HPF-2 Support for Dynamic Sparse Computations

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

August 1998
Technical Report No: UMA-DAC-98/11

Published in:

11th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'98)
Chapel Hill, North Carolina, pp. 230-246, August 7-9, 1998
published by Springer-Verlag, Berlin, Germany, S. Chatterjee, J.F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr and P.-C. Yew, Eds., LNCS no. 1656

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

HPF-2 Support for Dynamic Sparse Computations*

R. Asenjo[1], O. Plata[1], J. Touriño[2], R. Doallo[2], and E.L. Zapata[1]

[1] Dept. Computer Architecture, University of Málaga, Spain
{asenjo,oscar,ezapata}@ac.uma.es
[2] Dept. Electronics and Systems, University of La Coruña, Spain
{juan,doallo}@udc.es

Abstract. There is a class of sparse matrix computations, such as direct solvers of systems of linear equations, that change the fill-in (nonzero entries) of the coefficient matrix, and involve row/column operations (pivoting). This paper addresses the problem of the parallelization of these sparse computations from the point of view of the parallel language and the compiler. Dynamic data structures for sparse matrix storage are analyzed, permitting to efficiently deal with fill-in and pivoting issues. 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 a small set of new extensions to HPF-2 to parallelize these codes, and to support part of the new capabilities on a runtime library. This approach has been evaluated on a Cray T3E, implementing, in particular, the sparse LU factorization.

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 CM-Fortran, Vienna-Fortran [28], Fortran D [15] and *de facto* standard High-Performance Fortran (HPF) [19][20], 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 situation is different for irregular codes, where data-access patterns and workload are usually known only at runtime.

* This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC96-1125-C03), by the Xunta de Galicia (XUGA20605B96), by the European Union (BRITE-EURAM III BE95-1564), by the Human Capital and Mobility programme of the European Union (ERB4050P1921660), and by the Training and Research on Advanced Computing Systems (TRACS) at the Edinburgh Parallel Computing Centre (EPCC)

A approach to handle irregular computations is based on extending the data-parallel language with new constructs suitable to express non-structured parallelism. With this information, the compiler can perform at compile time a number of optimizations, usually embedding the rest of them into a runtime library. In Fortran D, for instance, the programmer can specify a mapping of array elements to processors using another array. Vienna-Fortran, on the other hand, lets programmers define functions to specify irregular distributions. HPF-2 [20] provides a generalized block distribution (GEN-BLOCK), where the contiguous array partitions may be of different sizes, and an indirect distribution (INDIRECT), where a mapping array is defined to specify an arbitrary assignment of array elements to processors.

A different approach is based on runtime techniques, that is, the non-structured parallelism is captured and managed fully at runtime. These techniques automatically manage programmer-defined data distributions, partition loop iterations, remap data and generate optimized communication schedules. Most of these solutions are based on the inspector-executor paradigm [22][8].

Current language constructs and the supportive runtime libraries are insufficiently developed, leading to low efficiencies when they are applied to a wide set of irregular codes. In the context of sparse computations, we found useful to inform the compiler not only about the data distribution, but also about how these data are stored in memory. We will call *distribution scheme* the combination of these two aspects (data structure + data distribution). We have developed and extensively tested a number of pseudo-regular distribution schemes for sparse problems, which combines natural extensions of regular data distributions with compressed data storages [2] [4] [23] [25] [26]. These distribution schemes can be incorporated to a data-parallel language (HPF) in a simple way. The programmer can use them easily and obtain high efficiencies from the parallelization of irregular codes.

The above mentioned distribution schemes are faced to static sparse problems (i.e. sparse matrices that do not change during computation). In this paper we will discuss data structures and distributions in the context of dynamic sparse matrix computations, involving fill-in and pivoting operations. Direct methods for solving sparse systems of linear equations, for instance, present this kind of computations. 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 limit fill-in. All these features make such sparse computations difficult to parallelize.

The rest of the paper is organized as follows. In Section 2 we describe and discuss the dynamic data distributions schemes we have tested to implement efficient parallel sparse codes involving pivoting and fill-in. Specifically, a direct method for the LU factorization is considered as a working example. In Section 3 we describe our proposal to extend HPF-2 for considering the above dynamic distributions. Experimental results validating our approach are presented in Section 4.

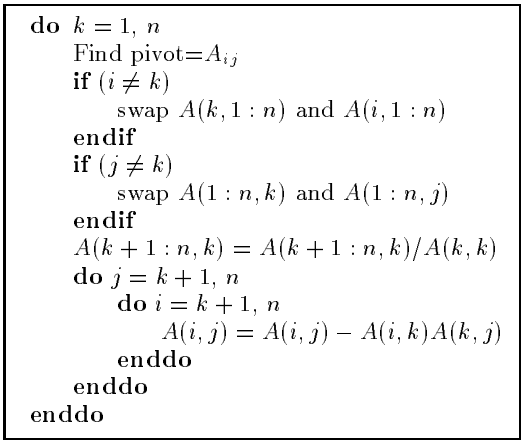


Fig. 1. LU algorithm (General approach, right-looking version)

2 Sparse Data Structures and Distributions

Distribution schemes are discussed in this section in the special context of dynamic sparse computations, where the fill-in and pivoting problems are both a key issue.

2.1 Sparse Data Structures

Usually, in order to save both memory and computation overhead, zero entries of sparse matrices are not explicitly stored. A wide range of methods for storing the nonzero entries of sparse matrices have been developed [5][12]. Here, we will consider two different approaches to store the sparse matrix: static data structures and dynamic data structures.

Static data structures are the most used in Fortran codes. Common examples are Compressed Row and Column Storages (CRS and CCS) [5]. If the computation includes fill-in entries and/or pivoting operations, even when we can simply take some variation of a compressed format (such as CRS or CCS), in many cases is preferably to use some other more complex and flexible data structure. We have experimented with linked lists, and some other hybrid (semi-)dynamic data structures, depending on the type of data accesses we have to deal with.

To simplify the discussion, let us take our working example application, the LU factorization of a sparse matrix, computed using a general method [1, 13]. These methods solve directly the sparse problem and shares the same loop structure of the corresponding dense code.

Fig. 1 shows an in-place code for the direct right-looking LU algorithm, where an n-by-n matrix A is factorized. The code includes a row and column pivoting operation (full pivoting) to provide numerical stability and preserve sparsity. Fig. 2 depicts the access patterns for the pivoting and updating operations on

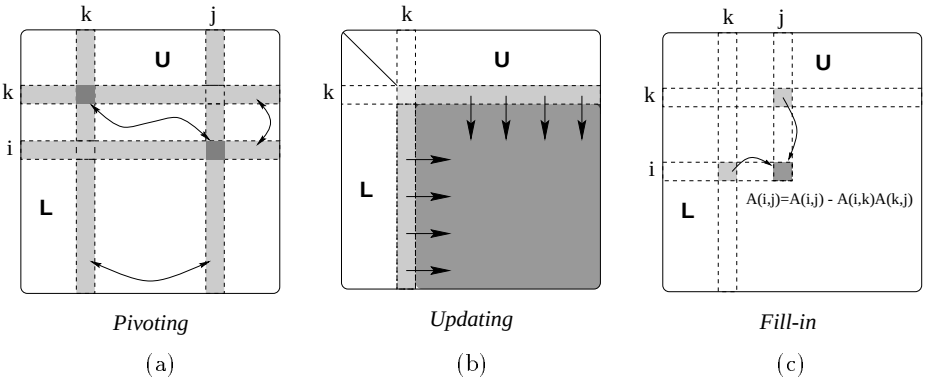


Fig. 2. Pivoting (a) and updating (b) operations, and fill-in (c) in right-looking LU

both matrices, L and U, and the generation of new entries. Note that efficient data accesses both by rows and columns are required.

The full-pivoting LU decomposition can store the coefficient matrix into a two-dimensional doubly linked list (see Fig. 3 (c)), to make efficient data accesses both by rows and columns. 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).

The complexity of the lists can be reduced if full pivoting is replaced by partial pivoting, where only columns (or rows) are swapped. This may imply large memory and computation savings as we can use a simple list of packed vectors, or a one-dimensional doubly linked list structure, to store the sparse matrices. As shown in Fig. 3 (b), each linked list represents one column of the sparse matrix, where its nonzero entries are arranged in growing order of the row index. Each item of the list stores the row index, the matrix entry and two pointers. A simplification of the linked list is shown in Fig. 3 (a), where columns are stored as packed vectors, and they are referenced by means of an array of pointers. The list of packed vectors do not have pointers and, therefore, this mixed structure requires much less memory space than the doubly linked list.

Compressed formats and lists of packed vectors are very compact and allow fast accesses by rows or by columns to the matrix entries (but not both at the same time). Linked lists are useful when more flexible data accesses are needed. Two-dimensional lists, for instance, allow accesses to both rows and columns with the same overhead. The fill-in and pivoting issues are easily managed when doubly linked lists are used, as they make easy the entry insertion and deletion operations. In the case of compressed formats (CRS, CCS ...) or a list of packed vectors, the fill-in problem is more difficult to solve. Compressed formats also have the inconvenience of not allowing the pivoting operation (column/row swapping) in an easy way. This can be overcome by using some mixed data structure, such as the list of packed vectors, or a linked list structure. Column pivoting is then implemented by just interchanging pointer values.

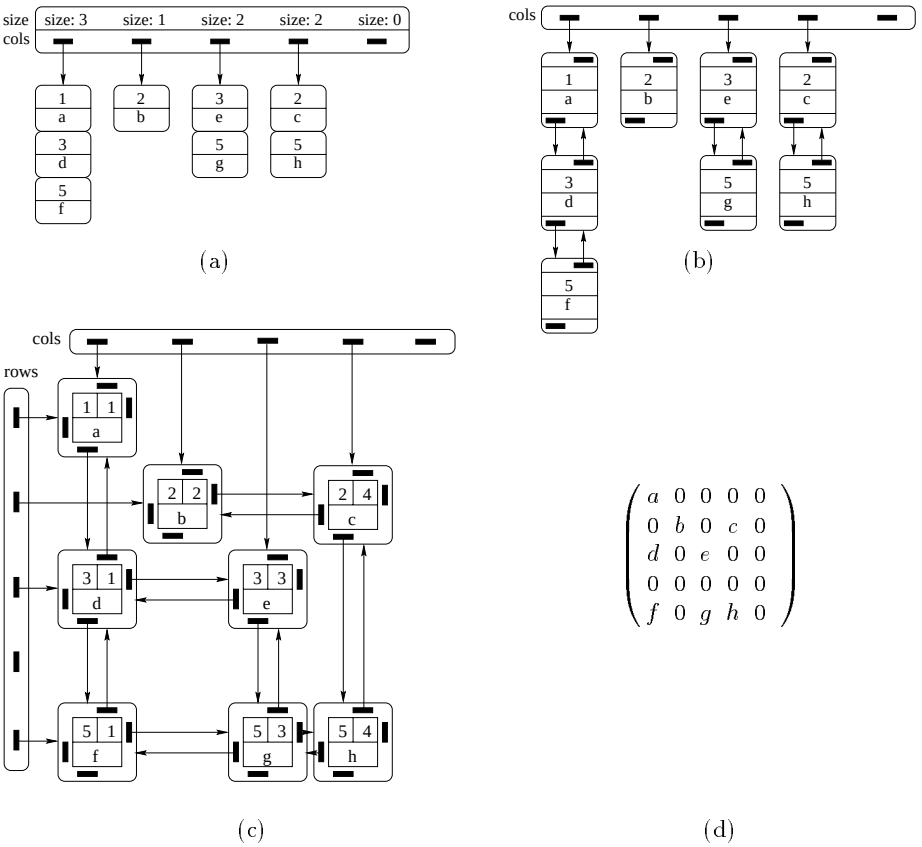


Fig. 3. Packed vectors and linked lists as efficient data structures for direct methods: (a) List of packed vectors; (b) one-dimensional doubly linked list; (c) two-dimensional doubly linked list; (d) local sparse matrix

Albeit their flexibility, linked lists have severe drawbacks. The dynamic memory allocation for each new entry, as well as the list traversing, are time-consuming operations. Additionally, they consume more space memory than packed vectors. But one major problem may be the memory fragmentation due to allocation/deallocation of items, and spatial data locality loss, which may make traversing rows and columns an expensive operation.

2.2 Dynamic Sparse Distribution Schemes

Four data storage schemes will be considered: `LLCS` (Linked List Column Storage), `LLRS` (Linked List Row Storage), `LLRCS` (Linked List Row-Column Storage) and `CVS` (Compressed Vector Storage), the first three schemes to represent sparse matrices, and the last one to represent sparse (one-dimensional) arrays. The `LLCS` storage scheme corresponds to the structure shown in Fig. 3 (b), that is, the ma-

```
!Doubly LLRS, LLCS (one-dimensional)        TYPE ptr
TYPE entry                                    TYPE (entry), POINTER:: p
  INTEGER:: index                           END TYPE ptr
  REAL:: value
  TYPE (entry), POINTER:: prev, next        TYPE (ptr), DIMENSION(n):: pex
END TYPE entry

!Doubly LLRCS (two-dimensional)
TYPE entry
  INTEGER:: indexi, indexj
  REAL:: value
  TYPE (entry), POINTER:: previ, prevj, nexti, nextj
END TYPE entry
```

Fig. 4. Fortran 90 derived types for the items of LLRS, LLCS and LLRCS storage schemes, and a definition of an array of pointers (pex) to these items

trix is represented by compressed columns stored as linked lists. Observe that in this figure the lists are doubly linked, but it can also be defined as singly linked, in order to save memory overhead. The `LLRS` storage scheme is similar to the `LLCS` scheme but considering linking by rows instead of columns. A combination of compressed columns and rows representation, interlinked among themselves, can be declared using the `LLRCS` storage scheme, as shown in Fig. 3 (c). As well as with the other two schemes, the entries can be singly or doubly linked. Finally, `CVS` scheme represents a sparse vector as two arrays and one scalar: the index array, containing the indices of the nonzero entries of the sparse array, the value array, containing the nonzero entries themselves, and the size scalar, containing the number of nonzero entries.

Fig. 4 displays the Fortran 90 derived types which may define the corresponding items of each kind of linked list. The first type corresponds to the `LLRS` and `LLCS` schemes (doubly linked), indistinctly, and the second one to the `LLRCS` scheme, doubly linked. The singly linked versions for these data types are equivalent but without the *prev* pointers. The list itself is declared also through a derived type, `pex`, which defines an array (or two) of pointers to the above items.

Once storage schemes have been defined, we can use the `SPARSE` directive to specify that a sparse matrix (or sparse array) is stored using a particular linked list scheme. This directive was previously introduced, for instance in [2] and in [23], in the context of static sparse applications. Fig. 5 shows the BNF syntax for the dynamic `SPARSE` directive. The first two data structures, `LLRS` and `LLCS`, are defined by two arrays of pointers (<*pointer-array-name*>), which point to the beginning and to the end, respectively, of each row (or column) list, and a third array (<*size-array-name*>), containing the number of elements per row (for `LLRS`) or per column (for `LLCS`). The option <*link-spec*> specifies the type of linking of the list data structure (singly or doubly). Regarding the `LLRCS` data structure, we have four arrays of pointers which point to the beginning and to

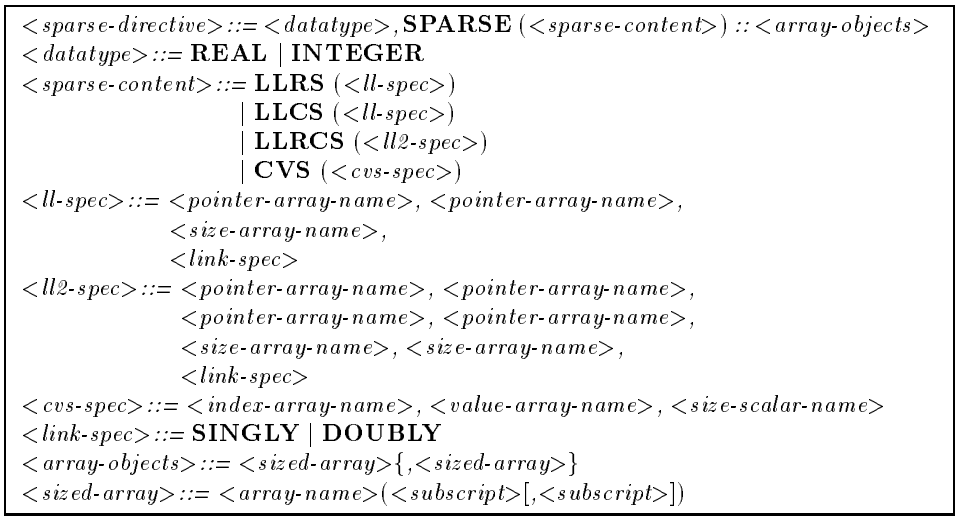


Fig. 5. Syntax for the proposed HPF-2 SPARSE directive with dynamic data structures

the end of each row and each column of the sparse matrix, and two additional arrays storing the number of elements per row and per column, respectively.

As an example, the following statement,

```
!HPF$ REAL, DYNAMIC, SPARSE (CVS(vi, vv, sz)):: V(10)
```

declares `V` as a sparse vector compressed using the CVS format. `V` will work in the code as a place holder of the sparse vector, which occupies no storage. What is really stored are the nonzero entries of the sparse array in `vv`, the corresponding array indices in `vi`, and the number of nonzero entries in `sz`. The place holder `V` actually provides an abstract object with which other data objects can be aligned and which can then be distributed. The `DYNAMIC` keyword means that the contents of the three arrays, `vi`, `vv` and `sz`, are determined dynamically, as a result of executing a `DISTRIBUTE` statement.

The HPF directives `DISTRIBUTE` and `ALIGN` can be applied to sparse place holders with the same syntax as in the standard. Distributing a sparse place holder is equivalent to distributing it as if it was a dense array (matrix). For instance, the statement,

```
!HPF$ DISTRIBUTE(CYCLIC) ONTO mesh:: V
```

considers `V` as a dense array (not compressed), mapping this array on the processors using the standard CYCLIC data distribution, and representing the distributed (local) sparse arrays using the CVS compressed format.

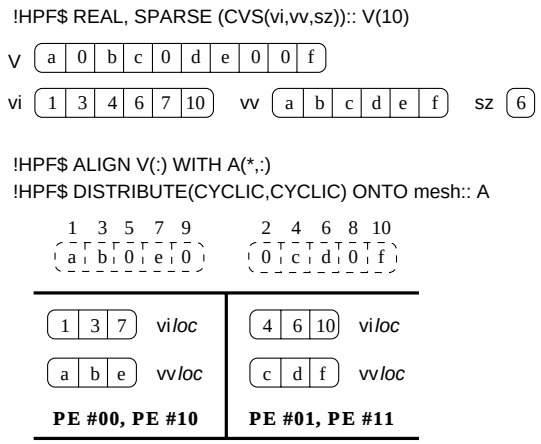


Fig. 6. Alignment and distribution of a sparse array on a 2×2 processor mesh

In the case of the `ALIGN` directive, however, the semantics is slightly different. From the next example code,

```
REAL, DIMENSION(10,10):: A
INTEGER, DIMENSION(10):: vi
REAL, DIMENSION(10):: vv
INTEGER:: sz
!HPF$ PROCESSORS, DIMENSION(2,2):: mesh
!HPF$ REAL, DYNAMIC, SPARSE (CVS(vi, vv, sz)):: V(10)
!HPF$ ALIGN V(:) WITH A(*,:)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO mesh:: A
```

the (nonzero) entries of `V` (that is, `vv`) are aligned with the columns of `A` depending on the positions stored in the array `vi`, and not in the corresponding positions in their own `vv` array (which is the standard semantics). Now, the `DISTRIBUTE` directive replicates the `V` array over the first dimension of the processor array `mesh`, and distributes it over the second dimension in the same way as the second dimension of the `A` matrix. Observe that in this distribution operation, `vi` is taken as the index array for the entries stored in `vv`. Fig. 6 shows the combined effect of alignment/distribution for a particular case.

The combination of the directives `SPARSE` and `DISTRIBUTE` defines the distribution scheme of a sparse matrix. The variable `V` in the example code above really works as a place holder for the sparse array. The `SPARSE` directive establishes the connection between the logical entity `V` and its actual representation (compressed format). The benefit of this approach is that we can use the standard HPF `DISTRIBUTE` and `ALIGN` directives applied to the array `V` and, at the same time, store the array itself using a compressed format. In the rest of the code, the sparse matrix is operated using directly its compressed format.

3 Parallel Dynamic Sparse Computations

The `SPARSE` directive establishes a link between the sparse matrix (or array) and its storage scheme. From this point on, we can choose to hide the storage scheme to programmers, and allow them to write the parallel sparse code using dense matrix notations. The compiler will be in charge of translating these dense notations into parallel sparse codes taking into account the storage schemes specified. However, this approach supposes a great effort in compiler implementation (the feasibility of its design is not clear), as well as the possibility of mixing in the same code place holders (dense notations) with *real* arrays. Bik and Wijshoff [7] and Kotlyar and Pingaly [21] propose a similar approach, based on the automatic transformation of a dense program, annotated with sparse directives, into a semantically equivalent sparse code. However, the design of such compiler is very complex, in such a way that no implementation of it is available for general and real problems.

A different approach is based on forcing programmers to use explicitly the compressed storage structures common in sparse codes, and allow them to use the place holders (dense notations) only for aligning and distributing purposes. Parallelism is constrained to the directives. If the parallel code is sequentially compiled (that is, the HPF-2 directives are taken as comments), the resulting code would run properly.

3.1 Parallel Sparse LU Code

The general (direct) right-looking LU factorization with partial pivoting (column swapping) will be considered. In most cases, partial pivoting leads to similar numerical error results than full pivoting, but at a lower cost. However, a matrix reordering stage (*analyze* stage) should be added before the factorization stage. This stage is in charge of updating the permutation vectors so as sparsity and numerical stability are preserved in the subsequent factorization stage. A partial numerical pivoting is however retained in the factorization stage to cover the case that the selected pivot in the analyze stage turns to be unstable during factorization.

Despite pivoting, the sparsity of the matrix usually decreases during the factorization. In such case, a switch to a dense LU factorization may be advantageous at some point of the computation. This dense code is based on Level 2 BLAS, and includes numerical partial pivoting in order to assure stability. At the switch point, the reduced sparse submatrix is scattered to a dense array. The overhead of the switch operation is negligible (as the analyze stage) and the reduced dense submatrix appears distributed in a regular cyclic manner. The threshold value used to switch from the sparse to the dense code was stated to 15% sparsity in our experiments.

Fig. 7 shows the declarative section of the parallel sparse LU code, using the proposed extensions to HPF-2. Matrix `A` is defined as sparse and stored using the LLCS data structure. The arrays of pointers `first` and `last` indicate the first and the last nonzero entry, respectively, of each column of `A` The array `vsize`

```
INTEGER, PARAMETER:: n=1000, dim=8
INTEGER:: k, i, j
REAL:: maxpiv, pivot, amul, product
INTEGER:: actpiv, pivcol
TYPE (entry), POINTER:: aux

TYPE (ptr), DIMENSION(n):: first, last, vpiv
INTEGER, DIMENSION(n):: vsize

REAL, DIMENSION(n):: vcolv, vmaxval
INTEGER, DIMENSION(n):: vcoli
INTEGER:: size

!HPF$ PROCESSORS, DIMENSION(dim):: linear
!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first, last, vsize, DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(n)
!HPF$ ALIGN iq(:) WITH A(*,:)
!HPF$ ALIGN vpiv(:) WITH A(*,:)
!HPF$ ALIGN vmaxval(:) WITH A(*,:)
!HPF$ ALIGN VCOL(:) WITH A(:,*)
!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
```

Fig. 7. Declarative section of the extended HPF-2 parallel sparse LU code

stores the number of nonzero entries on each column of `A`. The sparse array `VCOL` is also defined, stored using the CVS format. This array contains the normalized pivot column of `A`, calculated in each outer iteration of the algorithm.

The last sentence in the declaration section distributes cyclically the columns of the sparse matrix `A` over a one-dimensional arrangement of abstract processors (the one-dimensional characteristic is not essential). Previously, three dense arrays, `iq`, `vpiv` and `vmaxval`, were aligned with the columns of `A`. Therefore, after the distribution of `A`, these three arrays also appear distributed in a cyclic way over the processors. Finally, the sparse array `VCOL` is aligned with the rows of `A`. Hence, after distributing `A`, `VCOL` is replicated over all the processors. At each iteration of the main loop of the algorithm (loop k in Fig. 1), the owner of the column k of `A` selects and updates on `VCOL` the pivot column, which is consistently broadcast to the rest of processors to enable the subsequent parallel submatrix update. Fig. 8 shows an example of this declaration.

Fig. 9 presents the rest of the parallel LU code. The first action corresponds to the initialization of the array `vpiv`, which should point to the row that includes the pivot. This loop is parallel and no communications are required, as both arrays, `vpiv` and `first`, were aligned. Next, the outermost loop (loop k in Fig. 1) starts. Previously, the analyze stage has calculated the value `SwitchIter`, from which the sparse code switches to an equivalent dense one.

The first action inside the main loop corresponds to pivoting operations (column pivoting), in which we look for a stable pivot and, if possible, in agreement with the recommended permutation vector iq (obtained in the analyze stage). To fulfill the first condition, the pivot should be greater than the maximum ab-

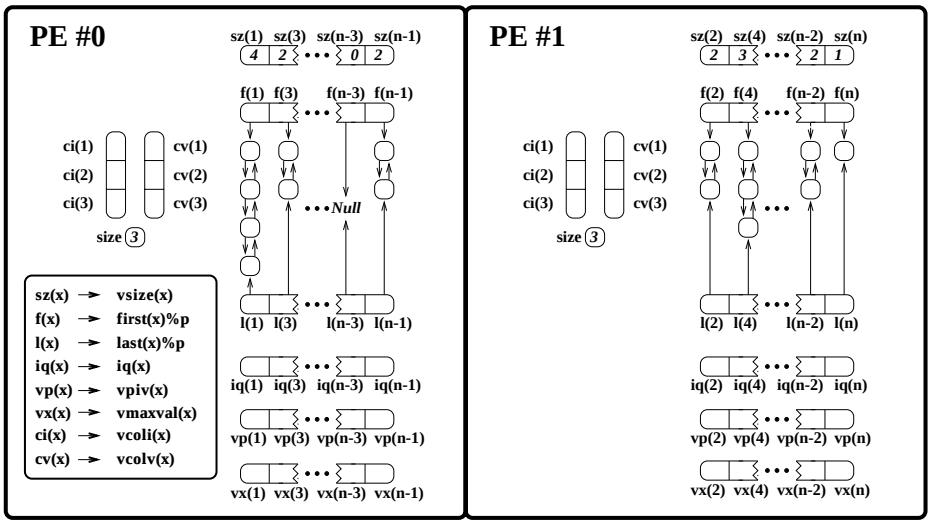
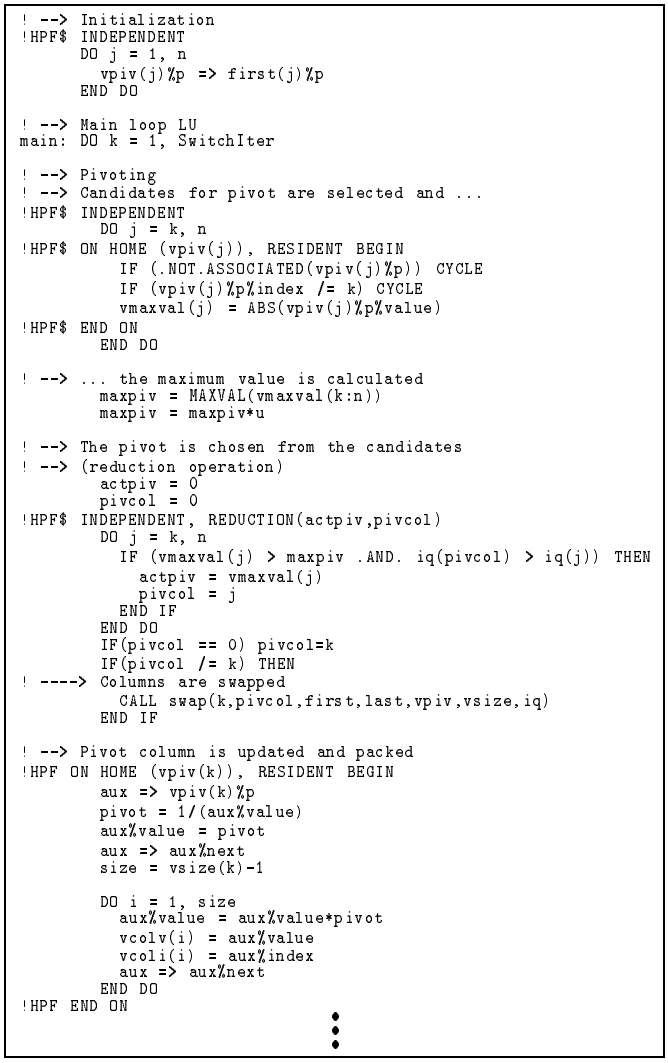


Fig. 8. Partitioning of most LU arrays/matrices on two processors, according to the HPF declaration of Fig. 7 (an even number of columns for `A`, and that the outer loop of the LU algorithm is in the fourth iteration, are assumed)

solute value of the pivot row times an input parameter called u ($0 \leq u \leq 1$). The maximum absolute value is calculated, using the Fortran 90 `MAXVAL()` intrinsic funtion, evaluated over `vmaxval` vector. The update of `vmaxval` takes place on the second INDEPENDENT loop which traverses the pivot row storing the absolute value of each entry on `vmaxval`. These entries are candidates for pivot. The `ON HOME (vpiv(j))` directive tells the compiler that the processor owning `vpiv(j)` will be encharged of iteration `j`. The `RESIDENT` annotation on the `ON HOME` directive points out to the compiler that all variables referenced during the execution of the directive's body are stored in the local memory of the owner of `vpiv(j)`. Thus, the compiler analysis is simplified and more optimized code may be generated.

Once the threshold `maxpiv` is obtained, the pivot is chosen in such a way that its value is greater than the above threshold, and, on the other hand, sparsity is preserved by following the *iq* recommendations. This computation is, in fact, a reduction operation, and consequently we annotate the corresponding `INDEPENDENT` loop with such directive. This user-defined reduction operation is indeed not considered by the HPF-2 standard, but its inclusion would not add any significant complexity to the compiler implementation. Finally, after selecting the pivot, the `swap()` routine is called to perform the permutation of the current column k and the pivot column of matrix `A`.

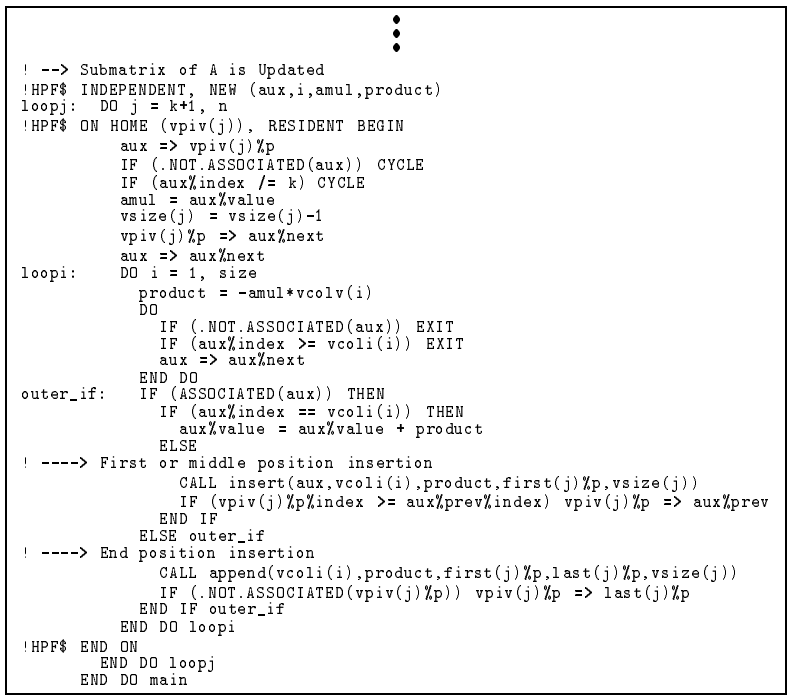
After the pivoting operation, the pivot column is updated and packed into the sparse `VCOL` array. This is computed by the owner of such column (`ON HOME` directive). The `ON HOME` directive is annotated with the `RESIDENT` clause, in-



```
! --> Initialization
!HPF$ INDEPENDENT
      DO j = 1, n
        vpiv(j)%p => first(j)%p
      END DO

! --> Main loop LU
main: DO k = 1, SwitchIter

! --> Pivoting
! --> Candidates for pivot are selected and ...
!HPF$ INDEPENDENT
        DO j = k, n
!HPF$ ON HOME (vpiv(j)), RESIDENT BEGIN
          IF (.NOT.ASSOCIATED(vpiv(j)%p)) CYCLE
          IF (vpiv(j)%p%index /= k) CYCLE
          vmaxval(j) = ABS(vpiv(j)%p%value)
!HPF$ END ON
        END DO

! --> ... the maximum value is calculated
        maxpiv = MAXVAL(vmaxval(k:n))
        maxpiv = maxpiv*u

! --> The pivot is chosen from the candidates
! --> (reduction operation)
        actpiv = 0
        pivcol = 0
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
        DO j = k, n
          IF (vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j)) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
        IF(pivcol == 0) pivcol=k
        IF(pivcol /= k) THEN
! ----> Columns are swapped
          CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF

! --> Pivot column is updated and packed
!HPF ON HOME (vpiv(k)), RESIDENT BEGIN
        aux => vpiv(k)%p
        pivot = 1/(aux%value)
        aux%value = pivot
        aux => aux%next
        size = vsize(k)-1

        DO i = 1, size
          aux%value = aux%value*pivot
          vcolv(i) = aux%value
          vcoli(i) = aux%index
          aux => aux%next
        END DO
!HPF END ON
                                  •
                                  •
                                  •
```

Fig. 9. Outline of an extended HPF-2 specification of the parallel right-looking partial pivoting LU algorithm (first part)

forming the compiler that all references by the processor that owns `vpiv(k)` are local. As `VCOL` is a replicated array, any update made on it is communicated to the rest of processors. Finally, the submatrix $(k+1:n, k+1:n)$ of `A` is updated. Loop `j` runs over the columns of the matrix, and it is parallel. The `NEW` directive prevents the compiler from considering inexistent data dependences due to variables that are actually private to each iteration.



```
                                    ⋮
! --> Submatrix of A is Updated
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
loopj:  DO j = k+1, n
!HPF$ ON HOME (vpiv(j)), RESIDENT BEGIN
          aux => vpiv(j)%p
          IF (.NOT.ASSOCIATED(aux)) CYCLE
          IF (aux%index /= k) CYCLE
          amul = aux%value
          vsize(j) = vsize(j)-1
          vpiv(j)%p => aux%next
          aux => aux%next
loopi:    DO i = 1, size
            product = -amul*vcolv(i)
            DO
              IF (.NOT.ASSOCIATED(aux)) EXIT
              IF (aux%index >= vcoli(i)) EXIT
              aux => aux%next
            END DO
outer_if:   IF (ASSOCIATED(aux)) THEN
              IF (aux%index == vcoli(i)) THEN
                aux%value = aux%value + product
              ELSE
! ----> First or middle position insertion
                CALL insert(aux,vcoli(i),product,first(j)%p,vsize(j))
                IF (vpiv(j)%p%index >= aux%prev%index) vpiv(j)%p => aux%prev
              END IF
            ELSE outer_if
! ----> End position insertion
              CALL append(vcoli(i),product,first(j)%p,last(j)%p,vsize(j))
              IF (.NOT.ASSOCIATED(vpiv(j)%p)) vpiv(j)%p => last(j)%p
            END IF outer_if
          END DO loopi
!HPF$ END ON
        END DO loopj
      END DO main
```

Fig.9 (cont.). Outline for an extended HPF-2 specification of the parallel right-looking partial pivoting LU algorithm (last part)

The code also contains the user-defined routines `append()` and `insert()` for list management, which are included in a Fortran 90 module. The `append()` routine adds an entry at the end of a list, while the `insert()` routine adds an element at the beginning or in the middle of a list.

4 Evaluating Results

Several parallel sparse implementations of the direct, right-looking LU algorithm have been designed. One of such implementations, for instance, is extensively described in [3]. Here, we will describe an implementation of the sparse right-looking partial pivoting LU algorithm using Fortran 90 and the Cray SHMEM library. All the experiments were conducted on a Cray T3E multiprocessor.

The columns of the sparse matrix A were cyclically distributed over the processors (linearly arranged), and stored in the local memories using one-dimensional doubly linked lists. This parallel algorithm is similar to the sequential version, but with local indices instead of the global ones, and Cray SHMEM routines performing communication/synchronization operations. All these operations were encapsulated into calls to the DDLY (Data Distribution Layer)

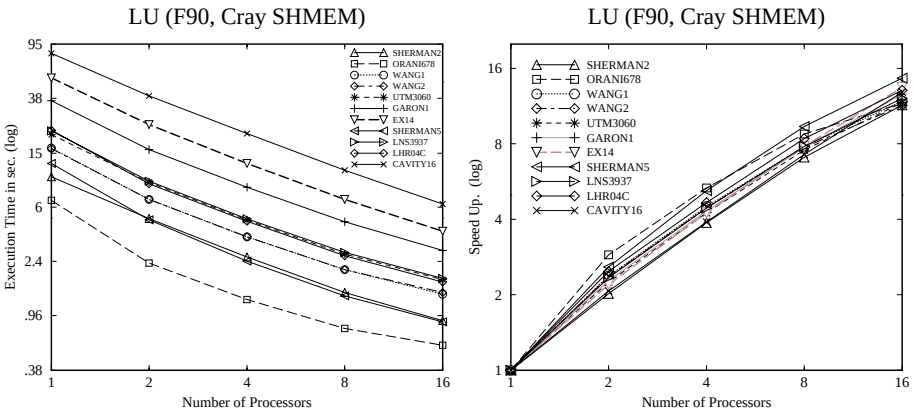


Fig. 10. Parallel sparse LU execution times and speed-up for different sparse matrices, using F90 linked lists and Cray T3E SHMEM

Table 1. Harwell-Boeing and Univ. of Florida test matrices

Matrix	Origin	n	# entries	sparsity
STEAM2	Oil reservoir simulation	600	13760	3.82%
JPWH991	Circuit physics modeling	991	6027	0.61%
SHERMAN1	Oil reservoir modeling	1000	3750	0.37%
SHERMAN2	Oil reservoir modeling	1080	23094	1.98%
ORANI678	Economic modeling	2529	90158	1.41%
WANG1	Discretized electron continuity	2903	19093	0.22%
WANG2	Discretized electron continuity	2903	19093	0.22%
UTM3060	Uedge test matrix	3060	42211	0.45%
GARON1	2D FEM, Navier-Stokes, CFD	3175	88927	0.88%
EX14	2D isothermal seepage flow	3251	66775	0.63%
SHERMAN5	Oil reservoir modeling	3312	20793	0.19%
LNS3937	Compressible fluid flow	3937	25407	0.16%
LHR04C	Light hydrocarbon recovery	4101	82682	0.49%
CAVITY16	Driven cavity problem	4562	138187	0.66%

runtime library [25]. The parallel code was designed in such a way that it could be the output of a hypothetic extended HPF-2 compiler (extended with the directives for the proposed distribution schemes). That is, it should be not considered as an optimized hand-coded program. Appendix A sketches such output parallel code (F90 plus DDLY calls).

Fig. 10 shows execution times and speed-up for the parallel LU algorithm. Test sparse matrices were taken from the Harwell-Boeing suite and University of Florida Sparse Matrix Collection [9] (see Table 1). The efficiency of the parallel code is high when the size of the input matrix is significantly large. We also carried out experiments considering meshes of processors instead of linear arrays,

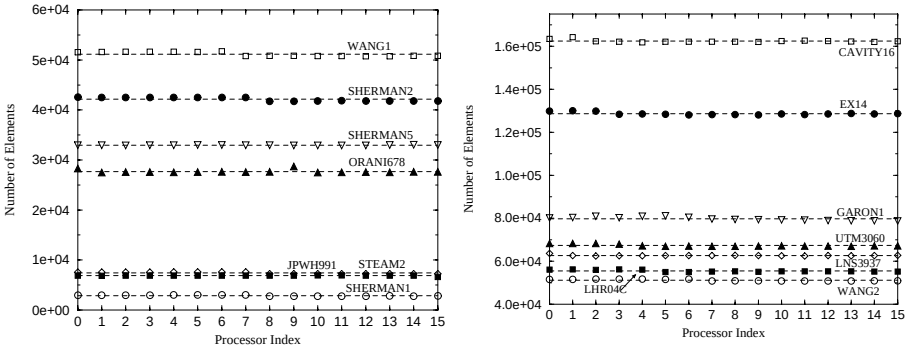


Fig. 11. Workload (non-null matrix values) on each processor after executing the parallel sparse LU factorization on a 16-processor system

Table 2. Comparison between Fortran 90 LU and MA48 (times in sec.)

	Times		Errors
Matrix	**F90 – MA48**	**ratio**	**F90 – MA48**
STEAM2	.5719 – .373	1.53	.15E-11 – .13E-11
JPWH991	1.039 – .563	1.84	.44E-13 – .82E-13
SHERMAN1	.5736 – .148	3.87	.14E-12 – .16E-12
SHERMAN2	10.05 – 9.77	1.02	.14E-5 – .15E-5
ORANI678	6.74 – 3.52	1.91	.70E-13 – .74E-13
WANG1	16.33 – 18.76	0.87	.11E-12 – .97E-13
WANG2	16.19 – 16.54	0.97	.53E-13 – .52E-13
UTM3060	20.57 – 22.68	0.90	.53E-8 – .58E-8
GARON1	36.39 – 28.80	1.26	.53E-9 – .21E-8
EX14	53.68 – 62.78	0.85	.28E+01 – .93E+01
SHERMAN5	12.58 – 6.09	2.06	.75E-12 – .59E-12
LNS3937	21.88 – 15.09	1.44	.15E-2 – .13E-2
LHR04C	22.15 – 10.42	2.12	.22E-3 – .10E-3
CAVITY16	81.23 – 88.48	0.91	.39E-9 – .49E-9

but the best times were obtained in the latter case and when the matrices were distributed by columns. Load imbalances due to fill-in (cyclic distribution) were not a problem for any matrix (see Fig. 11).

The sequential efficiency of the Fortran 90 implementation of the sparse LU algorithm was also tested. Table 2 presents comparison results from this implementation and the Fortran 77 MA48 routine [13]. We observe that the MA48 routine is significantly faster than our algorithm for many matrices, but it should be considered the fact that the Cray Fortran 90 compiler is not efficient generating code for managing lists. However, the resulting computing errors are practically the same for both algorithms. The main advantage of our approach is

its ease to be parallelized, as opposite to the MA48 routine, which is inherently sequential, as corresponds to a left-looking algorithm.

The analyze and solve (forward and backward substitution) stages of the LU algorithm were also implemented using the proposed methodology, but they are not presented here as no additional relevant aspect is contributed. Both execution times, and the fill-in, are comparable with those of the MA48 routine (they do not differ more than 10%).

5 Related Work

There are many parallel sparse LU factorization designs in the literature. From the loop-level parallelism point of view, the parallel pivot approach allows the extraction of an additional parallelism due to the sparsity of the matrix, besides the obvious one coming from the independences on the loops traversing rows and columns [3]. Coarser parallelism level can be exploited thanks to the elimination tree, which can be used to schedule parallel tasks in a multifrontal [14] code. It is also possible to use a coarse matrix decomposition to obtain an ordering to bordered block triangular form, as is done in the MCSPARSE package [17]. The supernodal [11] approach is also a parallelizable code [16].

Some of the above parallel solutions can be implemented using the approach described in this paper. Loop-level LU approaches can be implemented using the LLRCS data storage (in addition to LLCS), and some other complex reductions to choose a good parallel pivot set, but loosing some of the performance due to the semi-automatic implementation. The multifrontal approach, however, is not suitable to the linked list sparse directive, due to the use of different data storage schemes. However, they could be implemented using the basic BCS or BRS sparse distributions [2, 23]. The implementation of the supernodal code in [11] uses some sort of column compressed storage, but it would be necessary to simplify the memory management and the data access patterns to consider a data-parallel implementation of this code.

From the point of view of the automatic parallelization, dynamic data structures mean a lot of trouble to the compiler. In general, current data-parallel compilers, such as the T3D-Craft and the SGI MIPSpro Fortran77/90 and IRIS Power C (PCA) compilers [24], fail when dealing with codes involving pointers, such as those using linked lists. We can also identify a number of other compile time and runtime solutions to manage, in general, irregular codes. A significant portion of the work done was already introduced in Section 1 and Section 3.

6 Conclusions

This paper presented a solution to the parallelization of dynamic sparse matrix computations (applications suffering from fill-in and/or involving pivoting operations) in a HPF-2 environment. The programmer is allowed to specify a particular sparse data storage representation, in addition to a standard data distribution. Sparse computations are specified by means of the storage representation

constructs, while the (dense) matrix notation is reserved to declare alignments and distributions. Our experiments (a parallel sparse direct LU solver, *emulating* the output of an extended HPF-2 compiler) show that we can obtain high efficiencies using that strategy.

The research discussed in this paper gives new in-depth understanding in the semi-automatic parallelization of irregular codes dealing with dynamic data structures (list based), in such a way that the parallel code becomes a generalization of the original sequential code. An efficient parallel sparse code can be obtained by annotating the corresponding sequential version with a few number of HPF-like directives. The techniques described in this paper are not only useful to deal with the fill-in and pivoting problems, but they can also be applied to many other applications where the same or similar data structures are in use.

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 CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas), Spain, for giving us access to the Cray T3E multiprocessor.

References

1. R. Asenjo. Sparse LU Factorization in Multiprocessors. Ph.D. Dissertation, Dept. Computer Architecture, Univ. of Málaga, Spain, 1997.
2. R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata. Sparse Block and Cyclic Data Distributions for Matrix Computations. in *NATO Adv. Res. Works. on High Performance Computing: Technology, Methods and Applications*, Cetraro, Italy, 1994. (Elsevier Science B.V., The Netherlands, pp. 359–377, 1995).
3. R. Asenjo and E.L. Zapata. Sparse LU Factorization on the Cray T3D. *Int'l. Conf. on High-Performance Computing and Networking (HPCN'95)*, Milan, Italy, pp. 690–696, 1995.
4. G. Bandera, G.P. Trabado and E.L. Zapata. Extending Data-Parallel Languages for Irregularly Structured Applications. *NATO Adv. Res. Works. on High Performance Computing: Technology and Applications*, Cetraro, Italy, 1996. (Kluwer Academic Pub., The Netherlands, NATO ASI Series, Vol. 30, pp. 235–251, 1997).
5. R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Siam Press, 1994.
6. R. Barriuso, A. Knies. SHMEM User's Guide for Fortran, Rev. 2.2. Cray Research, Inc, 1994.
7. A. Bik. Compiler Support for Sparse Matrix Computations. Ph.D. Dissertation, University of Leiden, The Netherlands, 1996.
8. P. Brezany, K. Sanjari, O. Cheron and E. Van Konijnenburg. Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler. *Int'l. Conf. on High-Performance Computing and Networking (HPCN'95)*, Milan, Italy, pp. 526–531, 1995.

9. T. Davis, University of Florida Sparse Matrix Collection. *NA Digest*, 92(42), 1994, 96(28), 1996, 97(23), 1997. See `http://www.cise.ufl.edu/~davis/sparse/`.
10. J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM Press, USA, 1991.
11. J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li and J.W.H. Liu. A Supernodal Approach to Sparse Partial Pivoting. Tech. Report UCB/CSD-95-883, Computer Science Division, Univ. of California at Berkeley, CA, 1995.
12. I.S. Duff, A.M. Erisman and J.K. Reid (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, NY, 1986.
13. I.S. Duff and J.K. Reid. MA48, a Fortran Code for Direct Solution of Sparse Unsymmetric Linear Systems of Equations. Tech. Report RAL-93-072, Rutherford Appleton Lab., UK, 1993.
14. I.S. Duff and J.A. Scott. The Design of a New Frontal Code of Solving Sparse Unsymmetric Systems. *ACM Trans. on Mathematical Software*, 22(1):30–45, 1996.
15. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng and M. Wu. Fortran D Language Specification. Tech. Report COMP TR90-141, Computer Science Dept., Rice University, 1990.
16. C. Fu and T. Yang,. Run-time Compilation for Parallel Sparse Matrix Computations. *10th ACM Int'l Conf. on Supercomputing*, Philadelphia, pp. 237–244, May 1996.
17. K. Gallivan, B.A. Marsolf and H.A.G. Wijshoff. Solving Large Nonsymmetric Sparse Linear Systems Using MCSPARSE. *Parallel Computing*, 22(10):1291–1333, 1996.
18. G.H. Golub and C.F. van Loan. *Matrix Computations*, The Johns Hopkins University Press, MD, 1991.
19. High Performance Fortran Forum. High Performance Language Specification, Ver. 1.0. *Scientific Programming*, 2(1–2):1–170, 1993.
20. High Performance Fortran Forum. High Performance Language Specification, Ver. 2.0". Rice University, Houston, TX, February 1997.
21. V. Kotlyar and K. Pingali. Sparse Code Generation for Imperfectly Nested Loops with Dependences. *11th ACM Int'l Conf. on Supercomputing*, Vienna, Austria, 188–195, July 1997.
22. R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary and G. Fox. Supporting Irregular Distributions Using Data-Parallel Language. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(1):12–24, 1995.
23. L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *Parallel Computing*, 21(4):583–605, 1995.
24. Silicon Graphics, Inc.. IRIS Power C, User's Guide. SGI, Inc., Mountain View, CA, 1996.
25. G.P. Trabado and E.L. Zapata. Exploiting Locality on Parallel Sparse Matrix Computations. *3rd EUROMICRO Works. on Parallel and Distributed Processing*, San Remo, Italy, pp. 2–9, 1995.
26. M. Ujaldón, E.L. Zapata, B. Chapman and H.P. Zima. Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation. *IEEE Trans. on Parallel and Distributed Systems*, 8(10):1068–1083, 1997.
27. M. Ujaldón, E.L. Zapata, S.D. Sharma and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *Journal of Parallel and Distributed Computing*, 38(2):256–266, 1996.
28. H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald. Vienna Fortran – A Language Specification. Tech. Report ACPC–TR92–4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.

Appendix A

Fortran 90 code with calls to the DDLY library for the right-looking partial pivoting LU algorithm, as an output produced by an extended HPF-2 compiler.

```
!HPF$ PROCESSORS, DIMENSION(dim):: linear
! --> td is a topology descriptor for a (1 x dim) mesh
      CALL ddly_new_topology(td,1,dim)

!HPF$ REAL, DYNAMIC, SPARSE(LLCS(first, last, vsize, DOUBLY)):: A(n,n)
!HPF$ REAL, DYNAMIC, SPARSE(CVS(vcoli, vcolv, size)):: VCOL(n)
! --> Read a harwell-boeing matrix (av,c,r)
      CALL ddly_HB_read(n,alpha,av,c,r,b)
! --> Create a matrix descriptor, md_a, for A using CCS
      CALL ddly_new(md_a,CCS,DDLY_MF_REAL)
! --> Initialize md_a
      CALL ddly_init(md_a,n,alpha,av,c,r)
! --> Create an array descriptor, vd_vcol, for VCOL usign CVS
      CALL ddly_new(vd_vcol,CVS,DDLY_VF_REAL)
! --> Initialize vd_vcol
      CALL ddly_init(vd_vcol,vcoli,vcolv,size)

! --> Similar calls for other distributed arrays ...
      ...

!HPF$ DISTRIBUTE (*,CYCLIC) ONTO linear:: A
! --> BCS distribution of A (specified by md_a)
      CALL ddly_bcs(md_a, td)
! --> md_a is now the descriptor of the distributed matrix
! --> Change data storage CCS to LLCS
      CALL ddly_ccs_to_llcs(md_a,first,last,vsize,DOUBLY)

!HPF$ ALIGN iq(:) WITH A(*,:)
! --> iq aligned with the 2nd dimension of A
      CALL ddly_alignv(vd_iq,md_a,SecondDim)

!HPF$ ALIGN vpiv(:) WITH A(*,:)
      CALL ddly_alignv(vd_vpiv,md_a,SecondDim)

!HPF$ ALIGN vmaxval(:) WITH A(*,:)
      CALL ddly_alignv(vd_vmaxval,md_a,SecondDim)

!HPF$ ALIGN vcol(:) WITH A(:,*)
! --> VCOL aligned with 1st dimension of A
      CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)

!HPF$ INDEPENDENT
! --> Loop is partitioned
      DO j = ddly_LowBound(1), ddly_UpBound(n)
        vpiv(j)%p => first(j)%p
      END DO

! --> Main loop LU
main: DO k = 1, SwitchIter

! --> Pivoting
! --> Candidates for pivot are selected and ...
!HPF$ INDEPENDENT
! --> Loop is partitioned
        DO j = ddly_LowBound(k), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
              ****** Loop Body ****** (local to each processor)
!HPF$ END ON
        END DO

! --> ... the maximum value is calculated
! --> Parallel reduction splitted into two standard phases: ...
! --> ... First, local reductions, ...
        maxpiv = MAXVAL(vmaxval(ddly_LowBound(k):ddly_UpBound(n)))
! --> ... second, global reduction
        ddly_ReduceScalarMax(maxpiv)
        maxpiv = maxpiv*u

                                        •
                                        •
                                        •
```

```
                                  •
                                  •
                                  •
! --> The pivot is chosen from the candidates
! --> (reduction operation)
        actpiv=0
        pivcol=0
!HPF$ INDEPENDENT, REDUCTION(actpiv,pivcol)
! --> Loop is partitioned
        DO j = ddly_LowBound(k), ddly_UpBound(n)
          IF ( vmaxval(j) > maxpiv .AND. iq(pivcol) > iq(j) ) THEN
            actpiv = vmaxval(j)
            pivcol = j
          END IF
        END DO
! --> Global reduction
        pivcol = ddly_ReduceLocMaxAbs(actpiv,maxpiv,pivcol,iq)
        IF (pivcol == 0) pivcol=k
        IF (pivcol /= k) THEN
! ----> Columns are swapped
           CALL swap(k,pivcol,first,last,vpiv,vsize,iq)
        END IF

! --> Pivot column is updated and packed
!HPF ON HOME (vpiv(k)), RESIDENT (A(*,k)) BEGIN
! --> This loop appears due to the ON HOME directive
        DO dum = ddly_LowBound(k),ddly_UpBound(k)
          ****** ON HOME Body ******
        END DO
!HPF END ON

! --> Pivot column is broadcast (because VCOL is replicated)
        CALL ddly_aligncvs(vd_vcol,md_a,FirstDim)

! --> Submatrix of A is updated
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
! --> Loop is partitioned
        DO j = ddly_LowBound(k+1), ddly_UpBound(n)
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) BEGIN
          ****** ON HOME Body ******
!HPF$ END ON
        END DO
      END DO main

!
!=====================================================================
! The independent loops (DO j = globa, globb) are translated into
! (DO j = ddly_LowBound(globa), ddly_UpBound(globb))
! my_pe is a global variable which contains the processor id
! N$PES = # of PEs (global variable on the Cray T3E platform)

      INTEGER FUNCTION ddly_LowBound (i)
        INTEGER i
        ddly_LowBound = (i-1)/N$PES+1
        IF (my_pe < MOD(i-1,N$PES)) ddly_LowBound = ddly_LowBound+1
      END FUNCTION ddly_LowBound

      INTEGER FUNCTION ddly_UpBound (i)
        INTEGER i
        ddly_UpBound = i/N$PES
        IF (my_pe < MOD(i,N$PES)) ddly_UpBound = ddly_UpBound+1
      END FUNCTION ddly_UpBound
```