Data-Parallel Support for Numerical Irregular Problems

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

December 1999
Technical Report No: UMA-DAC-99/03

Published in:

J. Parallel Computing
vol. 25, no. 13-14, pp. 1971-1994, December 1999

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

Data-Parallel Support for Numerical Irregular Problems ⋆

E.L. Zapata, O. Plata, R. Asenjo and G.P. Trabado

Department of Computer Architecture, University of Málaga
P.O. Box 4114, E-29080 Málaga, Spain
E-Mail: {ezapata,oscar,asenjo,guille}@ac.uma.es

Abstract

A large class of intensive numerical applications show an irregular structure, exhibiting an unpredictable runtime behavior. Two kinds of irregularity can be distinguished in these applications. First, irregular control structures, derived from the use of conditional statements on data only known at runtime. Second, irregular data structures, derived from computations involving sparse matrices, grids, trees, graphs, etc. Many of these applications exhibit a large amount of parallelism, but the above features usually make that exploiting such parallelism become a very difficult task. This paper discusses the effective parallelization of numerical irregular codes, focusing on the definition and use of data-parallel extensions to express the parallelism that they exhibit. We show that the combination of data distributions with storage structures allows to obtain efficient parallel codes. Codes dealing with sparse matrices, finite element methods and molecular dynamics simulations are taken as working examples.

1 Introduction

Scientific and engineering applications are computational expensive, and so important efforts have been devoted to find efficient implementations on multiprocessors. This general situation, however, deserves to be analyzed in more detail. There is a class of numerical applications that exhibit a regular structure. Computationally, these problems are typically characterized by the existence of a simple relationship between the indices of two data dependent array accesses (often a linear function analyzable by the compiler).

⋆ This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03 and the Human Capital and Mobility programme of the European Union under project ERB4050P1921660

Regular applications are usually easy to parallelize, making efficient use of the processor cycles and the memory hierarchy of current multiprocessor architectures. These codes make easy to devise a simple mechanism to access remote data, as regular data dependences can often be satisfied, with little or no communication, by distributing data structures across memories using simple mapping strategies (block, cyclic or mixed block-cyclic).

Many important numerical applications, however, show an irregular structure. These problems arise in sparse matrix computations, computational fluid dynamics, image processing, molecular dynamics simulations, galaxy simulations, climate modeling, optimization problems, etc. [50,37] They exhibit an irregular and unpredictable runtime behavior, that does not fit directly the architectural features of contemporary multiprocessors. This makes that writing an efficient parallel program becomes a very difficult task.

We can distinguish two kinds of irregularity in these applications. First, irregular control structures may appear in the code, that is, conditional statements on data only known at runtime. Second, irregular data structures may also appear in the code, which include sparse matrices, grids, trees, graphs, etc. Usually these data structures are organized as arrays in the code, but in such a way that arise pairs of dependent array elements whose indices appear to be related by a complex function or by an array lookup (indirection access). Both kinds of irregularities cause irregular communication patterns and workload balancing problems. Issues like data layout, data/control dependences, locality, workload balancing, communication optimization, and others, become hard problems to be assessed.

Private memory (message-passing) architectures have been the most used machines to solve large numerical problems. Typically these machines are programmed using a standard sequential language, like Fortran or C, augmented with constructs for message-passing, like PVM or MPI. In this programming model, the programmer must explicitly deal with complex aspects, like data distribution, work partitioning, task communications, workload balance, and others. This fact leads to high developing costs and slow software production.

To facilitate the productivity of parallel numerical software, two more options are being available in recent years [47,15]. On one hand, a data-parallel programming model has been standardized, the High Performance Fortran (HPF) [30,17,29,36]. This paradigm is based on a single thread of control and a globally shared address space. On the other hand, scalable distributed shared memory machines [34] are starting to be used for solving numerical applications. For these systems shared memory is the native programming model. Recently there has been an important effort to develop a new standard for portable, high level, scalable and incremental shared memory parallelization, called OpenMP [39].

In any of the above parallel programming models a good distribution of data across the memories is a key point for obtaining a scalable parallel program [42]. It would be desirable to have a compiler capable of selecting the appropriate data distribution with no intervention from the programmer. However, this is a very tough task for numerical irregular problems. Currently the problem is open and unsolved, except for some simple classes of irregular codes [10,9,2]. The compilation task can be simplified if the programmer is allowed to be involved through the use of compiler directives. These directives may help the compiler in identifying the class of irregular application, that is the role of the main data structures in the code, as well as express directly data distributions and alignments. Data-parallel and scalable shared-memory programming models permit this alternative.

This paper reviews our research experience in parallelizing numerical irregular problems in scalable parallel systems, mainly message-passing machines, in the context of a data-parallel programming model. It shows a sample of the methods we have developed in the case of three example codes, that cover a representative set of irregular applications. In all cases we explain how to use and improve the data-parallel language HPF to inform the compiler about key aspects of the program and hence obtain an efficient and scalable parallel code.

The rest of the paper is organized as follows. Section 2 discusses the difficulties in the parallelization of numerical irregular programs, and how we can obtain efficient parallel codes if the compiler is aware of some problem properties. Next two sections explain the method we follow to obtain such parallel codes for various code examples, one in the sparse matrix area (Section 3) and two dealing with unstructured domains (Section 4). In Section 5 some related work is described, and finally, we draw some conclusions.

2 Revealing Problem Structure

Numerical applications that operate on regular data structures can often be efficiently implemented by using data-parallel language constructs. The regular nature of the problem to be solved can be easily comprised at the language and compiler levels. The same cannot be said about irregular numerical applications, that use irregular data structures. It is usual that in the process of expressing an irregular application using language constructs some problem properties are lost. For instance, codes may not exhibit key properties such as spatial or temporal localities associated to the nature of the problem. It is crucial, however, to exploit this kind of properties in order to obtain efficient parallel codes.

```
DO i = 1, N
   DO j = Row(i), Row(i+1)−1
      Y(i) = Y(i) + Data(j)*X(Col(j))
   END DO
END DO
```

(a)

```
DO ts = TimeStep, TimeStep+THop−1
   DO k = 2, NeighListLength, 2
      i = NeighbourList(k−1)
      j = NeighbourList(k)
      r = distance(X(i),X(j))
      IF (r .LT. cutoff) THEN
         ff = force(r)
         F(i) = F(i) + ff
         F(j) = F(j) − ff
      END IF
   END DO
   DO i = 1,N
      V(i) = K1 * V(i) + K2 * F(i)
      X(i) = X(i) + TS * V(i)
   END DO
END DO
```

(b)

Figure 1. *Codes for sparse matrix-vector multiplication (a) and molecular dynamics simulation (b)*

There is, therefore, usually a mismatch between the *problem-level* representation of the application (data and operations needed to process the data, as well as the problem properties of such data and operations) and its *language-level* representation (code using some conventional data-parallel language). In order to obtain an efficient parallel program the compiler needs to know the role of some key data structures in the code as well as some general properties that the data stored on such structures fulfill. Let us explain this using two small code examples, as shown in figure **1**. In figure **1** (a), a sparse matrix-vector multiplication code (SpMxV) is sketched. The source and resulting vectors are dense, and are represented by arrays X and Y. The matrix, however, is sparse and it is represented using a compressed row storage format (CRS) [19]. This format uses three arrays, Data, Col and Row to represent the matrix. Data stores the nonzero entries of the matrix, as they were traversed in a row-wise fashion. Col stores the column indices of the entries in Data, and Row stores the locations in Data that start a row (by convenience, Row(n+1) stores the number of nonzero entries of the matrix plus one, where n is the number of rows of the matrix).

The SpMxV code shown in figure **1** (a) is efficient, as only nonzero matrix entries are considered for the multiplication, avoiding a large amount (depending on the matrix fill-in) of null operations. However, the nature of the problem has been lost. A simple analysis of the code only sees a two-level nested loop that operates on five *dense* one-dimensional arrays. A naive parallelization of the nested loop probably produces an inefficient parallel code, as the length of the inner loop cannot be predicted at compile-time and the array X is accessed through an indirection. Simple block/cyclic distributions of the arrays may imply an unbalanced workload distribution and a high communication overhead.

A much better parallel code can be obtained if the knowledge that the three

arrays Data, Col and Row actually represents a sparse matrix (problem property) is considered. If the compiler knows that the arrays Row and Col only contains index information about the location of the values stored in Data on a matrix space, then a uniform distribution of Data is enough to assure a well-balanced code workload distribution. In addition, a careful assignment of the particular elements in Data to the processors allows us to optimize the communication requirements. The definition of this mapping may also be facilitated if the sparse matrix nature of the problem is known. When using a data-parallel language, such data distribution and assignment must be done at the language level. Hence the sparse matrix problem property should also be at that level.

A different situation is represented in figure **1** (b). The code shown is a section of a simple molecular dynamics (MD) simulation [5]. A neighbour list (NeighbourList) was previously built by storing pairs of interacting particles. The inner loop k sweeps over those pairs and, if the separation between the interacting particles is less than some cutoff distance, pairwise force contributions on them are computed. Afterwards, velocities and positions of all the particles in the system are updated. These computations are repeated during a small number of consecutive timesteps (outermost loop ts).

The force calculation stage is usually the most time-consuming part of the simulation. Through the use of a neighbour list, only non-null force contributions are computed, speeding up this stage. However, this fact increases the algorithmic complexity of the code, as indirections appear at both sides (LHS and RHS) of the statements computing the forces. A naive distribution of the particles across the processors (block distribution of the arrays, for instance) probably produces a parallel code with high communication requirements at the most computational part of the MD simulation.

However, if only short-range interactions are considered in the simulation (problem property), then the neighbour list exhibits a high degree of spatial locality. This knowledge does not appear in the code, but only in the input data. Exhibiting this knowledge at the language/compiler level, the communication overhead can be minimized. For instance, a spatial decomposition of the domain can be accomplished. Communication requirements are very low, as only particles near the frontier of the sub-domains interact with non-local particles (belonging to other sub-domains).

There are many parallel implementations of the above two codes, and of many other numerical irregular applications. Most of these implementations, in the case of large applications at least, has been done on massively parallel computers, resorting to message-passing libraries to deal with communication. All these parallel codes take advantage, in same way, of problem properties, such as the ones discussed above, in order to get efficiency. However, data-parallel

language constructs are less flexible than message-passing routines. The execution model is single-threaded with a globally shared address space. With these languages we have similar difficulties to expose problem properties than with a sequential language. While there are many efficient message-passing irregular numerical codes, for many problems, however similar implementations using a data-parallel language is currently an open question.

The approach we follow consists in defining new data-parallel language constructs in the context of HPF that permit to identify the role of the important data structures in the code, to express exploitable key properties of the problem nature (data properties) and to manipulate irregular data structures. The compiler is in turn instructed to recognize the new constructs to use them when producing the parallel code. For large applications, the complexity of the compiler may be lightened by leaving most of the work associated with data distribution, communication and synchronization to an external runtime library. The rest of the paper is dedicated to show the proposed solutions for a number of representative numerical irregular codes.

3 Sparse Matrix Algebra

Sparse matrices arise naturally when discretizing continuous operators. They appear in a large number of important scientific and engineering codes, which are classified as irregular applications [59,19,33]. This is because sparse matrices are represented using compact data formats, which necessitates heavy use of indirect addressing through pointers stored in index arrays. Such compact formats are used in order to not store zero elements, saving memory and avoiding arithmetic with zero elements to save floating-point operations.

Normally, it can be distinguished two broad categories of sparse-matrix representations [59]. *Structured representations*, that exploit some regular structure of the fill (set of locations of the nonzero elements), and *unstructured representations*, that do not impose any requirements to the fill. However, in this sections and the next one, we will consider an operative classification of sparse-matrix applications [19]. We will use the term *static* to comprise those applications that only read the sparse-matrix data. That is, the fill of the matrix does not change during computation. The other class is called *dynamic*, including those applications that read and write the sparse-matrix data. In this case, the fill of the matrix may change during computation.

An efficient parallelization of a sparse-matrix application requires exploitation of data locality and a simple and fast method to locate non-local data. Most parallel implementations are tuned according to the fill pattern of the sparse matrix, in order to attain maximum efficiency [33]. According to this, some

sparse compact representation is more appropriate than others to get the best efficiency.

In a data-parallel environment, such as HPF, we are constrained to the language constructs of a sequential language, such as Fortran90/95, and some directives used to express parallelism and to instruct the compiler on how to produce the parallel code. Currently HPF does not include any directive to link the arrays of the sparse-matrix representation to the sparse matrix itself [30]. However, having such directive permit to define sparse data distributions with the property of, on the one hand, exploiting data locality and, on the other hand, simplifying the location of non-local data. We have proposed the directive SPARSE to fill this gap [4,46]. For instance, for the example shown in figure **1** (a), we would write

```
!HPF$ SPARSE(CRS(Data,Col,Row)) :: A(N,N)
```

to express that Data, Col and Row are the arrays associated with the CRS representation of a sparse matrix. In addition, the sparse matrix is giving a name, A in the example shown. This name is actually a template, because it does not exist in the original sequential code. We can use it only inside other HPF directives. The sparse name can be referenced in a distribute directive. For example,

```
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) :: A
```

specifies that the sparse matrix A should have its rows and columns distributed cyclically over some two-dimensional arrangement of processors. Observe that A names a *complete* (uncompressed) sparse matrix. Therefore, both nonzero and zero elements are considered for distribution. As a consequence of this specification, the arrays Data, Col and Row will be mapped to the processors in such a way that the above distribute directive is obeyed. This kind of sparse distribution was called BRS (Block Row Scatter) in [46,4].

For mapping purposes, the sparse matrix is dealt as a dense matrix. Hence the workload is balanced using similar techniques than if the code were dense. Besides the compiler can derive simple formulas to determine the location of each sparse data entry. Communication schedules are easy to compute and optimize, with low overhead.

The above BRS pseudo-regular sparse data distribution can be used to obtain efficient parallel codes using static sparse matrices in an efficient way. For instance, [6] shows a simple parallel implementation of the Conjugate Gradient iterative method for solving large sparse systems of linear equations. A complete discussion of static sparse computations on the framework of the Vienna Fortran compilation system can be found in [58]. BRS (and MRD, Multiple Recursive Decomposition, discussed in the next Section) distribu-

```
DO k = 1, n
   Find pivot: A_ij
   IF (i  .NE. k)
      swap A(k,1:n) and A(i,1:n)
   END IF
   IF (j  .NE. k)
      swap A(1:n,k) and A(1:n,j)
   END IF
   A(k+1:n,k) = A(k+1:n,k) / A(k,k)
   DO j = k+1, n
      DO i = k+1, n
         A(i,j) = A(i,j) - A(i,k)A(k,j)
      END DO
   END DO
END DO
```

Figure 2. *Direct right-looking LU algorithm*

tions are described in detail and compilation and runtime issues related to their implementation are analyzed.

For dynamic sparse-matrix applications, the situation is much more complex. The explained approach must be extended to take into account that the fill of the matrix may change while computing. Our solution is to define a dynamic compact format to represent the sparse matrix, keeping charge of the possible fill-in. The rest of the section is dedicated to discuss in detail a dynamic case study.

3.1 Dynamic Case Study: Direct Sparse Solvers

A wide range of numerical applications include the solution of large sparse systems of linear equations. There are two different approaches to solve such systems, direct and iterative methods. In direct methods,[19,24] the system is converted into an equivalent one whose solution is easier to determine by applying a number of elementary row and/or column operations to the coefficient matrix. A different approach is taken in iterative methods[7,60], where successive approximations to obtain more accurate solutions are carried out.

Direct methods for solving sparse systems exhibit different problems to those of iterative methods. The coefficient sparse matrix is transformed, or factorized, operation that may change the fill of the matrix. The compact representation of the matrix must take into consideration this fact. Also, row and/or column permutations of the coefficient matrix (pivoting) are usually accomplished in order to assure numerical stability and reduce fill-in. All these features make direct methods much harder to parallelize than iterative solvers

Let us take the sparse LU factorization as a working example [19]. Figure **2** shows an in-place code for the direct right-looking LU algorithm, where a

```
!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
```

Figure 3. *HPF declarative section for the sparse LU code*

n-by-n matrix A is factorized. The code includes a row and column pivoting operation (full pivoting) to provide numerical stability an preserve the sparsity rate.

Apart from linking the compact representation arrays to the sparse nature of the problem, as was seen previously with the SPARSE directive, an additional effort is needed to select the appropriate compact format. Standard compressed formats are not the most suitable schemes to support efficiently the fill-in problem and pivoting operations. *Dynamic* formats, such as linked lists, are more flexible that standard *static* formats, such as CRS. Pivoting operations are accomplished by just interchanging pointer values. Fill-in is easily managed, as entry insertion/deletion operations are simple when using linked lists.

Full-pivoting sparse LU decomposition can be represented by two-dimensional doubly linked lists, in order to make data accesses both by rows and by columns. Each item in such a dynamic structure stores not only the nonzero 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 simplified if the full pivoting is replaced by a partial pivoting, where only columns (or rows) are swapped. This fact may imply large memory and computation savings because we can use a one-dimensional doubly linked list structure as the compact representation. In such structure, each linked list represents one column (row) of the sparse matrix, where its nonzero elements are arranged in growing order of the row (column) index. Each item of the list stores the row (column) index, the matrix nonzero entry and two pointers. These compact representations were called LLRCS (Linked List Row-Column Storage) and LLCS (Linked List Column Storage) (or LLRS for rows) in [54,3].

Figure **3** depicts an HPF declarative section for the parallel sparse LU code (only the HPF directives are shown). Matrix A is defined as sparse and represented by the LLCS format. The arrays of pointers first and last indicate the first and the last nonzero entry, respectively, of each column of A. The array vsize stores the number of nonzero entries on each column of A. We have also defined a sparse array VCOL, which is represented as a compressed (packed) vector (CVS format). This array will contain the normalized pivot column of A, calculated in each outer iteration of the algorithm.

```
!HPF$ INDEPENDENT, NEW (aux,i,amul,product)
loopj:  DO j = k+1, n
!HPF$ ON HOME (vpiv(j)), RESIDENT (A(*,j)) 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
                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
              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
```

Figure 4. *Outline for an extended HPF-2 specification of the submatrix update phase of the right-looking partial pivoting LU 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 (linear). 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 fashion over the processors. Finally, the sparse array VCOL is aligned with the rows of A. Hence, after distributing A, VCOL is completely replicated over all the processors. The reason of this replication is to make normalized pivot column visible to all processors, enabling the subsequent parallel submatrix update.

As sparse data is accessed through pointers, the default owner compute rule is not the most appropriate work distribution scheme for some phases of the computation. For instance, during the update submatrix phase (loops j and i) in figure **2**), we want that processor that owns entry j of the pivot row (vpiv()) be in charge of updating row i of the submatrix. The approved HPF-2 extension ON permits such computation assignment. The approved extension RESIDENT is also used to assert that column j of A is stored in the same processor that owns vpiv(j), hence no interprocessor data movement is required (see figure **4**).

An implementation of the above code was done on a Cray T3E platform. Due to the lack of a complete HPF-2 compiler, all directives were manually translated into calls to the DDLY (Data Distribution LaYer) runtime library [56]. The code was written in Fortran90, and the Cray SHMEM library was used for communications. Parallel execution times and speed-up of this implementa-

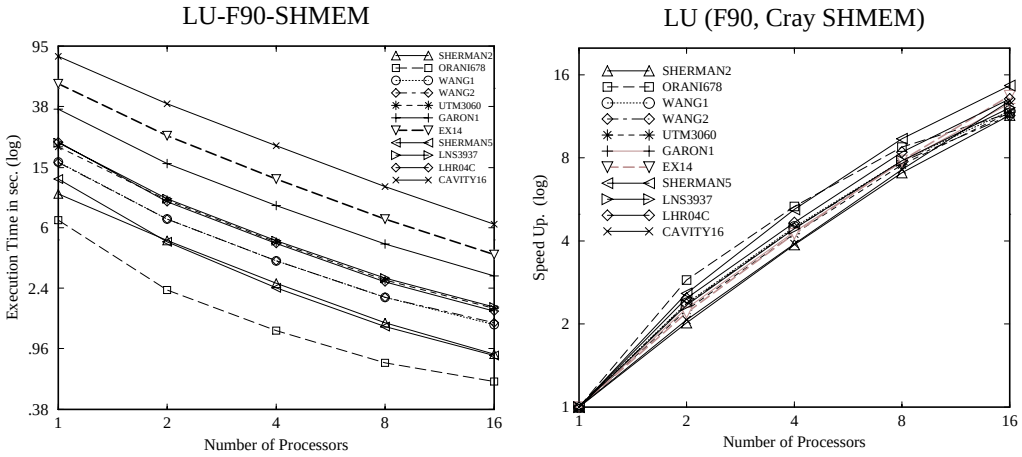


Figure 5. *Parallel sparse LU execution times and speed-up for different Harwell-Boeing sparse matrices, using F90 linked lists and Cray T3E SHMEM*

tion is shown in figure **5**, taking the sparse matrices from the Harwell-Boeing suite [20] and the University of Florida Sparse Matrix Collection [18]. The efficiency of the parallel code is high when the size of the input matrix is significantly large. Due to fill-in, some load imbalances may appear during factorization, but they were not significant in any of our experiments. An in-depth description of the sparse LU factorization can be found in [1,3,54].

4 Unstructured Domains

Many scientific and engineering applications work with unstructured domains to provide a general solver being able to handle any problem input to the application. In other case scientists should write specific solvers for the structure of each problem. However, while providing a useful tool for scientists, general solvers represent a challenge for high-performance implementation. Some representative examples include finite elements or differences for solving partial differential equation systems, computational fluid dynamics, or molecular dynamics simulations.

Finite element and difference methods are computational tools for deriving approximate solutions to a system of partial differential equations, that governs the behaviour of some physical system. In order to solve numerically the system, the physical domain is discretizing imposing a grid. In case of finite differences, the grid is usually regular. In general, at each grid point a linear equation that relates the value of some physical magnitude at such point to its value at neighbouring grid points has to be solved. The resulting sparse coefficient matrix of the full equation system has block diagonal fill structure, and can be solved using some iterative or direct solver, as was discussed in the previous Section. However, new difficulties arise if the grid is dynamic, as

in molecular dynamics simulations.

In case of finite elements, the grid divides the domain into a large number of small areas called elements, that comprise sets of interacting grid points. In addition, for most applications, the grid is not a regular structure, deriving a large, sparse and fill unstructured coefficient matrix.

These applications contain irregular data access patterns related to unstructured problem domains. Such common feature poses hard difficulties for automatic parallelization or data-parallel implementation. Many types of unstructured problems have been succesfully parallelized [33,37,23]. However, in most cases the software has been difficult to write. In addition, expressing the irregularities and efficient mapping onto particular multiprocessors has implied the development of tedious nonportable code.

In terms of high-performance implementation, these applications can be dealt as a sparse matrix problem, but with the additional difficulty that the sparse coefficient matrix is often unstructured. A different and more usual approach consists in directly handling the finite element grid. The grid is irregular but the computations fulfill locality properties.

As was discussed in section 2, obtaining an efficient data-parallel implementation of unstructured application needs to express some problem properties at the language and/or compiler level. Codes rely blindly on some simple language constructs that describes data access patterns, but not the high-level structure of the problem (computations). Data access patterns are irregular, however problem structure tends to be regular, exhibiting spatial and/or temporal localities. Compilers simply do not take advantage of such inherent regular properties.

We will explain how to extend the SPARSE HPF directive described in the previous Section to not only relate language data structures to problem data structures, but also to express *regularities* inherent to the nature of the problem. Such discussion will be done for two example cases, a static unstructured domain application (data is only read), finite element codes, and a dynamic unstructured domain application (data is read and written), molecular dynamics simulations.

4.1 Static Case Study: Finite Element Codes

In most programs handling unstructured meshes we can find pieces of code similar in structure to the sample code in figure **6**. This loop actually represents a matrix-vector product PROP2 = WEIGHT × PROP1. Arrays L, R, U and D represent the non zero weights in matrix WEIGHT. The use of such arrays and

```
INTEGER NEIGHBOR(NE,4)
INTEGER PROP1(NE+1),PROP2(NE),L(NE),R(NE),U(NE),D(NE)
...
DO i=1,NE
   PROP2(i)=PROP1(i)
         -L(i)*PROP1(NEIGHBOR(i,1))
         -R(i)*PROP1(NEIGHBOR(i,2))
         -U(i)*PROP1(NEIGHBOR(i,3))
         -D(i)*PROP1(NEIGHBOR(i,4))
END DO
```

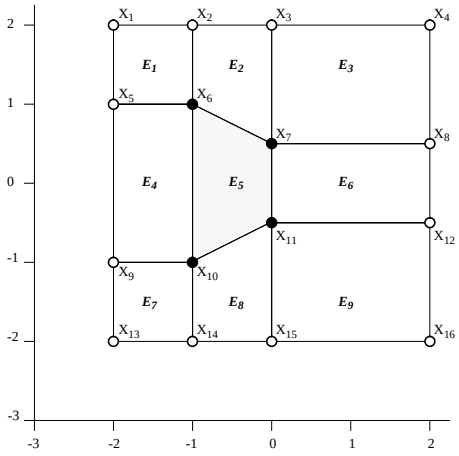
Figure 6. *Example piece of code handling a unstructured mesh*

the indexing array NEIGHBOR is due to the fact that this is an sparse problem and the programmer is optimizing the representation of the problem by using indirections instead of using one two-dimensional array for the sparse matrix WEIGHT. Each element in arrays PROP1 and PROP2 is an associated value corresponding to one *mesh element* in the physical model. So, only *physically* neighboring *mesh elements* are considered in the matrix-vector product.

From this code we can extract two observations that will become important while parallelizing unstructured meshes applications. First, the code performs a regular computation stencil which includes a limited number of array elements. Those elements correspond only to *physically* neighboring *mesh elements* which are statically defined in the input data of the program . Specifically in the contents of the array NEIGHBOR which is related with the structure of the problem mesh. Second, the term *physically neighboring* is not related with *storage neighboring. Mesh elements* adjacent in the space may not be stored contiguously in a memory array representing such mesh. Although the computation stencil is regular, speaking in terms of code analysis, the presented sample code is highly irregular because of the use of an array for indexing another one. Compilation analysis finds that accesses to array PROP1 are unpredictable because they depend on the run time contents of array NEIGHBOR.

The example solver shown in figure **6** uses only the array NEIGHBOR to direct computations through the problem domain. Nevertheless this solver does not use all existing information describing the unstructured mesh. Some other structures (arrays) are often used in external partitioners to compute data distributions. Figure **7** shows in which way an unstructured mesh is described by X, GRID and NEIGHBOR arrays.

The objective is, inside a data-parallel environment, to distribute this representation of the unstructured mesh preserving spatial locality in all the levels of the representation. The vertices of the mesh are stored in X and hence this array is the key data structure to express at the language level the spatial locality that exhibits the problem. As all data accesses in figure **6** are local at the mesh domain, we should distribute array X over the processors in such a



	X(NV,2)	
1	-2	2
2	-1	2
3	0	2
4	2	2
5	-2	1
6	-1	1
7	0	0.5
8	2	0.5
9	-2	-1
10	-1	-1
11	0	-0.5
12	2	-0.5
13	-2	-2
14	-1	-2
15	0	-2
16	2	-2

	GRID(NE,4)			
1	1	2	5	6
2	2	3	6	7
3	3	4	7	8
4	5	6	9	10
5	6	7	10	11
6	7	8	11	12
7	9	10	13	14
8	10	11	14	15
9	11	12	15	16

	NB(NE,4)			
1	10	2	10	4
2	1	3	10	5
3	2	10	10	6
4	10	5	1	7
5	4	6	2	8
6	10	5	3	9
7	10	8	4	10
8	7	9	5	10
9	8	10	6	10

Figure 7. *Up: Actual structure of the problem domain. Down:* X *stores coordinates for mesh vertices,* GRID *stores the definition of the mesh itself (each element is a set of 4 vertices),* NB *stores the list of neighbors for each element of the mesh.*

way that a spatial decomposition of the mesh is accomplished. This mapping can be expressed by extending the SPARSE directive as follows,

```
!HPF$ REAL, DYNAMIC, SPARSE(COORDINATES(X(NV,2)) :: A(:,:)
!HPF$ DISTRIBUTE (BLOCK,BLOCK):: A
```

Instead of specifying a number of arrays and a compact representation, as with sparse problems, the directive instructs the compiler to consider the indicated array (X) as a set of coordinates of spatial locations. The defined template A represents such spatial domain, and hence is a construct directly related to the mesh. The BLOCK distribution directive applied to the assigned *placeholder* A results in a spatial decomposition of the array X into subdomains, and the mapping of such subdomains over the processors. This type of decomposition was called MRD (Multiple Recursive Decomposition) in [4,46]. Observe that the distribution operation is executed at runtime, when the contents of X is known.

The next step is distributing other data structures representing higher levels in the mesh (like edges, polygons, cubes or other kind of mesh elements). Such *mesh elements* contain no geometric information but each one is related to some lower level structures (like vertices) already distributed in the former phase. These distributions can be written as follows,

```
!HPF$ REALIGN GRID(I,*) WITH X(GRID(I,1),:)
!HPF$ REALIGN NEIGHBOR(I,*) WITH GRID(I,:)
!HPF$ REALIGN PROP1(I,*), PROP2(I,*), L(I,*), R(I,*),
!HPF$+ U(I,*), D(I,*) WITH GRID(I,:)
!HPF$ REALIGN PROP1(NEIGHBOR(I,:)) WITH NEIGHBOR(I,:)
```

The first alignment is an extension of the standard HPF directive, and was called *pointer alignment* in [57,52]. This directive distributes an indirection array in such way that each row of the array is stored in the processor owning the row in some other array referenced by the pointer. Specifically, each ith-row of array GRID has been aligned with the row of array X pointed by GRID(I,1). The resulting distribution groups in each partition the elements of the array corresponding to closely located mesh elements. The next two alignment directives places several arrays in such a way that spatial locality is exploited. These are standard directives. The last directive is also a new extension of the standard, that we call *target alignment*. It is possible that, as consequence of the previous distributions, some partitioned arrays point to nonlocal array elements. Accesses using such indices can not be translated to local coordinates neither executed without fetching remote data before. But we can not fetch data before knowing which arrays are going to be accessed throught those indirections. The *target alignment* specifies that array PROP1 is going to be accessed through the indirections stored in array NEIGHBOR. This way, the compiler is instructed on how to grab needed nonlocal data at runtime.

Some evaluation tests have been carried out using a real code of 3-D computational fluid dynamics (*CFD*). The code was parallelized rewriting the HPF directives inserted in it by calls to the DDLY library. The executions presented in figure 8 were conducted in a IBM SP2 multicomputer. A PVM implementation of the DDLY library was used. The test data consisted in a mesh with approximately 276,000 cells. The solver computed 340 iterations until convergence was reached.

I/O and distribution time are the initialization phase of the program. The test yields excellent results for the execution of the solver loop, which indicate that the exploitation of the spatial locality through the use of the SPARSE directive and *indirection alignments* is nearly optimal. Also, the time spent in communications during the computation remains very low. A more detailed analysis of parallel finite element codes can be found in [52,57].

Nodes	CPU	Reduce	Scatter	Dist/Align	I/O	Total
1	*71.94*	*0.01*	*0.01*	*3.09*	*15.58*	*90.63*
2	*32.20*	*1.99*	*0.37*	*3.04*	*14.78*	*52.37*
3	*19.75*	*1.38*	*0.20*	*3.21*	*14.79*	*39.33*
4	*14.24*	*0.86*	*0.22*	*3.41*	*14.86*	*33.59*
5	*11.32*	*1.37*	*0.23*	*3.68*	*14.82*	*31.42*
6	*9.41*	*1.12*	*0.22*	*3.41*	*14.12*	*28.29*
7	*8.00*	*1.26*	*0.37*	*3.51*	*14.05*	*27.19*
8	*6.99*	*1.23*	*0.22*	*3.62*	*14.04*	*26.09*

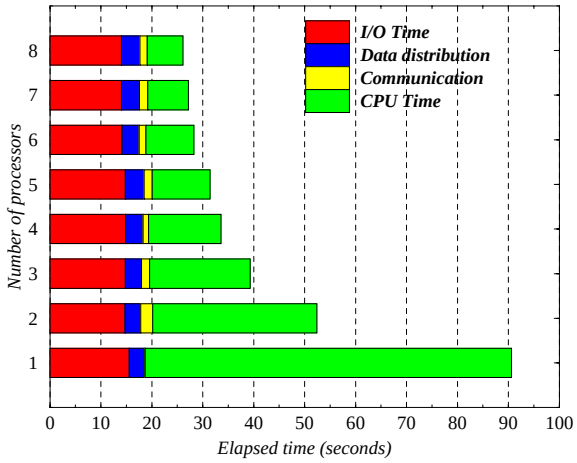


Figure 8. *Decomposition of execution times for GCCG program parallelized with the DDLY library on the IBM/SP2*

4.2 Dynamic Case Study: Molecular Dynamics

Molecular dynamics (MD) simulations consider a set of particles (atoms or molecules) subject to the Newton's classical equations of motion, which are integrated to compute and understand the movement of each particle. With this technique a number of microscopic and macroscopic values can be estimated, such as transport and diffusion coefficients, phase diagrams, and so on, many of them difficult to obtain experimentally.

To simplify the exposition we will consider the simulation of a two-dimensional ensemble of particles subject to a Lennard-Jones potential, $v^{LJ}(r) = 4\epsilon((\sigma/r)^{12} - (\sigma/r)^6)$, where the cutoff distance is $r_c = 2.5\sigma$. To integrate the equations of motion of the particles, a finite-difference leapfrog algorithm on a Nosé-Hoover thermostat dynamics is used.

Figure **9** shows the algorithmic structure of such simple short-range MD simulation. The first action is reading particle positions and velocities (*Read Data*). Some other quantities are also read, such as the size of the simulation box, the cutoff distance of the potential, the number of timesteps, etc. After some normalizations and simulation box adjustments (*Initialization*), a loop running over the timesteps is started. This loop is strip-mined into two (not perfectly) nested loops. The outer loop steps between strips of the same size (THop),

```
Read Data
Initialization
DO TimeStep = 1, NumTimeSteps, THop
   Link-Cell Building
   Neighbor List Building
   DO ts = TimeStep, TimeStep+THop−1
      Force Calculation
      Velocities and Positions Updating
      Macroscopic Parameters Calculation
   END DO
END DO
Write Data
```

Figure 9. *Algorithmic structure of a simple short-range MD simulation.*

and the inner loop steps between single timesteps within a strip.

The outer loop is used to determine at which timestep the link-cell and the neighbor list are built. These data structures are used as book-keeping methods, that avoid to waste CPU cycles in calculating null force contributions on each particle. For each iteration of that loop particles are sorted into the link-cell lattice, and all the pairs of neighboring particles (separated by a distance somewhat larger than the potential cutoff) are computed and stored in a list. To avoid border effects periodic boundary conditions are implemented at the link-cell level.

The inner loop reuses the neighbor list during a strip of iterations (THop). Force on each particle is calculated by integrating the discretized equations system, taking the interacting particles from the neighbor list. These computations are halved by resorting to the Newton's third law. Afterwards velocities and positions are updated for each particle and some macroscopic parameters are calculated also. A more detailed description of this phase was shown in figure **1** (b). Finally, after exhausting all timesteps, positions and velocities of all particles are written in a file, in addition to some macroscopic parameters.

MD codes exhibit a natural parallelism that comes from the fact that the force calculations, and the velocity/position updates, can be done simultaneously for all particles. To exploit this parallelism, a number of partitioning and mapping strategies has been developed. The approach we followed is termed *domain decomposition* [14,40], as shown in figure **10**.

An initial block-wise partitioning for the input data to be read is assumed. Each processor reads the input data (particle positions/velocities and input parameters) for its assigned initial partition. At the first timestep, link-cell lattices are built in parallel. A processor will have non-null data only in those cells belonging to its partition. Afterwards a hierarchical partitioning strategy is executed, similar to MRD. The simulation box is first partitioned at the link-cell level along the first dimension. Each strip is then sub-partitioned independently along the second dimension. In general, the number of par-

```
Read Data
Initialization
DO TimeStep = 1, NumTimeSteps, THop
   Link-Cell Building
   IF (TimeStep .EQ. 1) THEN
      Hierarchical Domain Decomposition
      Array Shuffle Redistribution
      GPU Particle Schedule Building
      Guest Particle Update (GPU)
      Link-Cell Updating
   ELSE
      POA Particle Schedule Building
      Particle Ownership Adjustment (POA)
      GPU Particle Schedule Building
      Guest Particle Update (GPU)
      Link-Cell Updating
   END IF
   Neighbor List Building
   DO ts = TimeStep, TimeStep+THop−1
      Force Calculation
      Velocities and Positions Updating
      Macroscopic Parameters Calculation
      Guest Particle Update (GPU)
   END DO
END DO
Write Data
```

Figure 10. *Algorithmic structure of a simple parallel short-range MD simulation.*

tition levels equals the number of dimensions of the box. The divisions are accomplished in such a way that some performance parameter is optimized. The imbalance in the number of particles is the most common parameter, but some other can be chosen, such as the imbalance in the number of interactions (density). The partitioner is internal, general enough to cover most cases with average efficiency sufficiently high. Besides, the partitioner generates a compact representation of the decomposition, that permits to optimize subsequent computation stages (mapping arrays are avoided).

After cell-decomposition of the simulation box, a shuffle redistribution of all data arrays (in particular, position and velocity arrays for the particles) is carried out. Each processor will store locally positions/velocities for the particles belonging to its cell-domain. The processors will also store a copy of the particle positions/velocities arranged at nearest neighbor link-cells. This communication interchange between neighbor processors is accomplished by the *guest particle update* (GPU) stage in figure **10**. Previously, a GPU particle schedule is built, an array that indicates what processors will receive particles from a particular set of border cells. GPU stage guaranties that all accessed data by a processor is locally available, either as owned data or guest data. As a consequence, all computations can be carried out using only local indices, without any type of control over memory references.

Each cell-domain has the same structure as the original program. A processor can operate on its domain, with no interaction with other processor, until the next synchronization point. A runtime library carries out all synchronization

work, in a transparent way for the user. At each synchronization point, the library updates the guest particles across domains (GPU), reusing the first schedule computed (if the cell-domain is not repartitioned, the schedule array does not change). The Newton's third law allow us to replicate force calculations instead of propagating non-local writes, saving communication overhead.

Due to the dynamic nature of the simulation, some particles may change of cell-domain. Instead of carrying out global redistributions, we have implemented a particle migration stage, called *particle ownership adjustment* (POA), that communicate to the new owner (always a neighboring processor) those particles that have abandoned its cell-domain. During this stage, the communication schedule for the GPU is also updated.

To exploit the problem spatial locality exhibited by the MD code, the extended SPARSE directive as defined in subsection 4.1 is used. For example,

```
!HPF$ REAL, DYNAMIC,
!HPF$+ SPARSE(
!HPF$+ COORDINATES(X(N),Y(N)),
!HPF$+ THRESHOLD=R
!HPF$+ LINKCELL(IX,IY)(LCPNT,LCCNT)
!HPF$+ ) :: B(0:LX,0:LY)
```

describes a set of N particles with 2D coordinates. Problem domain borders are defined by the intervals [0,LX] and [0,LY]. Note that the dimensions of the placeholder specify real instead of integer bounds because it is not defining an integer index space. The keyword COORDINATES specifies a list with the tables containing geometric information about the elements during program execution. In the particular example, arrays X and Y store the 2D coordinates of the spatial domain.

THRESHOLD controls the behavior of data replication for selected data distributions. It is used to provide an expression for the length of the overlapping area between any neighboring domains of data distribution. Replication guarantees that all particles of neighbor partitions that can be involved in a local computation will be available in local memory as a *guest element*.

The LINKCELL keyword describes the implementation of the link-cell used in the program. The first argument is the number of cells considered during execution. The second is a list with the arrays which contain link-cell data. We support both linked-lists and dense pointer-arrays implementations as they are the most popular.

The representations of particle sets in Fortran programs use to be very heterogeneous. The set of data arrays related to particles depends on what is done in each simulation, although the arrays containing geometric data are

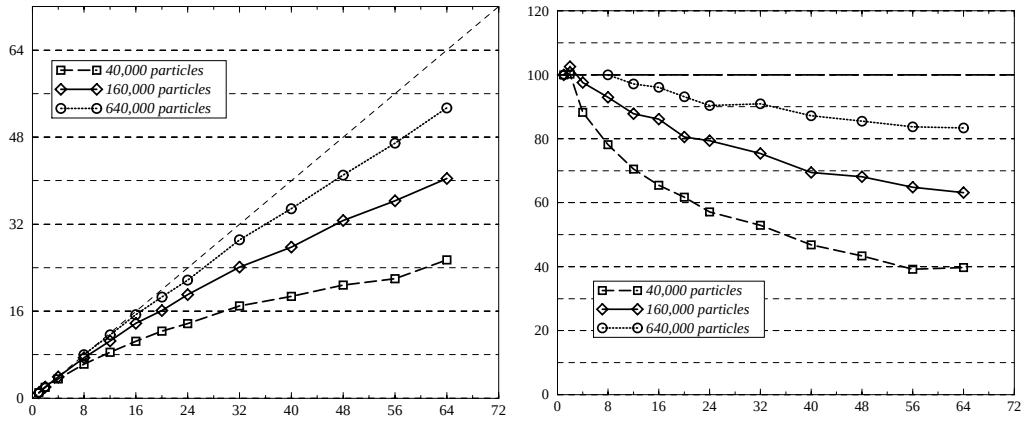


Figure 11. *Speedup and efficiency of the NBF simulation tested on the Cray T3E for three different particle sets.*

always present. For sake of simplicity in directives, the COORDINATES keyword specifies only those arrays, while the other related arrays are specified using standard HPF ALIGN directive with perfect alignment, as for example,

```
!HPF$ ALIGN WITH X :: VX, VY, FX, FY
!HPF$ REDISTRIBUTE(BLOCK,BLOCK) :: B
```

Finally, the placeholder of the problem domain, B must be distributed, using the standard directive. The BLOCK distribution method used in each dimension of the placeholder indicates geometric partitioning (i.e. a *value-based* distribution) of the set of arrays bound to the placeholder via SPARSE and ALIGN directives.

A DDLY-based implementation of the described MD code was executed on a Cray T3E, with different number of particles (domain size). Figure **11** shows the speedups and efficiencies obtained on the Cray T3E for each problem size. The neighbour list was updated every 10 timesteps. The cutoff distance was fixed to 2.5σ and the size of each link-cell was 2.9σ. The size of the simulation box was $250\sigma \times 250\sigma$. Each test has simulated 1000 timesteps using different number of processors ranging from 1 to 64. In the case of the 640,000 particle data set, we have considered speedup 1.0 for 8 processors, as this was the minimum number of processors required to execute the simulation.

We have to remark the good scalability of the parallel code. Efficiency gets closer to the ideal full parallel program as problem size increases because of the small time spent in communications (almost constant for different problem sizes and numbers of processors). The behaviour remains stable as dynamic problems evolve. In [55,52] a detailed discussion of parallel molecular dynamics codes can be found.

5 Related Work

We have distingyuished two different approaches to handle irregular computations. One possibility is based on extending the data-parallel language with new constructs suitable to express non-structured parallelism. With this information, the compiler can perform a number of optimizations, usually embedding the rest of them into a runtime library. In Fortran D [22], for instance, the programmer can specify a mapping of array elements to processors using another array. Vienna-Fortran [61], on the other hand, lets programmers define functions to specify irregular distributions. HPF-2 [30] 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.

All these language constructs usually lead to low efficiencies when they are applied to a wide set of irregular codes. The main drawback is that localing non-local data is in general an expensive task. Our SPARSE directive, however, allows to specify pseudo-regular distributions, that do not suffer from this problem.

The second approach is based on runtime techiques, 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 [38,42,12].

This technique consists in the insertion of runtime support in the final parallel code to analyze each access and decide if the access is local or remote. In the case of a remote access, remote data is sought and stored locally by the runtime support before the data access is allowed to proceed. The authors produced a runtime library known as PARTI/CHAOS [44,45] that was used in manual parallelization of irregular programs. The introduction of this paradigm in parallel compilation was carried out, for instance, at Rice University in the Fortran D compilation system [27,28,53].

The Fortran D compiler base domain partitioning on an user-provided function that map array elements to processors depending on their values. Usually, value-based decompositions [26,43] cannot be represented in a compact form, thus a mapping array must be returned by the partitioner. This solution is expensive and inviable for large systems. Also, this distribution method does not guaranty that all accessed data is locally available during all computation stages. As a consequence, inspector code is required to collect references to non-local data. Work distribution for the generated parallel code is derived

directly from the computed data decomposition. Although parallel code uses local indices when accessing arrays, indirection data structures keep containing indices pointing to the original global arrays. The presence of indirections through these data structures implies the use of a translation scheme to find out the current location of the requested data (processor + offset). Such translation scheme has a high overhead in memory or communications, as usually involves a problem-size translation table. Communication schedule for gathering non-local data is generated from the above operations.

Our parallelization method, however, is based on a provatization model included on the data-parallel framework. Data that represent the domain of the problem is partitioned using value-based decompositions (domain decompositions) trying to minimize the intractions among the subdomains. Those interactions (references to non-local data) are dealt transparently to the code by using overlapping areas, where the referenced non-local data elements are stored locally.

Opposite to the inspector-executor strategy, the computation of the overlapping areas is not based on the analysis at runtime of the data references in the code. Overlapping areas are precalculated at compile-time from the information about problem properties introduced in teh code through the directive SPARSE, as well as the data distribution specified through the standard DISTRIBUTE directive. Overlapping areas are finally computed in completion during the data distribution phase, when all neede runtime data is known. All irregular computations are executed using the owner compute rule. However, if the data distribution implies non local writes, these operations are replicated. All processors but the owner of the data element writes on the copy in their overlapping areas. These writes will be simply discarded. This way there is no need to propagate the non-local write operations and the semantics of the original application is maintained.

Finally, for sparse matrix computations there is a different approach proposed in the literature [8,31], based on the automatic transformation of a dense program, annotated with sparse directives, into a semantically equivalent sparse code. They show this approach on simple and static sparse codes. However, it is not clear its feasibility for more complex programs, like dynamic sparse computations (as in the LU) or simply the transposition of a sparse matrix. The design of such sparse compiler os very complex, in such a way that no implementation of it is available for general and real problems.

6 Conclusions

Numerical irregular applications show subscripted subscripts in their code, usually introduced by the programmer in order to reduce computation overhead and optimize the use of the memory hierarchy. These complexities in the code often hide to the compiler spatial and temporal localities inherent to the application.

Exposing the key data structures in the application as well as some problem properties that the data fulfill, the compiler can analyze the code, exploit data locality and generate efficient parallel code.

This paper describes a number of different applications that show the generality of the method discussed, and that HPF can be used to program efficiently these applications with only a small number of extensions.

Acknowledgements

We gratefully thank L.F. Romero, G. Bandera and M. Ujaldón, at the Dept. of Computer Architecture, University of Málaga, Spain, and R. Doallo and J. Touriño, at the Dept. of Electronics and Computation, University of La Coruña, Spain, for the development of some of the methods and experiments described in this paper.

We also thank Ian Duff and all members in the parallel algorihm team at CERFARCS, Tolouse (France), for their kindly help and collaboration, as well as Søren Toxvaerd, at the Dept. of Chemistry, University of Copenhagen, for providing us the example short-range molecular dynamics program.

Finally, we thank the CIEMAT (Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas), Spain, for giving us access to their Cray T3E multiprocessor, the CEPBA (Centro de Paralelismo de Barcelona), Spain, for the use of their IBM-SP2 machine, and the Forschungszentrum Jülich, Germany, for the access to their Cray T3E system.

References

[1] R. Asenjo, LU Factorization of Sparse Matrices on Multiprocessors, *PhD Dissertation*, University of Málaga, Dept. Computer Architecture, December 1997).

[2] R.Asenjo, E. Gutierrez, Y. Lin, D. Padua, B. Pottengerg and E. Zapata, On the Automatic Parallelization of Sparse and Irregular Fortran Codes, *Technical Report 1512*, (Univ. for Illinois at Urbana-Champaign, CSRD, December 1996).

[3] R. Asenjo, O. Plata, J. Touriño, R. Doallo and E.L. Zapata, HPF-2 Support for Dynamic Sparse Computations, *11th. Int'l. Workshop on Languages and Compilers for Parallel Computing*, (Chapel Hill, NC, USA, August 1998).

[4] R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata, Sparse Block and Cyclic Data Distributions for Matrix Computations, *Adv. Workshop in High Performance Computing: Technology, Methods and Applications*, (Cetraro, Italy, June 1994) 359–377.

[5] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, (Oxford University Press, UK, 1993).

[6] G. Bandera, M. Ujald'on, M.A. Trenas and E.L. Zapata, The Sparse Cyclic Distribution against its Dense Counterparts, *11th Int'l. Parallel Processing Symposium (IPPS'97)*, (Geneve, Switzerland, April 1997) 638–642.

[7] 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, USA, 1994).

[8] A. Bik, Compiler Support for Sparse Matrix Computations, *Ph.D. Dissertation*, (University of Leiden, The Netherlands, 1996).

[9] W. Blume, R. Doallo, R. Eigemann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu, Parallel Programming with Polaris, *IEEE Computer Magazine*, **29 (12)** (December 1996) 78-82.

[10] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger and P. Tu, Automatic Detection of Parallelism, A Grand Challenge for High-Performance Computing, *IEEE Parallel and Distributed Technology*, **2 (3)** (Fall 1994), 37–47.

[11] T. Brandes, S. Chaumette, M.C. Counilh, A. Darte, J.C. Mignot, F. Desprez and J. Roman, HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran. Part II: Data-Structures Visualization and HPF Support for Irregular Data Structures with Hierarchical Scheme, *J. Parallel Computing*, **23 (1–2)** (April 1997) 89–105.

[12] P. Brezany, O. Chéron, K. Sanjari and E.v. Konijnenburg, Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler, *High-Performance Computing and Networking (HPCN Europe'95)*, (Milán, Italy, May 1995) 526–531.

[13] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger and P. Tu, Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing, *IEEE Parallel and Distributed Technology*, **2 (3)** (Fall 1994) 37–47.

[14] D.M. Beazley, P.S. Lomdahl, N. Grønbech-Jensen, R. Giles and P. Tamayo, Parallel Algorithms for Short-Range Molecular Dynamics, *World Scientific's Annual Reviews in Computational Physics*, **3** (World Scientific, Ed. D. Stauffer, 1996) 119–175.

[15] S. Chatterjee, Programming Models, Compilers, and Algorithms for Irregular Data-Parallel Computations, *Int'l. J. High-Speed Computing*, **6 (2)** (June 1994) 183–222.

[16] B. Chapman, P. Mehrotra and H. Zima, HPF+: New Language and Implementation Mechanisms for the Support of Advanced Irregular Applications, *6th Workshop on Compilers for Parallel Computers*, (Aachen, Germany, December 1996) 195–206.

[17] B. Chapman, H. Zima and P. Mehrotra, Extending HPF for Advanced Data-Parallel Applications, *IEEE Parallel and Distributed Technology*, **2 (3)** (Fall 1994) 59–70.

[18] T. Davis, University of Florida Sparse Matrix Collection, *NA Digest*, **92 (42)** (October 16, 1994), **96 (28)** (July 23, 1996), **97 (23)** (June 7, 1997).

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

[20] I.S. Duff, R.G. Grimes and J.G. Lewis, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, (Research and Technology Div., Boeing Computer Services, Seattle, WA, 1992).

[21] J.-L. Dekeyser and P. Marquet, Supporting Irregular and Dynamic Computations in Data-Parallel Languages, in: G.-R. Perrin and A. Darte, eds., *The Data Parallel Programming Model*, (Springer-Verlag, Berlin, Germany, LNCS 1132, 1996).

[22] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng and H. Wu, Fortran D Language Specification, *Tech. Rep. COMP-TR90-141*, (Rice University, Computer Sience Dept., 1990).

[23] G. Fox, R.D. Williams and P.C. Messina, *Parallel Computing Works!*, (Morgan Kaufmann Pub., CA, USA, 1994).

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

[25] C. Germain, J. Laminie, M. Pallud and D. Etiemble, An HPF Case Study of a Domain-Decomposition Based Irregular Application, *4th Int'l. Conf. on Parallel Computing Technologies (PaCT'97)*, (Moscow, Russia, September 1997) 201–209.

[26] R.v. Hanxleden, Compiler Support for Machine Independent Parallelization of Irregular Problems, *Ph.D. Dissertation, Tech. Rep. CRPC-TR92301-S*, (Rice University, CRPC, November 1992).

[27] R.v. Hanxleden, K. Kennedy, C. Koelbel, R. Das and J. Saltz, Compiler Analysis for Irregular Problems in Fortran D, *5th Workshop on Languages and Compilers for Parallel Computers*, (New Haven, CT, August 1992).

[28] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer and T. Tseng, Fortran D Programming System, *4th Workshop on Languages and Compilers for Parallel Computers*, (Santa Clara, CA, August 1991).

[29] Y.C. Hu, S.L. Johnsson and S.-H. Teng, High Performance Fortran for Highly Irregular Problems, *5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'97)*, (Las Vegas, NV, USA, May 1997) 13–24.

[30] High Performance Fortran Language Specification, Version 2.0, *High Performance Fortran Forum*, (1996).

[31] V. Kotylar and K. Pingali, Sparse Code Generation for Imperfectly Nested Loops with Dependences, *11th ACM Int'l. Conference on Supercomputing*, (Vienna, Austria, July 1997) 188–195.

[32] J. M. Ku, The Design of an Efficient and Portable Interface between a Parallelizing Compiler and its Target Machine, *Master's thesis*, (Univ. of Illinois at Urbana-Champaign, CSRD, 1995).

[33] V. Kumar, A. Grama, A. Gupta and G. Karipis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, (The Benjamin/Cummings Pub., CA, USA, 1994).

[34] D.E. Lenoski and W.D. Weber, *Scalable Shared-Memory Multiprocessing*, (Morgan Kaufmann Pub., CA, USA, 1995).

[35] Y. Lin and D. Padua, On the Automatic Parallelization of Sparse and Irregular Fortran Programs, *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, (Pittsburgh, PA, May 1998).

[36] P. Mehrotra, J.V. Rosendale and H. Zima, High Performance Fortran: History, Status and Future, *J. Parallel Computing*, **24 (3–4)** (1998) 325–354.

[37] P. Mehrotra, J. Saltz and R. Voigt, Eds., *Unstructured Scientific Computation on Scalable Multiprocessors*, (The MIT Press, Cambridge, MS, USA, 1992).

[38] R. Mirchandaney, J. Saltz, R.M. Smith, D.M-. Nicol and K. Crowley, Principles of Run-Time Support for Paralle Processors, *ACM Int'l. Conference on Supercomputing*, (St. Malo, France, July 1988) 140–152.

[39] OpenMP: A Proposed Industry Standard API for Shared Memory Programming, (OpenMP Architecture Review Board, `http:www.openmp.org`), 1997.

[40] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, *J. of Computational Physics*, **117** (1995) 1–19.

[41] Bill Pottenger and Rudolf Eigenmann, Idiom Recognition in the Polaris Parallelizing Compiler, *9th ACM Int'l Conf. on Supercomputing*, (Barcelona, Spain, July 1995) 444-448.

[42] R. Ponnusamy, Y.-S. Hwang, R. Das, J.H. Saltz, A. Choudhary and G. Fox, Supporting Irregular Distributions Using Data-Parallel Languages, *IEEE Parallel and Distributed Technology*, **3 (1)** (Spring 1995) 12–24.

[43] R. Ponnusamy, J. Saltz and A. Choudhary, Runtime Compilation Techniques for Data Partitioning and Communication Reuse, *IEEE Supercomputing'93*, (November 1993) 361–370.

[44] R. Ponnusamy, J. Saltz, A. Choudhary and G. Fox, Design and Specification of PARTI Runtime Data Mapping Primitives, *Tech. Rep.*, (Univ. of Maryland, Intitute for Advanced Computer Studies, November 1992).

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

[46] L.F. Romero and E.L. Zapata, Data Distributions for Sparse Matrix Vector Multiplication, *J. Parallel Computing*, **21 (4)** (April 1995) 583-605.

[47] M. Raghavachari and A. Rogers, Understanding Language Support for Irregular Parallelism, *Int'l. Workshop on Parallel Symbolic Languages and Systems*, (October 1995) 174–189.

[48] *IRIS Power Fortran Acelerator, User's Guide*, (Silicon Graphics, Inc., SGI, Inc. 1996).

[49] *MIPSpro Automatic Parallelization*, (Silicon Graphics, Inc., SGI, Inc. 1998).

[50] Special Issue on Irregular Problems in Supercomputing Applications, *J. Parallel and Distributing Computing*, **50 (1–2)** (January 1998).

[51] Special Issue on Languages and Compilers for Parallel Computers, *J. Parallel Computing*, **24 (3–4)** (March 1998).

[52] G.P. Trabado, Compilation Support for Parallel Irregular Scientific Problems, *Ph.D. Dissertation*, (University of Málaga, Dept. of Computer Architecture, Spain, December 1998).

[53] C.-W. Tseng, An Optmizing Fortran D Compiler for MIMD Distributed-Memory Machines, *Ph.D. Dissertation*, (Rice University, Dept. Computer Science, 1993).

[54] J. Touriño, R. Doallo, R. Asenjo, O. Plata and E.L. Zapata, Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In, *6th Workshop on Compilers for Parallel Computers*, (Aachen, Germany, December 1996) 151–168.

[55] G.P. Trabado, O. Plata and E.L. Zapata, HPF Directives and Run-Time Support for Molecular Dynamics Problems, *7th Workshop on Compilers for Parallel Computers (CPC'98)*, (Linköping, Sweden, June 1998).

[56] G.P. Trabado, E.L. Zapata, Exploiting Locality on Parallel Sparse Matrix Computations, *3rd Euromicro Workshop on Parallel and Distributed Processing*, (San Remo, Italy, January 1995) 2–9.

[57] G.P. Trabado, E.L. Zapata, Data Parallel Language Extensions for Exploiting Locality in Irregular Problems, *Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'97)*, (Minnesota, MN, August 1997) 218–234.

[58] M. Ujaldón, E.L. Zapata, B.M. Chapman ad H. Zima, Vienna Fortran / HPF Extensions for Sparse and Irregular Problems and Their Compilation, *IEEE Transactions on Parallel and Distributed Systems*, **8 (10)** (October 1997) 1068–1083.

[59] E.F. van de Velde, *Concurrent Scientific Computing*, (Springer-Verlag, Germany, 1994).

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

[61] H. Zima, P.Brezany, B. Chapman, P. Mehrotra and A. Schwald, Vienna Fotran – A Language Specification, *Tech. Rep. ACPC-TR92-4*, (Austrian Center for Parallel Computation, University of Vienna, 1992).