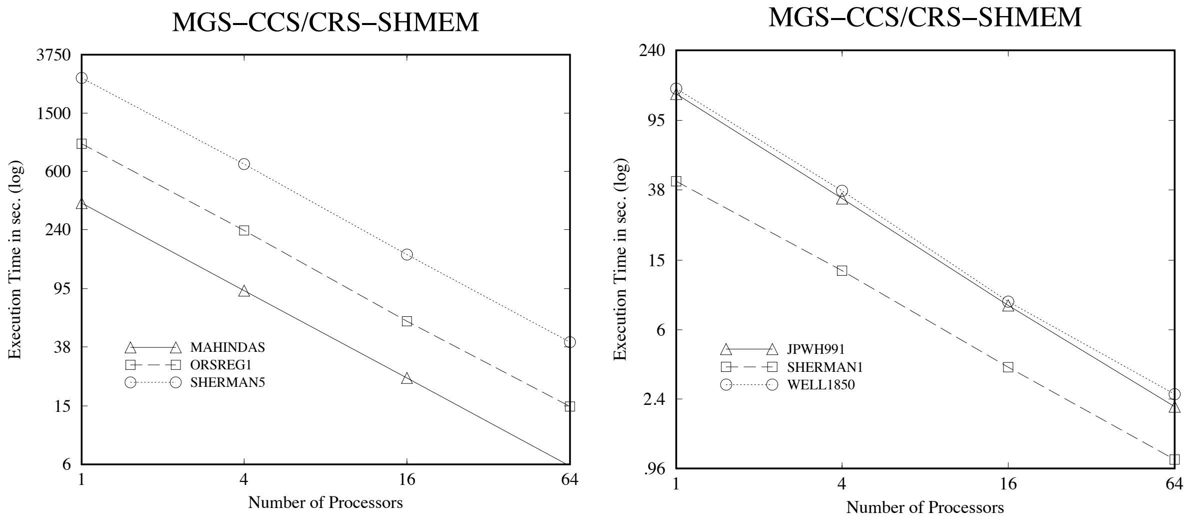


Figure 8: MGS execution times for different mesh sizes and Harwell-Boeing sparse matrices, using packed vectors (CCS and CRS) and Cray T3D SHMEM

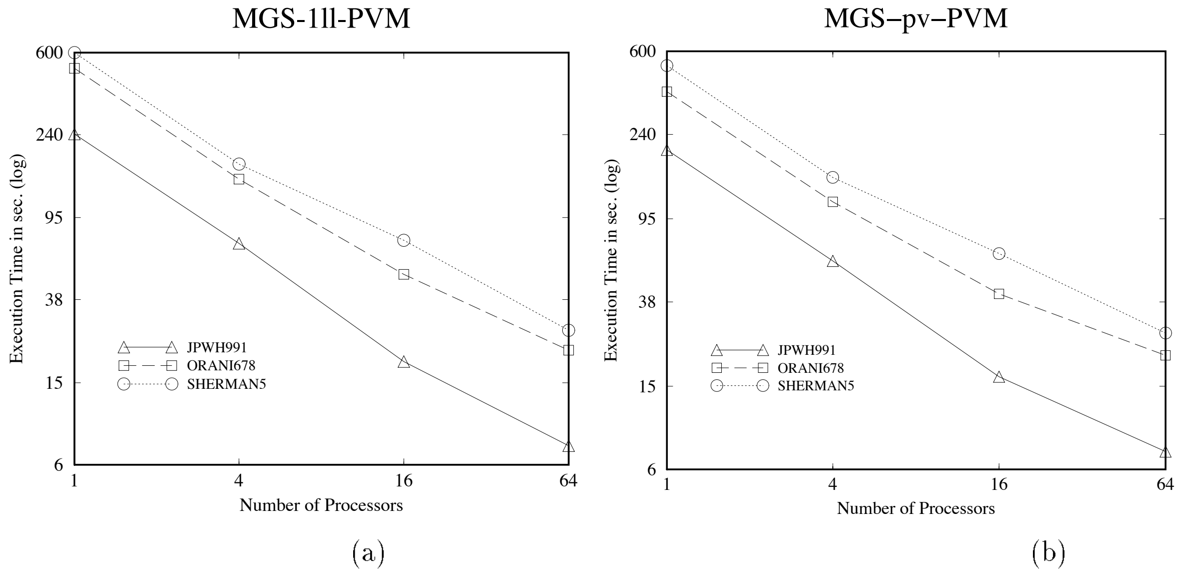
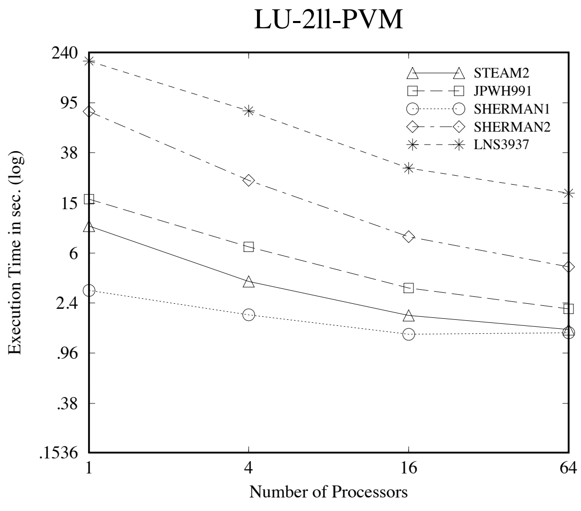
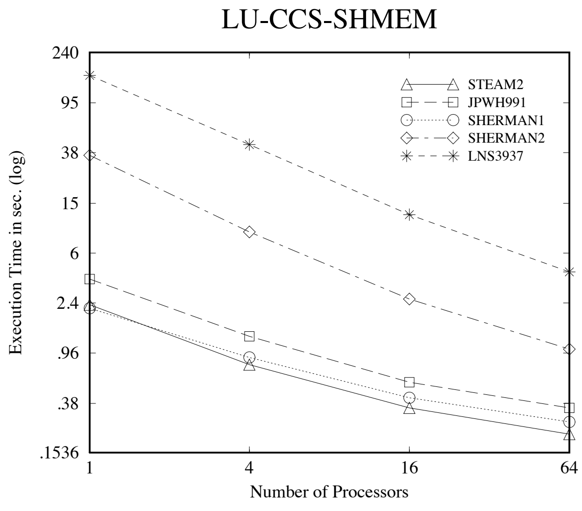


Figure 9: Parallel sparse MGS execution times for different mesh sizes and Harwell-Boeing sparse matrices, using linked lists (a) or packed vectors (b), and Cray T3D PVM

communication overhead (small number of communications and very efficient due to the use of SHMEM routines) and very high spatial locality exploitation (packed vectors allow us to avoid the use of pointers and dynamic memory allocations) justify this behaviour.

Two more parallel implementations of the complete MGS algorithm (including pivoting) were designed using the one-dimensional doubly linked list and the packed vectors shown in Figure 5 (b) and (a), respectively, for storing the local sparse matrices. Both versions were coded in C and using PVM routines 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. Efficient custom reduction operations suitable for the algorithm were also developed. Figure 9 shows the parallel execution times obtained for different processor mesh sizes and Harwell-Boeing sparse matrices (see Table 3) for both implementations. The execution times include the QR





(a) (b)

Figure 10: Parallel sparse LU execution times for different mesh sizes and Harwell-Boeing sparse matrices, using linked lists and Cray T3D PVM (a), and packed vectors (CCS) and Cray T3D SHMEM (b)

factorization as well as the solving of the least squares problem. The time required for data distribution and collection is not included because we assume that these algorithms are sub-problems within wider programs. As we can see, the execution times are in correspondence with the size of the input matrices. In the other hand, the dependence of the times with the size of the machine is almost linear, that is, the speedup (or efficiency) is very high. Comparing both figures it can be seen that the packed vectors based implementation is faster than the one using linked lists.

5.2 Parallel Sparse LU

Two parallel implementations of the right-looking LU algorithm, based on doubly linked lists and packed vectors, are described. Both parallel algorithms were designed using the same environment as in the parallel MGS.

The first parallel algorithm is mapped on a mesh of $P_1 \times P_2$ processors using a BCS (sparse cyclic) data distribution and a local representation with two-dimensional doubly linked lists. Note that the use of linked lists is almost unavoidable in order to handle in an efficient way the pivoting operation (that is, row and column swapping). This algorithm is extensively described and evaluated in [3]. Figure 10 (a) reproduces the parallel executions times obtained on the Cray T3D, for different mesh sizes and Harwell-Boeing sparse matrices (see Table 3 for a description). The parallel algorithm was coded in C and PVM routines were used for message-passing (the Cray T3D specific low latency PVM functions were used as in the MGS). There are some improvements to the linked list implementation developed by Jacko Koster [17], such as the reduction of the communications cost by an implicit pivoting together with sporadic workload re-balancing phases.

On the other hand, in the parallel version based on packed vectors, the A matrix is distributed following the BCS scheme and the sparse local matrices are stored in the implicit CCS format. While in the left-looking LU the fill-in only appears in the k column at each iteration, in the right-looking LU the fill-in affects to the whole reduced submatrix. This fill-in implies much more data movement than in the left-looking case, and also it could be associated with a garbage collection operation. We are also forced to minimize the number of row and column permutations during the factorization stage. To accomplish that the *analyze* and *factorize* steps are carried out separately[1]. For the *analyze* stage the MA50AD routine [12] (included in the MA48 software package) is used, but it has not been parallelized, it is just executed in only one processor before the *factorize* stage.

As the sparsity rate of the matrix decreases during the factorization, a switch to a dense LU factorization is advantageous at some point. The iteration beyond which a switch to dense code takes place is decided in the *analyze* stage. Hence, a sparse factorization code is executed initially, but when reaching the switch point a dense code continues the factorization. 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 stability. Once the sparse computations are carried out the reduced submatrix is scattered to a dense array. Therefore, the overhead of the switch operations is negligible and the reduced dense submatrix appears distributed in a regular cyclic manner.

Figure 10 (b) presents the parallel execution times for the BCS-based right-looking LU algorithm. This time the parallel algorithm was coded in Fortran 77 and the Cray T3D SHMEM [5] native shared-memory library was used for communication. The factorization numerical errors of our parallel algorithm are similar to those of the MA48 routine, and they can be reduced by applying a previous scaling to the matrix. Besides, the fill-in of our algorithm is also similar to that of the MA48 (using the same threshold). Nevertheless, the sequential time of our new algorithm is significantly higher than in the MA48 (very optimized).

6 A Proposal for Extending HPF Capabilities

Current data-parallel languages (HPF, Fortran D, Vienna Fortran, Craft ...) do not provide support to specify efficient data distributions for sparse matrices, nor flexible data structures for storing the local sparse matrices. However the distribution strategy of a sparse matrix across the processors and the data structures used to store the corresponding local sparse matrices are crucial decisions in order to obtain high parallel efficiencies. As an example, the experimental evaluation presented in the above section shows that we can obtain high efficiencies from parallel sparse direct methods codes just selecting carefully a suitable data structure and distribution. Indeed in all our parallel sparse algorithms we have used our pseudo-regular data distributions (specifically BRS and BCS and variants), obtaining very good speedups. None of them are included in the existing data-parallel languages.

It would be of major interest if we could incorporate to a current data-parallel language, such as HPF, some sort of data structure declaration, as linked lists and packed vectors, but without loosing the convenient matrix-like computation specifications. That is, if we could use some efficient data structure for storing the sparse matrices but with no need to deal with the complexities of use it directly. Figure 11 presents an example of such a specification for the MGS algorithm. The `SPARSE` directive is used to specify `A` as a sparse matrix that is stored using a LLCS (Linked List Column Storage) data structure (the structure shown in Figure 5 (b)). That is, with this directive we establish an identification between the matrix `A` and its machine storage representation. From this point on, we can specify the computations using matrix notations but with the confidence that the compiler will translate these specifications

[1]Obviously we must incorporate in some way a numerical pivoting to the *factorize* stage, only when the chosen pivot may introduce numerical instability.

```
!HPF$ PROCESSORS, DIMENSION(4,4) :: MESH
      PARAMETER (M=1850)
      PARAMETER (N=712)
      PARAMETER (ACCURACY=1.0E-20)
      INTEGER I, J, K, RANK, P
      REAL PIVOT
      REAL, DIMENSION(M) :: NORM, VSUM
      REAL, DIMENSION(N) :: VCOL

!HPF$ REAL SPARSE(LLCS()) :: A(M,N)
!HPF$ REAL SPARSE(LLCS()) :: R(N,N)

!HPF$ ALIGN VSUM(:),NORM(:) WITH A(:,*)
!HPF$ ALIGN VCOL(:) WITH A(*,:)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO MESH :: A, R

      RANK = N

!HPF$ INDEPENDENT
      DO J = 1, N
        NORM(J) = DOT_PRODUCT(A(:,J),A(:,J))
      ENDDO

      DO K = 1, N
        PIVOT = MAXVAL(NORM(K:N))

        IF (PIVOT < ACCURACY) THEN
          RANK = K-1
          EXIT
        ELSE
          P = MAXLOC(NORM(K:N))
          SWAP(A(:,K),A(:,P))
          SWAP(R(:,K),R(:,P))
          NORM(P) = NORM(K)
          NORM(K) = PIVOT
        ENDIF

        PIVOT = SQRT(PIVOT)
        R(K,K) = PIVOT
        A(:,K) = A(:,K)/PIVOT
        VCOL = UNPACK(A(:,K))

!HPF$ INDEPENDENT
        DO J = K+1, N
          R(K,J) = DOT_PRODUCT(VCOL, A(:,J))
        ENDDO

        VSUM(K+1:N) = UNPACK(R(K,K+1:N))

C  Local computations (following the owner compute rule)
        DO J = K+1, N
          NORM(J) = NORM(J) - VSUM(J) * VSUM(J)
!HPF$ FILLIN IN A
          DO I = 1, M
            A(I,J) = A(I,J) - VCOL(I) * VSUM(J)
          ENDDO
        ENDDO
      ENDDO
```

Figure 11: Outline for a HPF-like specification of the parallel MGS algorithm

to computations using linked lists.

We have to take a special action to deal with the fill-in problem, because the sparse matrices are stored compressed. For instance, consider the inner loop `I` at the end of the code in Figure 11. By default, this loop runs only over the non-zero elements of a `J` column of `A`. But for the column updating to be correct, the `I` loop must run over all the elements, zero and non-zero, of `A`, because `A(I,J)` could be zero but not `VCOL(I) * VSUM(J)` (fill-in). We have incorporated the `!HPF$ FILLIN IN A` just before the `I` loop in order to declare this fact, and change the normal behaviour of that loop.

In the proposed code, the meaning of some Fortran 90 and HPF standard procedures and functions should be extended. For instance, at the beginning of the code, the intrinsic F90 procedure `DOT_PRODUCT` is used for computing the square of the norm of the j column of A. In our case, that column is stored as a linked list and, therefore, this procedure should consider this storage representation. `DOT_PRODUCT` is also called later to compute the elements of the k row of R (first inner loop). In this case, its first argument is an array (`VCOL`) and the second one is a linked list (j column of A). There are also new procedures, `SWAP()` and `UNPACK()`. The first one is used to swap two arrays, in our case, two columns (pivoting operation). The second one converts a packed vector to an unpacked one (with zeroes). This second procedure is a local operation (no communications) because the sparse matrices are distributed using BCS, that is, a cyclic distribution as *dense* (unpacked) matrices, and then the local matrices are compressed locally, at each processor.

As the final indirections for access pattern depend on the actual input data, part of the analysis must be done during program execution. In order to support all this new functionality, we are in the process of extending our run-time library DDLY (Data Distribution Layer) [25] with a set of routines to handle linked lists, to be called from the output machine code generated by a HPF compiler.

7 Conclusions

One of the major reasons why data-parallel computation has not achieved outstanding results, in terms of functionality and efficiency, has been the development of very general programming and compilation techniques without a deep orientation to real codes.

It is clear that sparse direct methods are complex computations, in such a way that the current data-parallel technology does not have the elements to solve them efficiently and in an elegant manner. Two are the main difficult issues, pivoting and fill-in. Both are difficult to handle, time-consuming and with high memory overhead, because the matrices are stored in a compressed format (that is, only nonzero elements are stored). The more flexible way to manage these problems implies the use of some sort of linked list data structure to store the sparse matrices. Nowadays, there are no data-parallel tools with an efficient handling of this kind of data structures, and the cost of development for those strategies are still under evaluation.

This paper has presented a possible solution to deal with these computations in a HPF environment. The idea is to extend the HPF capabilities in such a way that the programmer may not only specify a particular sparse data distribution but also a particular sparse data storage representation. This way we establish an identification of the array representation of a sparse matrix at the programmer level and the storage representation (packed vectors, linked lists, ...) at the compiler (machine) level. The programmer deals with convenient matrix notations and the compiler translates them to machine code by using packed vectors and/or linked lists handling routines (from the DDLY runtime library). The results from our *manual* parallel implementations of direct solvers (*emulating* the output of a HPF compiler) show that we can obtain high efficiencies using the above strategy.

Acknowledgements

We gratefully thank Iain 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.

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, G.P. Trabado, M. Ujaldón and E.L. Zapata (1996), "Compilation Issues for Irregular Problems", *Works. on Parallel Programming Environments for High-Performance Computing*, L'Alpe d'Huez, France, pp. 187–199.

[3] 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).

[4] 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.

[5] R. Barriuso, A. Knies (1994), "SHMEM User's Guide for Fortran, Rev. 2.2", Cray Research, Inc.

[6] Cray Research, Inc. (1993), "Cray T3D. Technical Summary".

[7] R. Doallo, B.B. Fraguela, J. Touriño and E.L. Zapata (1996), "Parallel Sparse Modified Gram-Schmidt QR Decomposition", *Int'l. Symp. on High-Performance Computing and Networking (HPCN)*, Brussels, Belgium, pp. 646–653 (Springer-Verlag, LNCS 1067).

[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 (1996), "Sparse Numerical Linear Algebra: Direct Methods and Preconditioning", Tech. Report RAL-TR-96-047, Rutherford Appleton Lab., UK (State of the Art in Numerical Analisys Meeting, York).

[10] I.S. Duff, A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, NY.

[11] I.S. Duff and J.K. Reid (1993), "MA48, a Fortran Code for Direct Solution of Sparse Unsymmetric Linear Systems of Equations", Tech. Report RAL-93-072, Rutherford Appleton Lab., UK.

[12] 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.

[13] R. Doallo, J. Touriño and E.L. Zapata (1996), "Sparse Householder QR Factorization on a Mesh", *4th EUROMICRO Works. on Parallel and Distributed Processing*, Braga, Portugal, pp. 33–29.

[14] 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

[15] G.H. Golub and C.F. van Loan (1991), *Matrix Computations*, The Johns Hopkins University Press, MD.

[16] High Performance Fortran Forum (1993), "High Performance Language Specification, Ver. 1.0", *Scientific Programming*, 2 (1–2), 1–170.

[17] J. Koster (1993), "Parallel Solution of Sparse Systems of Linear Equations on a Mesh Network of Transputers", Tech. Report, Institute for Continuing Education, Eindhoven Univ. of Technology, The Netherlands.

[18] J.M. Ortega and W.G. Poole (1981), *Numerical Methods for Differential Equations*, Pitman Publishing, Marshfield, MS.

[19] H.M. Markowitz (1957), "The Elimination Form of the Inverse and its Application to Linear Programming", *Management Science*, 3, 255–269.

[20] D.M. Pase, T. MacDonald and A. Meltzer (1994), "The CRAFT Fortran Programming Model", *Scientific Programming*, 3, 227–253.

[21] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang and G. Fox (1995), "Runtime Support And Compilation Methods For User-Specified Data Distributions", *IEEE Trans. on Parallel and Distributed Systems*, 6 (8), 815–831.

[22] L.F. Romero and E.L. Zapata (1995), "Data Distributions for Sparse Matrix Vector Multiplication", *Parallel Computing*, 21, 583–605.

[23] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y. Hwang, M. Uysal and R. Das (1995), "A Manual for the CHAOS Runtime Library", Tech. Report CS-TR-3437 and UMIACS-TR-95-34, Computer Science Dept., University of Maryland.

[24] J. Touriño, R. Doallo and E.L. Zapata (1996), "Sparse Givens QR Factorization on a Multiprocessor", *2nd. Int'l. Conf. on Massively Parallel Computing Systems*, Ischia, Italy.

[25] G.P. Trabado and E.L. Zapata (1995), "Exploiting Locality on Parallel Sparse Matrix Computations", *3rd EUROMICRO Works. on Parallel and Distributed Processing*, San Remo, Italy, pp. 2–9.

[26] M. Ujaldón, S. Sharma, E.L. Zapata and J. Saltz (1996), "Experimental Evaluation of Efficient Sparse Matrix Distributions", *10th ACM Int'l. Conf. on Supercomputing*, Philadelphia, PN, pp. 78–85.

[27] M. Ujaldón, E.L. Zapata, B.M. Chapman and H.P. Zima (1995), "New Data-Parallel Language Features for Sparse Matrix Computations", *9th IEEE Int'l. Parallel Processing Symp. (IPPS'95)*, Santa Clara, CA, pp. 742–749.

[28] D.M. Young (1971), *Iterative Solution of Large Linear Systems*, Academic Press, NY.

[29] 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, University of Vienna, Austria.