Compilation Issues for Irregular Problems

R. Asenjo
G.P. Trabado
M. Ujaldon
E.L. Zapata

February 1996
Technical Report No: UMA-DAC-96/07

Published in:

Workshop on Parallel Programming Environments
L'Alpe d'Huez, France, April 1-5, 1996

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

Compilation issues for irregular problems [*]

R. Asenjo, G.P. Trabado, M. Ujaldon and E.L. Zapata
Computer Architecture Department
University of Malaga
29071 Malaga
SPAIN
e-mail: {asenjo,guille,ujaldon,ezapata}@atc.ctima.uma.es

Abstract

The paper presents a set of strategies for addressing the parallelization of irregular problems in distributed memory machines. Our methods are targeted to data-parallel compilers, though some of them are also useful for manual parallelization. We treat the specification of a broad range of irregular applications, like numerical algorithms, iterative and direct methods for the solution of linear systems, and some physic problems such as finite elements and molecular dynamics. These problems are characterized by a particular data representation which enforces the handling of complex elements like accesses through indirections, dynamic data creation and migration and pointer referencing that go beyond current compilation techniques.

1 Introduction

Sparse matrices are used in a large number of important scientific codes, such as linear systems solvers, molecular dynamics, finite element methods and climate modelling. Unfortunately, these applications are hard to parallelize efficiently, particularly using automated compiler techniques. This is because sparse matrices are represented using compact data formats which necessitate heavy use of indirect addressing through pointers stored in *index arrays*. Since these index arrays are read in at runtime, current compilers cannot analyze which matrix elements will actually be touched in a given loop, making it impossible to determine non-local memory accesses at compile-time.

We provide a set of data-parallel extensions for the specification of sparse algorithms, which allows the user the selection of a particular representation for the data as well as its distribution. With the knowledge of these elements, it is proven the capability of a compiler for handling the specification and generating efficient code [21].

The paper also deals with the *fill-in* problem, which takes place when new nonzero elements are added to the matrix at run-time: Those elements have to be stored dinamically on memory and their accesses have to be considered, which may also introduce some reorganization in the algorithm. There have been many efforts in the development of algorithms dealing with fill-in, such as the wide variety of direct methods for the solution of linear systems (LU, QR and many others). We focus on the LU decomposition in order to illustrate the list of problems and the methods we propose for their solution.

The third scenario with increasing complexity consists of scientific applications working with unstructured domains, such as molecular dynamics and finite element simulations. They do not use sparse matrices to represent problem domains, though its internal representation pose very similar problems to those of sparse linear algebra problems: Data migration through the problem domain yields the necessity of a data reorganization during the program execution and the key to achieve efficiency lies again on exploiting the spatial locality intrinsic to the algorithm.

The paper is structured as follows. Section 2 outlines different representation schemes widely used in irregular problems. Section 3 describes different data distribution schemes for sparse matrices in distributed-memory multiprocessors. Section 4 treats the parallelization of sparse data-parallel applications using those representations and distributions. Section 5 considers additional problems in the scope of irregular computation. Section 6 addresses different solutions for the implementation of the fill-in problem. We

[*]The work described in this paper was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC92-0942. The authors assume all responsibility for the contents of the paper.

conclude in Section 7 and 8 with some related work and the conclusions we draw from this work.

2 Irregular problems domain representation

This section provides a brief overview of different methods for the representation of data in the field of irregular problems. Though we focus on sparse matrices, our analysis can be extended to unstructured problem domains without loosing generality.

A distinction is made between static and dynamic formats; the latter is related to situations where the position and/or the number of elements of the domain change during the program execution.

2.1 Static formats for sparse matrices

Several methods have been proposed in the literature [3] for the representation of sparse matrices with the aim of saving both memory and computations. For the purposes of our work in static matrices we have chosen two formats:

- **CRS** (Compressed Row Storage) (e.g. Figure 1.b) represents a sparse matrix `A` using three vectors: the `Data` vector stores the non-zero values of the matrix, as they are traversed in a row-wise fashion; the `Column` vector stores the column index of each non-zero element in the `Data` vector; and the `Row` vector marks the beginning of each row in the `Data` vector.
- **CCS** (Compressed Column Storage) (e.g. Figure 1.c) is similar to CRS, just changing rows by columns.

Our approach can easily be extended to other format choices and therefore this selection doesn't affect severely to its generality.

2.2 Static formats for unstructured problem domains

Scientific applications handling unstructured domains deal essentially with unordered sets of geometric elements scattered through the geometric domain of the problem (or *problem space*). Each element is represented by a tuple of n real values determining its position on each dimension in the problem space.

Since sequential programmers didn't care about the possibility of distributing the representations mentioned, no special organization of the table is assumed, which leads to assign positions in the table for elements that are very far from others involved in the same computational step.

Nevertheless, this representation can be arranged internally to make it suitable for a distribution as we will see later on. So, a distributed representation can be derived with the same data structure as in the sequential program but applied to the local domain, thus yielding a very similar parallelized code than the sequential one. For example, the representation of a set of points in a 3-D problem space is given in Figure 2.

2.3 Dynamic formats

There are several strategies for storing sparse matrices that allow the insertion of new elements. Some of them have been proposed by Stappen et al [18], who target their analysis to the LU decomposition. We focus on two strategies:

First, in the two-dimensional doubly linked list, a dynamic data structure which links all the nonzero elements belonging to the same row in a list, and those belonging to the same column in another list. Therefore, each item stores, together with the value and the local indices, pointers pointing to the previous and next element in its row and column. This data structure allows a fast data access, either by rows or by columns, and makes it easier the deletion operation. However, each item occupies a significant amount of memory. Though HPF-2 covers pointers handling [11], it remains to be seen whether the pointers will be implemented within semi-automatic tools in a near future.

The second alternative consists of the data structure used by the MA48BD routine[1], which is not actually a dynamic one, but it deserves to be mentioned here for being able of storing the elements the fill-in process creates. It stores the elements in a CCS format and writes the L and U components for the matrix in another area of memory, also in CCS format. For the particular case of the LU decomposition, the price to pay for using such strategy lies in loosing the in-place behaviour of the algorithm.

Figure 3 describes this approach, where the matrix containing the linear system is represented in the first three vectors, (`AA`, `IRNA` and `IPTRA`). Similarly, `FACT` stores the nonzeros for matrices `L` and `U`, sortered by columns (first, those of `U`, then, those of `L`), `IRNF` stores the row indexes, and `IPTRU` and `IPTRL` contain

[1]Stage *Factorize* in the MA48 Harwell Subroutine Library for performing a LU decomposition [6].

$$\begin{pmatrix}
0 & 53 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\
0 & 0 & 0 & 17 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 13 & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
0 & 23 & 69 & 0 & 37 & 0 & 0 & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{pmatrix}$$

(a)

DA	CO	RO
53	2	1
21	7	2
19	1	3
16	8	5
72	6	6
17	4	7
93	5	8
13	7	9
44	5	11
19	8	14
23	2	17
69	3	
37	5	
27	1	
11	4	
64	7	

(b)

DA	RO	CO
19	3	1
27	10	3
53	1	5
23	9	6
69	9	8
17	5	11
11	10	12
93	6	15
44	8	17
37	9	
72	4	
21	2	
13	7	
64	10	
16	3	
19	8	

(c)

Figure 1: Sample sparse matrix A (a) and CRS (b) and CCS (c) representations for it.

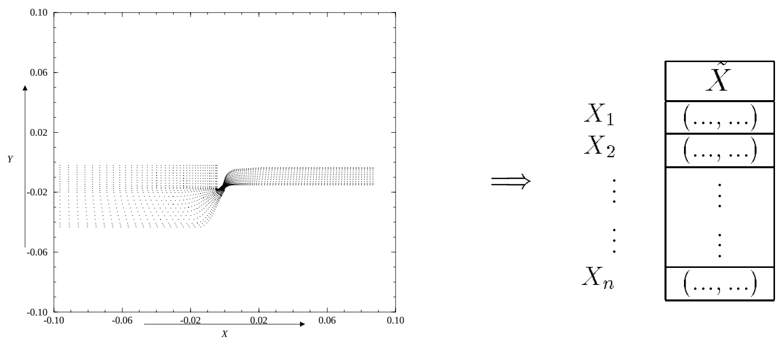


Figure 2: Left: 2-D problem domain. Right: Internal representation for problem space.

pointers to the end of each `U` and `L` column, respectively. The remaining positions in vectors `FACT` and `IRNF` may be used as workspace.

3 Data Distributions

In current approach, the selected representation formats for sparse matrices involve three arrays (`Row`, `Data` and `Col`) for a single sparse matrix. Instead of decomposing these three arrays separately, as is commonly done, we adopt the approach of first mapping the (imaginary) dense matrix across a logical mesh ($P_1 \times P_2$) of P processors, followed by assigning each non-zero element to the processor that owns the corresponding index of the dense matrix.

Thus, the *dense* matrix's index space `(1:nrows, 1:ncols)` is distributed in such a way that each processor is assigned a *region* of the index-space described by the expression:
$(Row_{lo} : Row_{hi} : stride_r, \quad Col_{lo} : Col_{hi} : stride_c)$. In this way, processors may be assigned regions of unequal sizes but the indexes of each region can be described in a regular manner.

Each region is again a sparse matrix which is locally represented on each processor using the very same format than in the global case.

Distribution strategies fullfilling such conditions are:

- The **Multiple Recursive Decomposition (MRD)** [1, 15] recursively decomposes the sparse matrix over P processors using horizontal and vertical partitions, until the matrix has been de-

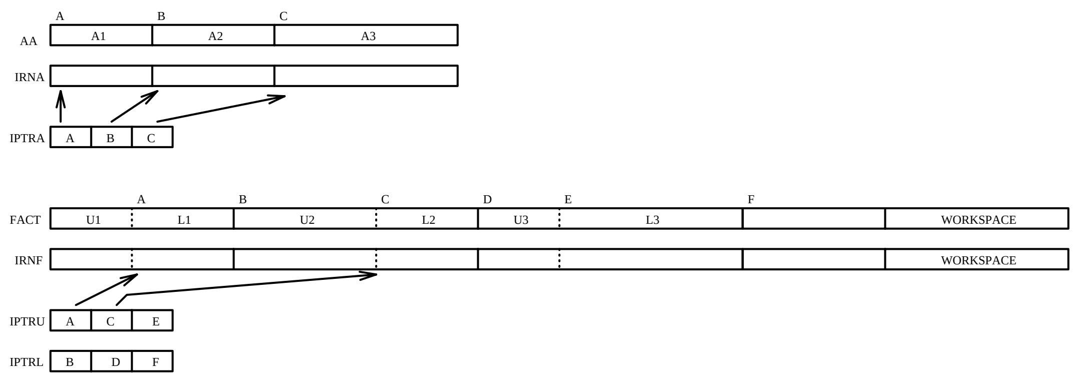


Figure 3: Main data structure for the MA48-Factorize Subroutine.

composed into $P_1 \times P_2$ rectangular submatrices. At each stage of the partitioning process, the non-zeros in the submatrix of that stage are divided as evenly as possible (see Figure 4). For a CRS local representation, the MRD-CRS distribution scheme is originated, and similarly for MRD-CCS with respect to CCS.

- The **Block Row Scatter (BRS)** [1, 15] uses a cyclic mapping of the matrix represented by CRS among P processors. The matrix is subdivided using a stencil of size P_1, and each processor gets the non-zero elements matching its position in the stencil. For situations where the matrix is represented by CCS, the **Block Column Scatter (BCS)** scheme is considered.

  BRS and BCS similar to scatter-decomposition distribution schemes, and are useful in situations where the concentration of non-zeros may be extremely uneven across the domain, and unpredictable.

For unstructured problem domains only one array is used to represent the elements. As computations in scientific applications only implicate elements closely situated in the space of problem domain, it only makes sense distributing it in such a way that the resulting distribution preserves most of the spatial locality of the algorithm.

Then, we developed the **Non Discrete Multiple Recursive Decomposition (ND-MRD)**. It yields a decomposition of problem domain similar to that of the MRD, but with the main singularity of dealing with real coordinates for the position of elements in domain, instead of integer indices for row and column.

Again, the local representation of each resulting partition is exactly the same, so that code structure can be preserved in a later step generating parallel code, thus making compiler development easier. Figure 6 shows a partition of the problem domain presented in figure 2. ND-MRD balances the number of elements comprised in each region of the partition.

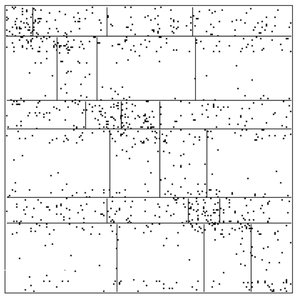
4 Compilation techniques for sparse codes

4.1 Language Support

Using user-specified directives [24], a compiler can automatically insert the runtime calls necessary to distribute matrices and vectors as described earlier. To achieve this, the user has to give to the compiler the following pieces of information for each sparse matrix: the name of the matrix, its index domain, the type of its elements, the sparse storage format together with the names of the arrays responsible for representing such a format, and finally the data distribution selected for the matrix. The following set of directives give an example of the extensions that data-parallel languages require to convey this information to a compiler.

```
REAL A(N,M),  SPARSE (CRS(Data, Column, Row)),  DYNAMIC
DISTRIBUTE  A::(MRD)
REAL Y(N)  DYNAMIC, CONNECT (C) WITH A(C,:)
```

The keyword DYNAMIC indicates that the distribution will be determined at runtime as a result of executing the DISTRIBUTE statement. The CONNECT keyword is used to align a dense vector with one of the dimensions of the sparse matrix; here the vector `Y` is aligned with the columns of `A`.



(a)

$$\begin{pmatrix}
\mathbf{0} & \mathbf{53} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 21 & 0 \\
\mathbf{19} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 & 16 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 72 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{17} & \mathbf{0} & 0 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{93} & 0 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 13 & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
0 & 23 & 69 & 0 & 37 & 0 & 0 & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{pmatrix}$$

(b)

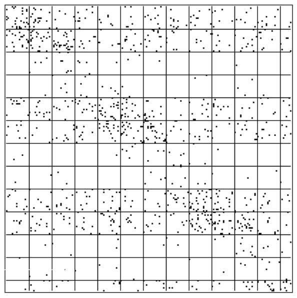
DA^0
53
19
17
93

CO^0
2
1
4
5

RO^0
1
2
2
3
3
4
5
6

(c)

Figure 4: (a) MRD partitioning of a sparse matrix onto a mesh of 6x4 processors. (b) A smaller matrix mapped onto 2x2 processors; P(0,0)'s region in bold. (c) The local submatrix data-structures.



(a)

$$\begin{pmatrix}
\underline{0} & 53 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 21 & 0 \\
\underline{19} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{0} & 16 \\
0 & 0 & 0 & 0 & 0 & 72 & 0 & 0 \\
\underline{0} & 0 & \underline{0} & 17 & \underline{0} & 0 & \underline{0} & 0 \\
0 & 0 & 0 & 0 & 93 & 0 & 0 & 0 \\
\underline{0} & 0 & \underline{0} & 0 & \underline{0} & 0 & \underline{13} & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 19 \\
\underline{0} & 23 & \underline{69} & 0 & \underline{37} & 0 & \underline{0} & 0 \\
27 & 0 & 0 & 11 & 0 & 0 & 64 & 0
\end{pmatrix}$$

(b)

DA^0
19
13
69
37

CO^0
1
4
2
3

RO^0
1
1
2
2
3
5

(c)

Figure 5: (a) BRS partitioning of a sparse matrix. (b) A smaller BRS matrix partition using a 2x2 processor stencil; P(0,0)'s elements underlined. (c) The local submatrix data-structures.

4.2 Parallelization

The compiler uses the database as a source of information from where the work distribution strategies obtain the input data required for splitting the loop iterations onto processors as evenly as possible. For this purpose, the *owner computes rule* is applied: Assignments with scalars or replicated arrays in the left hand side are executed by all the processors, whereas assignments with distributed arrays in the left hand side are only executed by its *owner*. *Parallel loops* [2] are in this way split by assigning to every processor the set of iterations in which the corresponding left hand side variable is local.

However, this task becomes difficult in sparse loops, because of the references to sparse vectors that are normally used in loop bounds: The way the sparse data are stored in memory makes *Row* to be used as loop bound in the CRS format every time the sparse algorithm requires to process the elements of an entire row of the matrix, and the same happens with the *Column* vector in the CCS format. Loops with these features can be partitioned by translating the global meaning of that *Row* or *Column* vector into its local meaning so that each processor sweeps over the elements in its local matrix. The benefits that this strategy yields are a well-balanced workload and a local access to the loop bounds [3].

The only exception to the owner computes rule is the REDUCE statement [26], in which a reduction operation is defined over a entire dimension of a sparse matrix.. The distribution of iterations is performed this time first by making the reduction over the local sparse matrix of every processor and then inserting an extra loop that traverses the corresponding dimension of the processors mesh to produce the final global

[2]Loops in which the data dependence don't prevent the parallelization of their iterations to be performed

[3]In the inspector-executor paradigm followed here (see [14] and subsequent sections of this paper), non-local loop bounds yield an extra communication stage, as the values of those bounds have to be known before entering the loop. Hence, the optimization of ensuring the bounds as local becomes valuable since avoids a level of communication

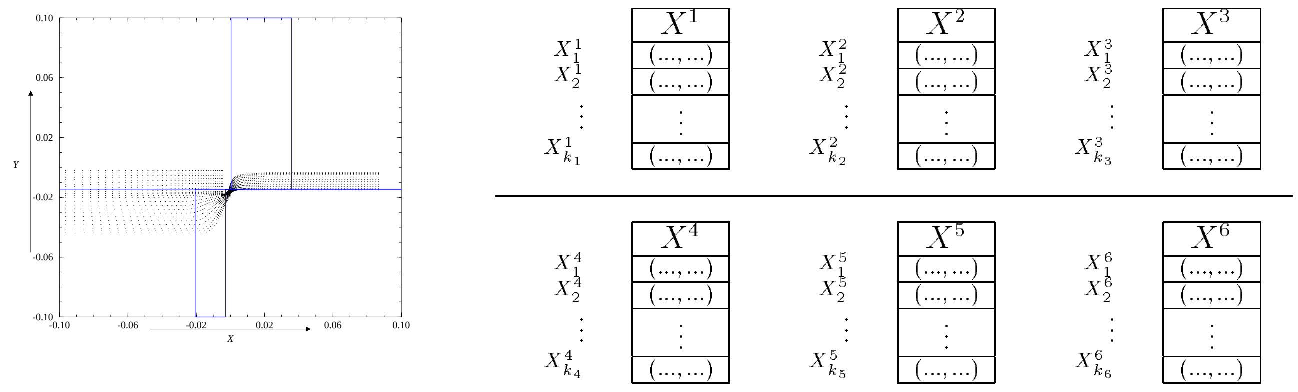


Figure 6: Left: ND-MRD partitioning of a 2-D problem domain onto a mesh of 2 × 3 processors. Right: Local representation of resulting partitions

value from the set of partial results. The parallelization is guided by the array(s) in the right hand side, which improves the locality and the communications overhead.

When the *parallelization* phase concludes, the output SPMD program has already delimited the set of computations to be performed by each processor.

4.3 Code Generation

At this point of the compilation process, the variables participating in the computation of the right hand side may address non-local memory; when the pattern of nonzero entries of the sparse matrix is unknown at compile-time, the analysis process that carries this out has to be postponed until run-time (basically, till the MRD or BRS/BCS distributions are computed and the local vectors of the CRS or CCS format are determined).

In such cases, the compiler uses an *inspector-executor* strategy. Loops are tranformed so that a preprocessing step, called an **inspector** is generated. During program execution, the inspector examines the global addresses referenced and determines which (if any) non-local elements must be fetched. It starts with a loop collecting all the sparse references into auxiliar arrays, which are then analized by a run-time routine (*localize*) to determine the set of local and non-local indices used in the loop computation. This information is saved on a *schedule* so that the inspector phase doesn't have to be computed again when the loop is reused and the communication pattern remains invariant, thus amortizing the cost of this inspector.

Then, an **executor** uses this information to fetch the data and to perform the computation in the original loop body. It starts with a *gather* routine, which uses the local sparse vectors and the schedule as input to fill the buffer with the corresponding non-local data. The references in the original loop are then changed to point either to the local index of the array or the location in the buffer where the non-local indices were brought. Those locations are stored on auxiliar arrays which are traversed through loop iterations by means of counters explicitly inserted by the compiler for this purpose. Those transformations lead to an executor loop in which all the computation is local.

Run-time techniques based on the inspector/executor paradigm have been fairly well studied and described in [14]. However, the distributions and the storage formats used in this case allow some important optimizations to decrease the overhead of such techniques when applied over sparse codes. More precisely, the compile-time information provided by the user-directives is exploited, and, on top of this, efficient schemes are developed for solving indirections [22] and traslating the global indices to the processor and local index where each datum may be located [23].

5 Compilation for scientific codes handling unstructured domains

5.1 Language support

To allow optimization of these scientific codes, the programmer inserts some directives describing the problem space (number of dimensions) and identifying the storage format and names of the array holding the elements of the domain. With this directives, the compiler has enough knowledge of the special role of these data structures to handle the code accessing them in a different way [20].

The next code lines show an example of how to identify to the compiler and distribute a 2-D problem domain with a set of N elements representing vertices.

```
REAL A(:,:), SPARSE(NONDISCRETE(X(N,2))),DYNAMIC
REAL F(N), DYNAMIC, CONNECT (F(I) WITH X(I,:))
DISTRIBUTE A::(MRD,MRD)
```

Note the keyword for distribution is the same used when distributing sparse matrices, although both implementations are different. However, the style of directives is the same.

Again, the keyword DYNAMIC indicates that distribution depends on the value of data at the time of program execution. So, distribution is not computed until run time.

Note also that once the compiler has information about the special characteristics of an array (X), some other standard directives like CONNECT can relate other objects with that array, inheriting also some of those special characteristics. In this example, aligning F with X results in distributing F following the ND-MRD distribution of the problem domain.

After distributing elements in problem domain, we must solve the other main problem in scientific codes: accessing elements though the usage of indirection arrays. These indirection arrays define relationships between elements (usually representing a static mesh) and are predefined outside of the program (they form part of problem input data).

The challenge here is distributing these arrays in such a way that each member of an indirection array is located in the same processor than the elements pointed by the indirection. This is indicated to the compiler by using an extension of the ALIGN directive.

```
INTEGER G(M), DYNAMIC
ALIGN (G(I) WITH X(G(I),:)) :: G
```

This directive in the example example above indicates that each element *G(i)* must be stored in the same processor as the element of X indexed by the value in *G(i)*.

The following code is an example of how to modify a scientific code to give the compiler information of the role of some special data structures in the code that will be treated with special optimizations.

```
      REAL A(:,:,:),DYNAMIC,SPARSE(NONDISCRETE(X(NV,3))
      INTEGER GRID(NNINTC,8), DYNAMIC,
     *  ALIGNED (GRID(I,*) WITH X(GRID(I,1))

      DOUBLE PRECISION DIREC1(NNCELL), DYNAMIC
      DOUBLE PRECISION DIREC2(NNINTC), DYNAMIC
      DOUBLE PRECISION VAR(NNCELL), DYNAMIC
      DOUBLE PRECISION BP(NNINTC), BE(NNINTC), DYNAMIC
      DOUBLE PRECISION BW(NNINTC), BN(NNINTC), DYNAMIC
      DOUBLE PRECISION BS(NNINTC), BH(NNINTC), DYNAMIC
      DOUBLE PRECISION BL(NNINTC), DYNAMIC

      ALIGN ($(I) WITH GRID(I,:)) :: DIREC1, DIREC2
      ALIGN ($(I) WITH GRID(I,:)) :: BP, BE, BW, BN
      ALIGN ($(I) WITH GRID(I,:)) :: BS, BH, BL

      INTEGER LCC(NNINTC,6), DYNAMIC
      ALIGN (LCC(I,:) WITH GRID(I,:))

      INTEGER NINTCI,NINTCF,NEXTCI,NEXTCF

      OPEN(20,STATUS='OLD',FILE='gccg-input')
      READ(20) NINTCI,NINTCF,NEXTCI,NEXTCF
      READ(20) LCC
      READ(20) BS, BE, BN, BW, BL, BH, BP
      CLOSE(20)

      DISTRIBUTE A::(MRD)
      DO 22 T=1,TIMESTEPS
C   need neighbor elements in DIREC1 pointed by LCC
      ALIGN (DIREC1(LCC(I,:)) WITH DIREC2(I))
         ...
         FORALL 11 NC=NINTCI,NINTCF
         DIREC2(NC)=BP(NC)*DIREC1(NC) -
     *              BS(NC)*DIREC1(LCC(NC,1)) -
     *              BW(NC)*DIREC1(LCC(NC,4)) -
     *              BL(NC)*DIREC1(LCC(NC,5)) -
     *              BN(NC)*DIREC1(LCC(NC,3)) -
     *              BE(NC)*DIREC1(LCC(NC,2)) -
     *              BH(NC)*DIREC1(LCC(NC,6))
 11      CONTINUE
C   actualize DIREC1 from data in DIREC2
         ...

 22   CONTINUE
      STOP
      END
```

Note the **SPARSE** qualifier for the 3-D problem domain $A(:,:,:)$, describing its representation by means of the $X(NV,3)$ array. Once the compiler is aware of the special type of the X array, other data structures are specially related to X by using extended **ALIGN** directives.

In this example, $GRID$ represents the definition of a mesh, in which each element is a cube delimited by 8 elements of X. This relationship must be taken into account at the time of distributing $GRID$. So, the attribute

```
    ALIGNED (GRID(I,*) WITH X(GRID(I,1))
```

gives the compiler a valuable tip about how $GRID$ is to be distributed (actually, the distribution of $GRID$ will rely on the distribution of X).

Something similar is done with LCC. This array represents the set of neighbor cubes from a given cube. The directive

```
     ALIGN (DIREC1(LCC(I,:)) WITH DIREC2(I))
```

indicates that LCC is being used as an indirection to access the elements in $DIREC1$ and thus, $DIREC1$ must be realigned taken into account this fact.

5.2 Parallelization

In codes dealing with unstructured domains, the order of data in arrays is not relevant (we have sets of elements of the same class). Then, the *compute on owner* rule is applied while distributing the work load. So, each computation is performed where the accessed elements are located. In this way, data accesses are mostly local, keeping low the communications required.

The parallelization step of loops during compilation will be very simple because each processor has only to sweep over the elements stored in local memory. The bounds of loops for each partition will be provided at run time by a simple call to the same run time system which has performed data distribution in a previous step.

However, the compiler has to check some constraints in the code before applying these simple parallelization process. It must be guaranteed that, at the time of entering a loop, all data accessed will be stored in local memory. If not, the compiler must insert code for an inpector phase before the loop to look for data out of local memory.

These constraints on the code allow the compiler to know that the work distribution in a loop will match exactly the selected data distribution, so that communication can be predicted before actually executing the loop. So, accessed to elements in distributed arrays must not be related with their position in the global array (that is, neither indexing formulas nor indexing constants are allowed. If indirection arrays are used, they must be aligned before with the accessed elements by an extended ALIGN statement. Also, in the case of using indirection arrays, the *compute on owner* rule can consider ownership of the indirection array instead of the data elements accessed through this indirection.

5.3 Code generation

As the final data distribution and the indirections for access pattern depend on the actual input data during program execution, part of the analysis must be done at run time by using a run time system developed to support irregular data distribution and manegement.

This run time system is a set of library routines designed to handle arrays as abstract objects. The library is known as **Data Distribution Layer (DDLY)** and has been developed by our research group. An early description of the interface and functionality of the library can be found in [19].

The basic capabilities of the DDLY run time support are distributing arrays following both regular and pseudo-regular distributions [1], handling of indirection arrays (complex alignment with other arrays yet distributed), gathering of non local elements, reductions over several partitions and parallel input/output between arrays in memory and disk files.

Data distribution and indirections alignment is performed after input data is available, by inserting calls to the DDLY routines which perform data analysis and distribute data following the results of that analysis.

If the parallelized code matched the code restrictions to guarantee that program access pattern complies with data access locality expected by the library, then we need no inspection of data accesses in loops. In that case, the compiler inserts a DDLY routine call before the loop to prefetch non local accesses before actual execution. The communication pattern for this operation was computed at the time data was distributed by another call to the library.

In case the compiler can not guarantee any conditions about the behaviour of data accesses, an inspector is added before loops to construct a communication schedule for execution.

6 Sparse matrix fill-in

The LU decomposition is typically the most outstanding example where the fill-in problem occurs. It is applied to the solution of linear systems, $Ax = b$, where `A` (the system matrix) is sparse and with dimensions $n \times n$. The algorithm factorizes the matrix in a product of a lower triangular matrix, `L`, by an upper triangular matrix, U, such that:

$$PAQ = LU \tag{1}$$

where the permutation matrices, P and Q, are needed due to the permutation process that takes place during the factorization in rows and columns for A, with the aim of preserving the sparsity rate and ensuring the numerical stability. Matrices L y U have also dimensions $n \times n$, but their density is bigger than the one of the A matrix because of the fill-in.

Once the system decomposition has been performed, $LUx = b$ is solved by means of forward-substitution and back-substitution stages. In general, the system resolution may be organized into four different steps:

1. **Reordering**: It realigns the matrix with the aim of reducing the complexity of its processing in subsequent stages. For instance, A can be sorted in order to produce a block triangular or block diagonal matrix, and after that, each block can be independently factorized (where the fill-in is confined).
2. **Analyze**: It chooses row and column permutations (P and Q) suitable for the factorization. These P and Q matrices will require the selection of pivot elements, which have to be chosen such that, on the one hand, we preserve the sparsity rate (applying, for instance, the Markowitz criterium), and on the other hand, the numerical stability is guaranteed (choosing those pivots greater than a certain threshold value).
3. **Factorize**: It accepts a matrix A together with recommended permutations and perform the factorization $PAQ = LU$. This may be the most consuming-time stage, since it performs the update operations for floating-point numbers.
4. **Solve**: It uses the factorization to solve the equation $Ax = b$ or $A^T x = b$. It includes the forward and backward substitution stages.

The **Analyze** and **Factorize** stages can be joined together in a single (**Analize-Factorize**) step if, for each update iteration, we perform a column and row permutation with the aim of ensuring the numerical stability and maintaining the sparsity rate as well. From now on, we only consider step 3 (**Factorize**), since it is the only one that contains fill-in.

6.1 Right-looking LU

There are several ways of organizing a LU decomposition algorithm for general sparse unsymmetric matrices [7]. Perhaps the most popular one is that named *right-looking LU* or submatrix-based method. It consists of performing n sequential iterations by means of a loop with index k. Each of these iterations chooses a pivot, permutes the matrix so that the pivot occupies the position (k, k) and finally the submatrix defined by pivot k is updated, that is, the positions (k+1:n,k:n).

The parallel algorithm can be mapped over a mesh of $P \times Q$ processors using a *scatter* data distribution and a local representation with 2-D doubly linked list. This algorithm is extensively described in [2], together with a performance evaluation on the Cray T3D.

From the point of view of an efficient computation, the dynamic memory allocation for each new element is time-consuming, and the list traversing even more. Finally, it is very well known the problem exhibited by the semiautomatic paralelization of codes when pointer-based structures are involved. Nowadays, there are no tools with an eficient handling of this kind of data structures, and the costs of development for those strategies are still under evaluation.

Therefore, the main problem that presents the LU algorithm from the point of view of the semiautomatic parallelization is the pointer-based data structure used. This structure is almost unavoidable due to the right-looking organization of the algorithm, which enforces, for each iteration, to update the whole reduced submatrix, thus requiring the fill-in handling all over the submatrix. In this way, is is necessary to traverse almost all the rows and columns in that submatrix, updating some elements and adding others. Fortunately, we can avoid this drawback reorganizing the algorithm: The right-looking implementation for the algoritm can be replaced by another called *left-looking*. This strategy is described in the subsequent section.

6.2 Left-looking LU (MA48)

As it will be seen, from the data access point of view, the left-looking variant is better than the right-looking one. This strategy, also known as column-based method (or row-based method), updates for each iteration in loop k the $k-th$ column using the $k-1$ columns previously updated.

For this kind of methods is common the use of ordinary partial pivoting, that is, the pivot is chosen from the current column to be processed, having into account just numerical considerations. This implies the Analyze phase to be given the corresponding columns preordering in such a way that the sparsity rate is guaranteed.

An outline for the left-looking algorithm is showed in Figure 7.

Note that the structure of the algorithm is divided in two different steps for each iteration:

- Symbolic factorization (line 3): Initial stage, where the nonzero structure for the column is predicted. This prediction is not more time-consuming than the numerical factorization when we use depth-first search and topological ordering, such as proven by Gilbert and Peierls [10]. This simbolic factorization was speeded-up by Eisenstat and Liu [8], who designed a pruning technique (line 10) to reduce the amount of structural information required for the symbolic factorization.
- Numerical factorization (lines 5, 8 and 9): They perform the arithmetic operations strictly needed, using the information generated by the symbolic factorization.

Figure 8 illustrates the updating process for the k-th column by two other formerly updated columns, c_{r1} and c_{r2}.

As it can be seen, this column-oriented organization, matches perfectly the second data structure explained in Section 2.3. Once every column has been updated (maybe using temporarily the *workspace* area), it will be stored on the $FACT$ and $IRNF$ vectors by properly updating the $IPTRU$ and $IPTRL$ pointers as Figure 3 showed.

However, from the point of view we are interested on, such algorithm organization will not let us to extract as much parallelism as in the right-looking version. In order to avoid a high number of communications, we should distribute rows by cyclic such as done by the BRS strategy, thus being able of updating columns in parallel.

On the other hand, skipping the use of linked lists and choosing a more static data structure instead, makes it easier the data-parallel compiler handling of this algorithm. For this purpose, new directives have to be developed for conveying information to the compiler about the distribution and representation the user chooses for the data.

As far as the fill-in operation is concerned, a static representation enforces the user to know the statements of the algorithm in which the fill-in occurs, since it is not possible to give in a vector reference the global coordinates for other elements than those already included in the matrix. For that purpose, we need to enable an intrinsic function in the data-parallel language, `FILL-IN(...)`, whose functionality lies in accepting parameters for the matrix being updated as well as the value, row, and column for the new element being added. The compiler accepts this function and is responsible for inserting within the target parallel code the corresponding run-time support for carrying out the insertion of the element in the static structure. That is, fill-in is not transparent to the user whatsoever, and this is the cost to pay for simplifying the compiler development and implementation in a more realistic manner.

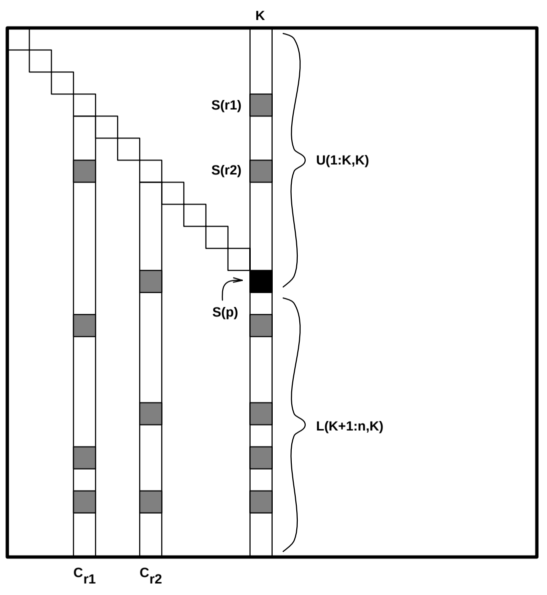


Figure 8: Column k updated by two previously updated columns.

7 Related work

There have been many efforts aimed at providing compile-time and run-time support for irregular problems such as [13, 12, 17]. Most of the research on irregular problems in Fortran has concentrated on handling single-level indirections, like the PARTI and CHAOS [16] toolkits, well known runtime libraries extensively used to parallelize irregular codes.

Based on the standard inspector-executor

```
DO k=1,n                                              ! For each column
    S = A(:,k)                                        ! Column k of A
    Symbolic factor: determine the columns $C = \{c_{r1}, c_{r2}, ..., c_{ri}\}$   $(ri < k)$ of L to update S
    DO for each $c_r \in C$ in topological order
        $S = S - S(r) \cdot c_r$
    END DO
    Pivot: interchange $S(k)$ and $S(p)$, where $|S(p)| = max|S(k:n)|$
    $U(1:k,k) = S(1:k)$
    $L(k+1:n,k) = S(k+1:n,k)/U(k,k)$
    Prune simbolic structure based on column k
END DO
```

Figure 7: Outline for the left-looking LU algorithm.

paradigm, those libraries have been used as run-time support in compiler prototypes, such as the ARF compiler [25], as well as by the Fortran 90D [9] and Vienna Fortran compilers [26]. Those compilers handle irregular problems in a generic manner, providing either a descriptor (mapping array) that describes the target processor for each element of an array individually or, using a more general concept, user-defined distribution functions [26]. Nevertheless, due to their generality and a lack of compile-time information on the way in which data is accessed, these strategies produce relatively inefficient target code and introduce a big memory overhead.

In practice, irregular application codes have complex access functions that go beyond the scope of current compilation techniques. The first attempt at dealing with multiple levels of indirection inside a compiler was by Das et. al.[5], who suggested a technique based on *program slicing* that transforms the code containing multi-level indirect references into code that contains only a single level of indirection by using multiple inspector stages. The technique used by SAR (Sparse Array Rolling) [22] is different in that it resolves multiple levels of indirection by exploiting the semantic relations between the index arrays involved in the indirect accesses.

Another approach for parallelizing sparse codes is that followed by Bick and Wijshoff [4], who have implemented a restructuring compiler which automatically transforms programs operating on dense 2-dimensional matrices into codes that operate on sparse storage schemes. During this transformation, characteristics of both the target machine as well as the nonzero structures is accounted for, so that one original dense program can be mapped to different implementations that tailored for particular instances of the same problem. This method simplifies the task of the programmer at the risk of inefficiencies that can result from not allowing the user to choose the most appropriate sparse structures.

8 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 compilation strategies without a deep orientation to real codes.

It is clear that the data-parallel compilation of irregular problems still requires a specific study of each problem, as current compilers do not provide general solutions for a broad range of algorithms.

This paper has presented a set of extensions particularly targeted to the elements involved in the specification of parallel and irregular applications. Those language elements convey additional information to the compiler that can be used to simplify its implementation as well as to produce a more efficient final code.

The information provided through annotations comes from two different sources: The data representation, which is usually well known to the user and it does not require many efforts to be provided, and the choice of a particular data distribution, which necessitates knowledge about the features of the accesses to the distributed arrays in the code if the user wants to achieve a good average performance.

The overall result is a powerful mechanism that can be used by the user to hide the low-level work when parallelizing irregular problems and exploit the locality exhibited by the application, thus leading to a target parallel code without sacrificing the performance of its execution.

References

[1] R. Asenjo, L.F. Romero, M. Ujaldon and E.L. Zapata *Sparse Block and Cyclic Data Distributions for Matrix Computations.* High Performance Computing: Technology, Methods and Applications. J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik, eds. Advances in Parallel Computing, vol. 10, pp. 359-378. Elsevier Science, North-Holland, Amsterdam. 1995.

[2] R. Asenjo and E.L. Zapata. *Sparse LU Factorization on the Cray T3D*, Int'l Conf. on High-Performance Computing and Networking Milan, Italy, May 3-5, 1995, pp. 690-696. (published by Springer-Verlag, Berlin, Germany, B. Hertzberger and G. Serazzi, Eds., LNCS no. 919).

[3] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine y Henk van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods*, Ed. SIAM, 1994.

[4] A.J.C. Bik and H.A.G. Wijshoff, *Automatic Data Structure Selection and Transformation for Sparse Matrix Computations.* IEEE Transactions on Parallel and Distributed Systems, 1995 (to appear).

[5] R. Das, J. Saltz and R. von Hanxleden. *Slicing Analysis and Indirect Access to Distributed Arrays*, Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing, Aug 1993, pp 152-168. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[6] I.S. Duff and J.K. Reid. *MA48, a Fortran Code for Direct Solution of Sparse Unsymmetric Linear Systems of Equations*, Rutherford Appleton Laboratory. Technical Report RAL-93-072. October 1993.

[7] J.J Dongarra, I.S, Duff, D.C. Sorensen and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*, Society for Industrial and Applied Mathematics, 1991.

[8] S.C. Eisenstat and J.W.H. Liu, *Exploiting Structural Symmetry in a Sparse Partial Pivoting Code*, SIAM J. Scientific and Statistical Computing, 14(1):253-257, Jan. 1993.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D language specification*, Dept of Computer Science Rice COMP TR90079, Rice University, 1991.

[10] J.R. Gilbert and T. Peierls, *Sparse Partial Pivoting in Time Proportional to Arithmetic Operations.* SIAM J. Sientific and Estatistical Computing, 9(5):862-874, Sep. 1988.

[11] HPF-2. Scope of Activities and Motivating Applications, High Performance Fortran Forum, Version 0.8, Nov. 1994.

[12] A. Krishnamurthy, D.E. Culler, A. Dusseau, S.C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. *Parallel Programming in Split-C*, Proceedings Supercomputing'93. Nov. 1993, pp 262-273.

[13] P. Mehrotra and J. Van Rosendale. *Programming distributed memory architectures using Kali.* In A. Nicolau, D. Gelernter, T. Gross and D. Padua, editors, Advances in Languages and Compilers for Parallel Processing, pp. 364-38 4. Pitman/MIT-Press, 1991.

[14] R. Mirchandaney, J. Saltz, R.M. Smith, D.M. Nicol and Kay Crowley. *Principl es of run-time support for parallel processors.* In Proceedings of the 1988 ACM International Conference on Supercomputing, pages 140-152, July, 1988.

[15] L.F. Romero and E.L. Zapata. *Data distributions for sparse matrix vector multiplication solvers.* Journal of Parallel Computing vol. 21, no. 4, April 1995, pp. 583-605.

[16] J. Saltz, R. Das, B. Moon, S. D. Sharma, Y. Hwang, R. Ponnusamy, M. Uysal *A Manual for the CHAOS Runtime Library*, Computer Science Department, Univ. of Maryland, Dec. 22, 1993.

[17] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das and J. Saltz. *Run-time and Compile-time Support for Adaptive Irregular Problems*, Proceedings Supercomputing '94, Nov. 1994, pp. 97-106.

[18] A. F. van der Stappen R. H. Bisseling and J. G. G. van de Vorst, *Parallel sparse LU decomposition on a mesh network of Transputers*, SIAM J. Matrix Anal. Appl. 14(3):853-879, Jul. 1993.

[19] G. P. Trabado and E. L. Zapata, *Exploiting Locality on Parallel Irregular Computations*, Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing, San Remo, Italy, 1995.

[20] G. P. Trabado and E. L. Zapata, *Data Parallel Languages Extension for Exploiting Locality in Irregular Problems*, Department of Computer Architecture, University of Malaga, Technical Report UMA-DAC-95/32

[21] M. Ujaldon, S. Sharma, J. Saltz and E.L. Zapata. *On the evaluation of parallelization techniques for sparse applications.* Proceedings of 10th ACM International Conference on Supercomputing. Philadelphia, May, 1996.

[22] M. Ujaldon and E.L. Zapata, *Efficient Resolution of Sparse Indirections in Data-Parallel Compilers.* Proceedings of 9th ACM International Conference on Supercomputing, Barcelona (Spain), July 1995, pp. 117-126.

[23] M. Ujaldon, E. L. Zapata, B. Chapman and H. Zima. *Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation*, Submitted to IEEE Transactions on Parallel and Distributed Systems. Also available as Technical Report 95-2, Institute for Software Technology and Parallel Systems, University of Vienna, Austria.

[24] M. Ujaldon, E. L. Zapata, B. Chapman and H. Zima. *New Data-Parallel Language Features for Sparse Matrix Computations.* Proceedings of 9th IEEE International Parallel Processing Symposium. Santa Barbara, California. April, 1995, pp. 742-749.

[25] J. Wu, R. Das, J. Saltz, H. Berryman, S. Hiranandani *Distributed Memory Compiler Design for Sparse Problems.* IEEE Transactions on Computers, Vol. 44, number 6, June 1995.

[26] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. *Vienna Fortran - A language Specification Version 1.1*, University of Vienna, ACPC-TR 92-4, March 1992.