Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation

Manuel Ujaldon, Emilio L. Zapata, Member, IEEE, Barbara M. Chapman, Member, IEEE, and Hans P. Zima, Member, IEEE

Abstract—Vienna Fortran, High Performance Fortran (HPF), and other data parallel languages have been introduced to allow the programming of massively parallel distributed-memory machines (DMMP) at a relatively high level of abstraction, based on the SPMD paradigm. Their main features include directives to express the distribution of data and computations across the processors of a machine. In this paper, we use Vienna-Fortran as a general framework for dealing with sparse data structures. We describe new methods for the representation and distribution of such data on DMMPs, and propose simple language features that permit the user to characterize a matrix as “sparse” and specify the associated representation. Together with the data distribution for the matrix, this enables the compiler and runtime system to translate sequential sparse code into explicitly parallel message-passing code. We develop new compilation and runtime techniques, which focus on achieving storage economy and reducing communication overhead in the target program. The overall result is a powerful mechanism for dealing efficiently with sparse matrices in data parallel languages and their compilers for DMMPs.

Index Terms—Data-parallel language and compiler, sparse computation, distributed-memory machines, runtime support.

1 INTRODUCTION

During the past few years, significant efforts have been undertaken by academia, government laboratories, and industry to define high-level extensions of standard programming languages, in particular, Fortran, to facilitate data parallel programming on a wide range of parallel architectures without sacrificing performance. Important results of this work are Vienna Fortran [10], [27], Fortran D [15], and High Performance Fortran (HPF) [17], which is intended to become a de facto standard. These languages extend Fortran 77 and Fortran 90 with directives for specifying alignment and distribution of a program’s data among the processors, thus enabling the programmer to influence the locality of computation while retaining a single thread of control and a global name space. The low-level task of mapping the computation to the target processors in the framework of the Single-Program-Multiple-Data (SPMD) model, and of inserting communication for nonlocal accesses, is left to the compiler.

HPF-1, the original version of HPF, focused its attention on regular computations, and on providing a set of basic distributions (block, cyclic, and replication). Although the approved extensions of HPF-2 include facilities for expressing irregular distributions using INDIRECT, no special support for sparse data structures has been proposed.

In this paper, we consider the specific requirements for sparse computations as they arise in a variety of problem areas, such as molecular dynamics, matrix decompositions, solution of linear systems, image reconstruction, and many others.

In order to parallelize sequential sparse codes effectively, three fundamental issues must be addressed:

1) We must distribute the data structures typically used in such codes.
2) It is necessary to generalize the representation of sparse matrices on a single processor to distributed-memory machines in such a way that the savings in memory and computation are also achieved in the parallel code.
3) The compiler must be able to adapt the global computation to the local computation on each processor, resolving the additional complexity that sparse methods introduce.

This paper presents an approach to solve these three problems. First, a new data type has been introduced in the Vienna Fortran language for representing sparse matrices. Then, data distributions have been explicitly designed to map this data type onto the processors in such a way that we can exploit the locality of sparse computations and preserve a compact representation of matrices and vectors, thereby obtaining an efficient workload balance and minimizing communication. Some experiments in parallelizing sparse codes by hand [2] not only confirmed the suitability of these distributions, but also the excessive amount of time spent during the development and debugging stages of manual parallelization.

This encouraged us to build a compiler to specify these algorithms in a high-level data-parallel language. In this way, new elements were introduced to Vienna Fortran to
extend its functionality and expressivity for irregular problems. Subsequently, compiler and runtime techniques were developed to enable specific optimizations to handle typical features of sparse code, including indirect array accesses and the appearance of array elements in loop bounds.

The result is a powerful mechanism for storing and manipulating sparse matrices, which can be used in a data-parallel compiler to generate efficient SPMD programs for irregular codes of this kind. In this paper, we assume the representation and distribution of sparse data to be invariant. However, the fact the representation for sparse data is computed at runtime simplifies the additional support for handling more complex features, such as dynamic redistribution or the matrix fill-in (i.e., the runtime insertion of additional nonzero elements into the sparse matrix).

The rest of the paper is organized as follows. Section 2 introduces some basic formalism and background for handling sparse matrices. Section 3 presents several data distributions for sparse problems; Section 4 describes new directives for the specification of these distributions in the Vienna Fortran language. Sections 5 and 6, respectively, outline the runtime support and compilation technology required for the implementation of these features. Sections 7 and 8 present experimental results; we finish in Sections 9 and 10 with a discussion of related work and conclusions.

2 REPRESENTING SPARSE MATRICES ON DISTRIBUTED-MEMORY MACHINES

A matrix is called sparse if only a small number of its elements are nonzero. A range of methods have been developed which enable sparse computations to be performed with considerable savings in terms of both memory and computation [3]. Solution schemes are often optimized to take advantage of the structure within the matrix.

This has consequences for parallelization. First, we want to retain as much of these savings as possible in the parallel code. Second, in order to achieve a good load balance at runtime, it is necessary to understand how this can be achieved in terms of the data structures which occur in sparse problem formulations.

In this section, we discuss methods for representing sparse matrices on distributed-memory machines. We assume here that the reader is familiar with the basic distribution functions of Vienna Fortran and HPF [10], [27], [17], namely, BLOCK and CYCLIC(k).

Throughout this paper, we denote the set of target processors by PROCESSES and assume that the data is being distributed to a two-dimensional mesh PROCESSES of X × Y processors, numbered from 0 in each dimension. Specifically, we assume that the Vienna Fortran or HPF code will include the following declaration:

```
PROCESSES PROCESSES(0 : X - 1, 0 : Y - 1)
```

Note that this abstract processor declaration does not imply any specific topology of the actual processor interconnection network.

2.1 Basic Notation and Terminology

Each array A is associated with an index domain which we denote by I^A. A (replication-free) distribution of A is a total function δ^A : I^A → PROCESSES that maps each array element to a processor which becomes the owner of the element and, in this capacity, stores the element in its local memory. Further, for any processor p ∈ PROCESSES, we denote by X^p(p) the set of all elements of A which are local to p; this is called the local segment of A in p.

In the following, we will assume that A is a two-dimensional real array representing a sparse matrix, and declared with index domain I = [1 : n] × [1 : m]. Most of the notation introduced below will be related to A, without explicitly reflecting this dependence. We begin by defining a set of auxiliary functions.

**Definition 1.**

1) The symbolic matrix associated with A is a total predicate, τ : I → [true, false], such that for all i ∈ I, τ(i) = true iff A(i) ≠ 0.
2) α := |{i | i ∈ I ∧ τ(i)}| specifies the number of matrix elements with a nonzero value.
3) µ : I → [1 : n * m] is a bijective enumeration of A, which numbers all elements of A consecutively in some order, starting with 1.
4) Assume an enumeration µ to be selected. Then, ν : [1 : α] → I is a total function such that for all t ∈ [1 : α], ν(t) = i_t, where i_t is the tth index under µ which is associated with a nonzero element of A.

By default, we will use an enumeration in the following, which numbers the elements of A row-wise, i.e., we will assume µ(i, j) = (i - 1) * m + j for all (i, j) ∈ I.

When a sparse matrix is mapped to a distributed-memory machine, our approach will require two kinds of information to be specified by the user. These are:

1) The representation of the sparse matrix on a single processor. This is called a sparse format.
2) The distribution of the matrix across the processors of the machine.

In this context, the concept of a distribution is used as if the matrix was dense.

The combination of a sparse format with a distribution will be called a distributed sparse representation of the matrix.

2.2 Sparse Formats

Before we discuss data distribution strategies for sparse data, we must understand how such data is usually represented on a single processor. Numerous storage formats have been proposed in sparse-matrix literature; for our work, we have used the very commonly used CRS (Compressed Row Storage) format; the same approach can be extended to CCS (Compress Column Storage) by swapping rows and columns in the text.

In the following, we will for simplicity only consider sparse matrices with real elements; this can be immediately generalized to include other element types such as logical, integer, and complex.

**Definition 2.** The Compressed Row Storage (CRS) sparse format is determined by a triple of functions, (DA, CO, RO):
1) DA : [1 : α] → ℝ, total, the data function, is defined by DA(t) := A(v(t)) for all t ∈ [1 : α].
2) CO : [1 : α] → [1 : m], total, the column function, is defined by CO(t) := v(t),2 for all t ∈ [1 : α].
3) RO : [1 : n + 1] → [1 : α + 1], total, the row function, is defined as follows:
   a) Let i ∈ [1 : n] denote an arbitrary row number. Then, RO(i) := min{t | v(t),1 = i} if at least one t with
      the specified property exists; otherwise RO(i) := RO(i + 1).
   b) RO(n + 1) := α + 1

These three functions can be represented in an obvious way as vectors of α real numbers (the data vector), α column
numbers (the column vector), and n + 1 integer numbers in the range [1 : α + 1] (the row vector), respectively
(see Fig. 1b). The data vector stores the nonzero values as they are traversed in a row-wise fashion. The column
vector stores the column indices of the elements in the data vector. Finally, the row vector stores the indices in the
data vector that correspond to the first nonzero element of each row (if such an element exists).

The storage savings achieved by this approach is usually significant. Instead of storing n * m elements, we need only
2α + n + 1 storage locations.

\[
\begin{pmatrix}
0 & 53 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 21 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 72 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 17 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 44 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 19 & 0 & 0 & 0 & 0 \\
23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 64 & 0 \\
27 & 0 & 0 & 0 & 11 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

(a)

\[
\begin{pmatrix}
DA & CO & RO \\
53 & 2 & 1 \\
21 & 7 & 1 \\
19 & 1 & 1 \\
16 & 8 & 1 \\
72 & 6 & 1 \\
17 & 4 & 5 \\
93 & 5 & 6 \\
13 & 7 & 7 \\
44 & 5 & 8 \\
19 & 8 & 9 \\
23 & 2 & 11 \\
69 & 3 & 14 \\
37 & 5 & 17 \\
27 & 1 & 1 \\
11 & 4 & 1 \\
64 & 7 & 1
\end{pmatrix}
\]

(b)

Fig. 1. (a) Sample sparse matrix A₀(10, 8) with α = 16, and (b) its CRS representation.

Sparse matrix algorithms designed for the CRS format typically use a nested loop, with the outer loop iterating
over the rows of the matrix and an inner loop iterating over the nonzeros in that row (see examples in Section 4). Matrix
elements are identified using a two-dimensional index set, say (i, jj), where i denotes the ith row of the matrix and
jj denotes the jth nonzero in that row. The matrix element referred to by (i, jj) is the one at row number Rᵢ, column
number CO(RO(i) + jj) and has the nonzero value stored in DA(RO(i) + jj).

The heavy use of indirect accesses that sparse representations require introduces a major source of complexity and inefficiency
when parallelizing these codes on distributed-memory machines. A number of optimizations will be presented later on to
overcome this.

3 DISTRIBUTED SPARSE REPRESENTATIONS

Let A denote a sparse matrix as discussed above, and δ be an associated distribution. A distributed sparse representa-
tion for A results from combining δ with a sparse format. This is to be understood as follows: The distribution δ is
interpreted in the conventional sense, i.e., as if A were a dense Fortran array: δ determines a locality function, λ, which,
for each p ∈ PROCs, specifies the local segment λ(p). Each λ(p) is again a sparse matrix. The distributed sparse
representation of A is then obtained by constructing a representation of the elements in λ(p), based on the given
sparse format. That is, DA, CO, and RO are automatically converted to the sets of vectors DA', CO', and RO', p ∈ PROCs. Hence, the parallel code will save computation and storage using the very same mechanisms that were
applied in the original program.

For the sparse format, we use CRS to illustrate our ideas. For the data distributions, we introduce two different schemes
in subsequent sections, both decomposing the sparse global domain into as many sparse local domains as required.

3.1 Multiple Recursive Decomposition (MRD)

Common approaches for partitioning unstructured meshes while keeping neighborhood properties are based upon
coordinate bisection, graph bisection, and spectral bisection [8], [18]. Spectral bisection minimizes communication, but
requires huge tables to store the boundaries of each local region and an expensive algorithm to compute it. Graph
bisection is algorithmically less expensive, but also requires large data structures. Coordinate bisection significantly
tends to reduce the time to compute the partition at the expense of a slight increase in communication time.

Binary Recursive Decomposition (BRD), as proposed by Berger and Bokhari [4], belongs to the last of these catego-
ries. BRD specifies a distribution algorithm where the sparse matrix A is recursively bisected, alternating vertical
and horizontal partitioning steps until there are as many submatrices as processors. Each submatrix is mapped to a
unique processor. A more flexible variant of this algorithm produces partitions in which the shapes of the individual
rectangles are optimized with respect to a user-determined function [7].

In this section, we define Multiple Recursive Decomposition (MRD), a generalization of the BRD method, which also
improves the communication structure of the code.
DEFINITION 3 MRD DISTRIBUTION. We again assume the processor array to be declared as \( \text{PROCS}(0 : X - 1, 0 : Y - 1) \). Let \( X \ast Y = P_1 \ast P_2 \ast \ldots \ast P_k \) be the prime factor decomposition for \( X \ast Y \), ordered in such a way that the prime factors of \( X \), sorted in descending order, come first and are followed by the factors of \( Y \), sorted in the same fashion.

The MRD distribution method produces an \( X \ast Y \) partition of matrix \( A \) in \( k \) steps, recursively performing horizontal divisions of the matrix for the prime factors of \( X \), and vertical ones for the prime factors of \( Y \):

- Step 1: Matrix \( A \) is partitioned into \( P_1 \) submatrices in such a way that the nonzero elements are spread across the submatrices as evenly as possible. When a submatrix is partitioned horizontally, any rows with no nonzero entries which are not uniquely assigned to either partition are included in the lower one; in a vertical step, such columns are assigned to the right partition.

- Step \( i \) (\( 1 < i \leq k \)): Each submatrix resulting from step \( i-1 \) is partitioned into \( P_i \) submatrices using the same criteria as before.

When this process terminates, we have created \( \prod_{i=1}^{k} P_i \) disjoint sparse submatrices. We enumerate these consecutively from 0 to \( X \ast Y - 1 \) using a horizontal ordering scheme. Now the submatrix with number \( r \ast X + s \) is mapped to processor \( \text{PROCS}(r, s) \).

This distribution defines the local segment of each processor as a rectangular matrix which preserves neighborhood properties and achieves a good load balance (see Fig. 2). The fact that we perform all horizontal partitioning steps before the vertical ones reduces the number of possible neighbors that a submatrix may have, and, hence, simplifies further analysis to be performed by the compiler and runtime system. When combined with the CRS representation for the local segments, the MRD distribution produces the MRD-CRS distributed sparse representation. This can be immediately generalized to other storage formats; however, since we only use CRS here to illustrate our ideas, we refer to MRD-CRS as MRD itself.

3.2 BRS Distributed Sparse Representation

The second strategy is based on a cyclic distribution (see Fig. 4a). This does not retain locality of access; as in the regular case, it is suitable where the workload is not spread evenly across the matrix nor presents periodicity, or when the density of the matrix varies over time. Many common algorithms are of this nature, including sparse matrix decompositions (LU, Cholesky, QR, WZ) and some image reconstruction algorithms.

In this section, we assume both dimensions of \( A_0 \) to be distributed cyclically with block length 1 (see Fig. 4b). Several variants for the representation of the distribution segment in this context are described in the literature, including the MM, ESS, and BBS methods [1]. Here, we consider a CRS sparse format, which results in the BRS (Block Row Scatter) distributed sparse representation. A very similar distributed representation is that of BCS (Block Column Scatter) [25], where the sparse format is compressed by columns, just changing rows by columns and vice versa.
Fig. 4. (a) Cyclic decomposition of a sparse matrix. Each area represents one processor in a 13 × 13 processor grid; each processor owns those elements matching its position in the processor mesh. (b) Cyclic matrix partition for \( A_0 \) on \( \text{PROCS}(0 : 1, 0 : 1) \); \( \text{PROCS}(0, 0) \)'s elements underlined. (c) BRS distributed sparse representation for it.

The mapping which is established by the BRS choice requires complex auxiliary structures and translation schemes within the compiler. However, if such data are used together with cyclically-distributed dense arrays, then the structures are properly aligned, leading to savings in communication.

4 Vienna Fortran/HPF Extensions for the Support of Sparse Matrix Computations

4.1 Language Considerations

This section proposes new language features for the specification of sparse data in a data parallel language. Clearly, block and cyclic distributions as offered in HPF-1 are not adequate for this purpose; on the other hand, Indirect distributions [15], [27], which have been included in the approved extensions of HPF-2, do not allow the specification of the structure inherent in distributed sparse representations, and, thus, introduce unnecessary complexity in memory consumption and execution time. Our proposal makes this structure explicit by appropriate new language elements, which can be seen as providing a special syntax for an important special case of a user-defined distribution function as defined in Vienna Fortran or HPF+ [11], [12].

The new language features provide the following information to the compiler and the runtime system:

- The name, index domain, and element type of the sparse matrix are declared. This is done using regular Fortran declaration syntax. This array will not actually appear in the original code, since it is represented by a set of arrays, but the name introduced here is referred to when specifying the distribution.
- An annotation is specified which declares the array as being Sparse and provides information on the representation of the array. This includes the names of the auxiliary vectors in the order data, column and row, which are not declared explicitly in the program. Their sizes are determined implicitly from the matrix index domain.
- The Dynamic attribute is used in a manner analogous to its meaning in Vienna Fortran and HPF: If it is specified, then the distributed sparse representation will be determined dynamically, as a result of executing a Distribute statement. Otherwise, all components of the distributed sparse representation can be constructed at the time the declaration is processed. Often, this information will be contained in a file whose name will be indicated in this annotation.

In addition, when the input sparse matrix is not available at compile-time, it must be read from a file in some standard format and distributed at runtime. The name of this file may be provided to the compiler in an additional directive.

Concrete examples for typical sparse codes illustrating details of the syntax (as well as its HPF variant) are given in Figs. 5 and 6.

4.2 Solution of a Sparse Linear System

There is a wide range of techniques to solve linear systems. Among them, iterative methods use successive approximations to obtain more accurate solutions at each step. The Conjugate Gradient (CG) [3] is the oldest, best known, and most effective of the nonstationary iterative methods for symmetric positive definite systems. The convergence process can be speeded up by using a preconditionator before computing the CG itself.

We include in Fig. 5 the data-parallel code for the unpreconditioned CG algorithm, which involves one matrix-vector product, three vector updates, and two inner products per iteration. The input is the coefficient matrix, \( A \), and the vector of scalars \( B \); also, an initial estimation must be computed for \( \text{Xvec} \), the solution vector. With all these elements, the initial residuals, \( R \), are defined. Then, in every iteration, two inner products are performed in order to update scalars that are defined to make the sequences fulfill certain orthogonality conditions; at the end of each iteration, both solution and residual vectors are updated.
### 4.3 Lanczos Algorithm

Fig. 6 illustrates an algorithm in extended HPF for the tridiagonalization of a matrix with the Lanczos method [23]. We use a new directive, indicated by !NSD$, to specify the required declarative information. The execution of the DISTRIBUTE directive results in the computation of the distributed sparse representation. After that point, the matrix can be legally accessed in the program, where several matrix-vector and vector-vector operations are performed to compute the diagonals of the output matrix.

### 5 Runtime Analysis

Based on the language extensions introduced above, this section shows how access to sparse data can be efficiently translated from Vienna Fortran or HPF to explicitly parallel message passing code in the context of the data parallel SPMD paradigm.

In the rest of the paper, we assume that the input matrix is not available at compile-time. Under such an assumption, the matrix distribution has to be postponed until runtime, and this obviously enforces the global to local index translation to be also performed at runtime.

To parallelize codes that use indirect addressing, compilers typically use an inspector-executor strategy [21], where each loop accessing to distributed variables is tranformed by inserting an additional preprocessing loop, called an inspector. The inspector translates the global addresses accessed by the indirection into a (processor, offset) tuple describing the location of the element, and computes a communication schedule. The executor stage then uses the preprocessed information to fetch the nonlocal elements and to access distributed data using the translated addresses.

An obvious penalty of using the inspector-executor paradigm is the runtime overhead introduced by each inspector stage, which can become significant when multiple
levels of indirection are used to access distributed arrays. As we have seen, this is frequently the case for sparse-matrix algorithms using compact storage formats, such as CRS. For example, the \( Xvec(DA(RO(i) + jj)) \) reference encountered in Fig. 5 requires three preprocessing steps—one to access the distributed array \( RO \), a second to access \( DA \), and yet a third to access \( Xvec \). We pay special attention to this issue in this section and outline an efficient solution for its parallelization.

### 5.1 The SAR Approach

Though they are based on the inspector-executor paradigm, our solution for translating CRS-like sparse indices at runtime within data-parallel compilers significantly reduces both time and memory overhead compared to the standard and general-purpose CHAOS library [22].

This technique, that we have called “Sparse Array Rolling” (SAR), encapsulates into a small descriptor information on how the input matrix is distributed across the processors. This allows us to determine the (processor, offset) location of a sparse matrix element without having to plod through the distributed auxiliary array data-structures, thus saving the preprocessing time required by all the intermediate arrays.

Fig. 7 provides an overview of the SAR solution approach. The distribution of the matrix represented in CRS format is carried out by a partitioner, the routine responsible for computing the domain decomposition giving as output the distributed representation as well as its associated descriptor. This descriptor can be indexed through the translation process using the row number (\( R \)) and the nonzero index (\( X \)) to locate the processor and offset at which the matrix element is stored. When the element is found to be nonlocal, the dereference process assigns an address in local memory where the element is placed once fetched. The executor stage uses the preprocessed information inside a couple of gather/scatter routines, which fetch the marked

---

**Fig. 6.** Extended HPF code for the Lanczos Algorithm.

```hpf
//HPF$ PROCESSORS PROCES(0:3,0:3)
PARAMETER (N=1000) ! Dimension of matrices
REAL Q(N,N) ! Dense Orthonormal Matrix
//HPF$ DISTRIBUTE Q(CYCLIC,CYCLIC)
C T: Tridiagonal matrix. Alpha: Diagonal of T.
C Beta: Both strips surrounding diagonal of T.
REAL, DIMENSION (N) :: R, S, T ! Dense vectors
REAL ALPHA(N), BETA(N)
//HPF$ DISTRIBUTE (CYCLIC) ONTO PROCES(*,*) ::
* R, S, T, ALPHA, BETA

!HPF$ REAL A(N,N), SPARSE(CRS(DA,CO,RO)),
* DYNAMIC
- I/O operations to read sparse matrix A -
!HPF$ DISTRIBUTE A(CYCLIC, CYCLIC)

Q(1,1) = 1
DO INDEPENDENT I = 1, N
  R(I) = 0.0
  ALPHA(I) = 0.0
  BETA(I) = 0.0
END DO

C MAIN LOOP:
DO I = 1, N
  DO INDEPENDENT J = 1, N ! Sparse MxV
    DO K = RO(J), RO(J+1)-1
      R(J) = R(J) + DA(K)*Q(CO(K),I)
    END DO
  END DO
  DO J = 1, N ! Vector-Vector Product
    ALPHA(I) = ALPHA(I) + Q(J,I)*R(J)
  END DO
END DO

C REORTHOGONALIZATION PHASE
DO INDEPENDENT J = 1, N ! Dense MxV
  S(J) = 0
  DO K = 1, I
    S(J) = R(K)*Q(J,K)
  END DO
END DO

DO INDEPENDENT J = 1, N ! Dense MxV
  T(J) = 0
  DO K = 1, I
    T(J) = Q(J,K)*S(K)
  END DO
END DO

DO J = 1, N ! Vector-Vector Product
  BETA(I) = BETA(I) + R(J)*R(J)
  IF (BETA(I) .EQ. 0) THEN STOP
  DO INDEPENDENT J = 1, N
    Q(J+1) = R(J) / BETA(I)
  END DO
  DO INDEPENDENT J = 1, N
    R(J) = -BETA(I) * Q(J,I)
  END DO
END DO
END
```
nonlocal elements and place them in their assigned locations. Finally, the loop computation accesses the distributed data using the translated addresses.

The efficiency of the translation function and the memory overheads of the descriptor are largely dependent on how the matrix is distributed. The following sections provide these details for each of the distributions studied in this paper.

5.2 MRD Descriptor and Translation
The MRD distribution maps a rectangular portion of the dense index space \((n \times m)\) onto a virtual processor space \((X \times Y)\). Its corresponding descriptor is replicated on each of the processors and consists of two parts: A vector \(\text{part}_H\) stores the row numbers at which the horizontal partitions are made and a two dimensional array \(\text{part}_V\), of size \(n \times Y\), which keeps track of the number of nonzero elements in each vertical partition for each row.

**Example 1.** For the MRD distributed matrix in Fig. 3, the corresponding descriptor replicated among the processors is the following:

\[
\begin{align*}
\text{part}_H(1) & = 8; \text{part}_V(1,1:2) = 1,1; \\
\text{part}_V(2,1:2) & = 0,1; \\
\text{part}_V(3,1:2) & = 1,2; \text{part}_V(4,1:2) = 0,1; \\
\text{part}_V(5,1:2) & = 1,1; \text{part}_V(6,1:2) = 1,1; \\
\text{part}_V(7,1:2) & = 0,1; \text{part}_V(8,1:2) = 0,2; \\
\text{part}_V(9,1:2) & = 2,3; \text{part}_V(10,1:2) = 2,3;
\end{align*}
\]

\(\text{part}_H(1) = 8\) denotes that the horizontal partition is made at row 8. Each row has two vertical partitions. The values of \(\text{part}_V(9, 1 : 2) = 2, 3\) say that the first section of row 9 has two nonzero elements while the second section has one \((3-2=1)\).

We assume \(\text{part}_H(0) = 1, \text{part}_H(X) = N + 1, \text{part}_V(k, 0) = 0,\) and \(\text{part}_V(k, Y) = \text{RO}(k + 1) - \text{RO}(k),\) for all \(1 \leq k \leq N\).

Given any nonzero element identified by \((i, jj)\) we can perform a translation by means of its descriptor to determine the processor that owns the nonzero element. Assuming that processors are identified by their position \((myR, myC)\) in the \(X \times Y\) virtual processor mesh, the values \(myR\) and \(myC\) of the processor that owns the element satisfies the following inequalities.

\[
\begin{align*}
\text{part}_H(myR) & \leq i < \text{part}_H(myR+1), \quad (1) \\
\text{part}_V(i,myC) & \leq jj < \text{part}_V(i,myC+1). \quad (2)
\end{align*}
\]

Searching for the right \(myR\) and \(myC\) that satisfies these inequalities can require a search space of size \(X \times Y\). The search is optimized by first checking to see if the element is local by plugging in the local processor’s values for \(myR\) and \(myC\). Assuming a high degree of locality, this check frequently succeeds immediately. When it fails, a binary search mechanism is employed. The offset at which the element is located is \((\text{Xvec-part}_V(i, myC))\). Thus, the column number of the element \((i, jj)\) can be found at \(\text{CO}(\text{Xvec-part}_V(i, myC))\) on processor \((myR, myC)\), and the nonzero value can be accessed from \(\text{DA}(\text{Xvec-part}_V(i, myC))\) on the same processor, without requiring any communication or additional preprocessing steps.

5.3 BRS Descriptor and Translation
Unlike MRD, the BRS descriptor is different on each processor. Each processor \((myR, myC)\) has elements from \(n \times X\) rows mapped onto it. The BRS descriptor stores, for each local row of the matrix, an entry for every nonzero element on that row, regardless of whether that element is mapped locally or not. For those elements that are local, the entry stores the local index into \(\text{DA}\). For nonlocal elements, the entry stores the global column number of that element in the original matrix. To distinguish between the local entries and nonlocal entries, we swap the sign of local indices so that they become negative. The actual data-structure used is a CRS-like two-vector representation—a vector called \(\text{CS}\) stores the entries of all the elements that are mapped to local rows, while another vector, \(\text{RA}\), stores the indices at which each row starts in \(\text{CS}\).

**Example 2.** For the sparse matrix \(A\) and its partitioning, shown in Fig. 4, the values of \(\text{CS}\) and \(\text{RA}\) on processor \((0, 0)\) are the following:

\[
\begin{align*}
\text{RA}(1) & = 1; \text{RA}(2) = 2; \text{RA}(3) = 4; \text{RA}(4) = 5; \text{RA}(5) = 6; \text{RA}(6) = 9; \\
\text{CS}(1) & = 2; \text{CS}(2) = -1; \text{CS}(3) = 8; \text{CS}(4) = 4; \text{CS}(5) = -2; \text{CS}(6) = 2; \text{CS}(7) = -3; \text{CS}(8) = -4; \\
\text{CS}(1) & = 2; \text{CS}(2) = -1; \text{CS}(3) = 8; \text{CS}(4) = 4; \text{CS}(5) = -2; \text{CS}(6) = 2; \text{CS}(7) = -3; \text{CS}(8) = -4;
\end{align*}
\]

\(\text{CS}(1) = 2\) says that the element \((53)\) is stored on global column 2, and is nonlocal. \(\text{CS}(2) = -1\) signifies that the element \((19)\) is mapped locally and is stored at local index 1. The remaining entries have similar interpretations.

The processor owning the element \(X, X\) is identified as follows. First, the local row is identified using the simple
The entry for the element is obtained using \( M = \text{CS}(\text{RA}(r) + jj) \). If \( M \) is negative, then it implies that the element is local and can be accessed at \( \text{DA}(-M) \). If it is positive, then we have the global row \( i \) and column number \( M \) of the element. This implies that the processor owning the element is \( Q = (i \mod X, M \mod Y) \). We save the \([i, jj]\) indices in a list of indices that are marked for later retrieval from processor \( Q \). During the execution, a Gather routine will send these \([i, jj]\) indices to \( Q \), where a similar translation process is repeated; this time, however, the element will be locally found and sent to the requesting processor.

## 6 Compilation

This section describes the compiler implementation within the Vienna Fortran Compilation System (VFCS). The input to the compiler is a Vienna Fortran code extended with the sparse annotations described in Section 4. The compilation process results in a Fortran 77 code enhanced with message-passing routines, as well as the runtime support already discussed in the previous section.

The tool was structured in a set of modules such as shown in Fig. 8. We now describe each module separately.

### 6.1 Front-End

The first module is the only part of the tool which interacts with the declaration part of the program. It is responsible for:

1. The scanning and parsing of the new language elements presented in Section 4. These operations generate the abstract syntax tree for such annotations and a table summarizing all the compile-time information extracted from them. Once this table is built, the sparse directives are not needed any more and the compiler proceeds to remove them from the code.
2. The insertion of declarations for the local vectors and the auxiliary variables that the target code and runtime support utilize.

### 6.2 Parallelizer

At this stage, the compiler first scans the code, searching for the sparse references and extracting all the information available at compile-time (i.e., indirections, syntax of the indices, loops, and conditionals inside of which the reference is, etc.). All this information is then organized in a database for its later lookup through the parallelization process.

Once this is done, the loop decomposition starts. The goal here consists of distributing the workload of the source code as evenly as possible among the processors. This task turns out to be particularly complex for a compiler when handling sparse codes, mainly because of the frequent use of indirections when accessing the sparse data and the frequent use of sparse references in loop bounds.

In such cases, multiple queries to distributed sparse data are required by all processors in order to determine their own iteration space, leading to a large number of communications. To overcome this problem, we address the problem in a different way: Rather than trying to access the actual sparse values requested from the loop headers, we apply loop transformations that not only determine the local iteration space but also map such values into semantically equivalent information in the local distribution descriptor. This approach has the double advantage of reusing the compiler auxiliary structures while ensuring the locality of all the accesses performed in the loop boundaries. The result is a much faster mechanism for accessing data at no extra memory overhead.

For the MRD case, for example, arrays \( \text{partH} \) and \( \text{partV} \), determine the local region for the data in a sparse matrix based on global coordinates. In this way, the loop partitioning can be driven with very similar strategies to those of BLOCK, with the only difference of the regions having a different size (but similar workload) which is determined at runtime when the descriptor is generated from the runtime support.

For the BRS case, the solution is not that straightforward. Let us take, as an example, the CG algorithm in Fig. 5, where the dense vectors are distributed by dense CYCLIC and the sparse matrix follows a BRS scheme. Note that most of the CG loops only refer to dense structures. Its decomposition can be performed just by enforcing the stride of the loops to be the number of processors on which the data dimension traversed by the loop is distributed. This is because consecutive local data in CYCLIC are always separated by a constant distance in terms of the global coordinates. However, when references to sparse vectors are included in the loops, this fact is only true for the first matrix dimension; for the second one, the actual sparsity degree of the matrix determines the distance of consecutive data in terms of their global columns. Since this becomes unpredictable at compile-time (recall our assumption of not having the sparse matrix pattern available until runtime), a runtime checking defined as a function of the BRS distribution descriptor needs to be inserted for loops traversing the second matrix dimension to be successfully parallelized. This checking can be eventually moved to the inspector phase when the executor is computed through a number of iterations, thus decreasing the overall runtime overhead (see transformation in the final code generation, Fig. 10).
Fig. 9 provides a code excerpt that outlines the loop decomposition performed within the VFCS for the two sparse loops in Fig. 5. RA and CS are the vectors for the BRS descriptor on processor with coordinates (myR, myC). RA stores indices in the very same way that the local RO does, but considering all the elements placed in global rows \(i \times X + myR\) for any given local row \(i\). A CYCLIC-like approach is followed to extract the local iterations from the first loop and then RA traverses all the elements in the second loop and CS delimits its local iterations in a subsequent IF.

Note the different criteria followed for parallelizing both loops. In the first loop, the well-known owner’s compute rule is applied. In the second loop, though, the underlying idea is to avoid the replication of the computation by first calculating a local partial sum given by the local elements and then accumulate all the values in a single reduction phase. In this way, computations are distributed based on

CALL PARTITIONER( DA(), CO(), RO(), RA(), CS(), ...)

ND = 0
NP = 0
DO J=myR, N, X
   jl = J/X
   ibound[jl] = RA(jl)
   ubound[jl] = RA(jl)
   - We determine the local iterations here
   - rather than in the executor itself
   DO K=RA(jl), RA(jl+1)-1
      IF (CS(K) > 0) THEN
         idxD(ND) = J
         idxD(ND+1) = K
         ND = ND + 2
         idxP(NP) = J
         idxP(NP+1) = K
         NP = NP + 2
         ubound[jl] = ubound[jl] + 1
         One more iteration for the executor loops
      ENDIF
   END Do
END Do

CALL BRS.TRANSLATE(X, Y, RA, CS, idxD, ND, transD)
CALL BRS.DEREFERENCE(X, Y, RA, CS, transD, ND, f)
CALL BRS.GATHER(_)
CALL Aligned.BRS.TRANSLATE(X, Y, RA, CS, idxP, NP, transP)
CALL Aligned.BRS.DEREFERENCE(X, Y, RA, CS, transP, NP, g)
CALL Aligned.BRS.GATHER(_)

ND = 0
NP = 0
DO J=myR, N, X
   jl = J/X
   DO K=ibound(jl), ubound(jl)
      Q(jl) = Q(jl) + DA((ND) * P(g(NP))
      NP = NP + 1
      ND = ND + 1
   ENDDo
END Do

Fig. 10. Final SPMD program for the sparse loops in the Conjugate Gradient algorithm when the sparse matrix is distributed by BRS. All changes performed in the Back-End are underlined, namely, the inspector-executor phases and the set of calls to the SAR runtime support. The loop bounds are now computed in the inspector to alleviate the runtime overhead in the executor.

I/O ROUTINES

INSPECTOR LOOPS

RUNTIME ROUTINES

EXECUTOR LOOPS
the owner of every single \text{DA} and \text{P} value for a given index \text{k}, which makes them match always on the same processor. This achieves a complete locality.

6.3 Back-End
Once the workload has been assigned to each processor, the compiler enters in its last stage, whose output is the target SPMD code. To reach this goal, the code has to be transformed into inspector and executor phases for each of its loops.

Fig. 10 shows the final SPMD code for the sparse loops parallelized in Fig. 9. Overall, the next sequence of steps are carried out in this compiler module:

1) An inspector loop is inserted prior to each loop computation. The header for this loop is obtained through the syntax tree after the parallelization and statements inside the loop are generated to collect the indices to distributed arrays into auxiliary vectors. These vectors are then taken as input to the translation process.
2) Calls to the \text{translate}, \text{dereference}, and \text{scatter/gather} routines are placed between the inspector and executor loops to complete the runtime job.
3) References to distributed variables in the executor loop are syntactically changed to be indexed by the translation functions produced as output in the inspector (see functions \text{f} and \text{g} in Fig. 10).
4) Some additional I/O routines must be inserted at the beginning of the execution part to merge on each processor the local data and descriptors. In our SAR scheme, this is done by the \text{partitioner} routine.

7 Evaluation of Distribution Methods
The choice of distribution strategy for the matrix is crucial in determining performance. It controls the data locality and load balance of the executor, the preprocessing costs of the inspector, and the memory overhead of the runtime support. In this section, we discuss how BRS and MRD distributions affect each of these aspects for the particular case of the sparse loops in the CG algorithm. To account for the effects of different sparsity patterns, we chose two very different matrices coming from the Harwell-Boeing collection [14], where they are identified as PSMIGR1 and BCSSTK29. The former contains population migration data and is relatively dense, whereas the latter is a very sparse matrix used in large eigenvalue problems. Matrix characteristics are summarized in Table 1.

\begin{table}[h]
\centering
\caption{Characteristics of Benchmark Matrices}
\begin{tabular}{|c|c|c|c|c|}
\hline
Matrix & \(n\) & \(m\) & Nonzeros & Sparsity Rate \\
\hline
BCSSTK29 & 13,992 & 13,992 & 316,740 & 0.0016 \\
PSMIGR1 & 3,140 & 3,140 & 543,162 & 0.0551 \\
\hline
\end{tabular}
\end{table}

7.1 Communication Volume in Executor
Table 2 shows the communication volume in executor for 16 processors in a \(4 \times 4\) processors mesh when computing the sparse loops of the CG algorithm. This communication is necessary for accumulating the local partial products in the array \(q\). Such an operation has been implemented like a typical reduction operation for all the local matrix rows over each of the processor rows. We note two things: first, the relation between communication volume and the processor mesh configuration and, second, the balance in the communication pattern (note that comparisons of communication volumes across the two matrices should be relative to their number of rows).

\begin{table}[h]
\centering
\caption{Communication Volume for the Sparse Loops of the Conjugate Gradient Algorithm on 16 Processors}
\begin{tabular}{|c|c|c|c|c|}
\hline
Matrix & MRD & \multicolumn{3}{|c|}{BRS} \\
\cline{3-5}
 & Min & Max & Avg & Avg \\
\hline
PSMIGR1 & 6,706 & 7,584 & 6,996 & 6,996 \\
BCSSTK29 & 1,260 & 1,746 & 1,570 & 1,570 \\
\hline
\end{tabular}
\end{table}

In general, for an \(X \times Y\) processor mesh and an \(n \times m\) sparse matrix, the communication volume is roughly proportional to \((n/X) \times \log(Y)\). Thus, an \(8 \times 2\) processor mesh will have four times less total communication volume than a \(4 \times 4\) mesh. For BRS, each processor accumulates exactly the same amount of data, while, for MRD, there are minor imbalances stemming from the slightly different sizes of the horizontal partitions (see Fig. 11). Communication time in executor is showed in black in Fig. 13.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig11.png}
\caption{Shape of the MRD-distributed matrices on 16 processors.}
\end{figure}

7.2 Loop Partitioning and Workload Balance
As explained in Section 6.2, each iteration of the sparse loops in the CG algorithm is mapped to the owner of the \text{DA} element accessed in that iteration. This results in perfect workload balance for the MRD case, since each processor owns an equal number of nonzeros. BRS workload balance relies on the random positioning of the elements, and, except for pathological cases, it too results in very good load

\begin{table}[h]
\centering
\caption{Load Balance Index for the BRS Distribution}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Matrix & \(2 \times 1\) & \(2 \times 2\) & \(4 \times 2\) & \(4 \times 4\) & \(8 \times 4\) & \(8 \times 8\) \\
\hline
BCSSTK29 & 2.6% & 3.8% & 3.9% & 9.1% & 9.3% & 19.8% \\
PSMIGR1 & 2.6% & 5.0% & 5.8% & 6.9% & 16.3% & 25.5% \\
\hline
\end{tabular}
\end{table}
balance. Table 3 shows the Load Balance Index for BRS (maximum variation from average divided by its average).

### 7.3 Memory Overhead

Vectors for storing the local submatrix on each processor require similar amounts of memory in both distributions. However, the distribution descriptor used by the runtime support can require substantially different amounts of memory. Table 4 summarizes these requirements. The first row indicates the expected memory overhead and the next two rows show the actual overhead in terms of the number of integers required. The “overhead” column represents the memory overhead as a percentage of the amount of memory required to store the local submatrix.

Vectors `partV` and `CS` are responsible of most overhead of its distribution, since they keep track of the positions of the nonzero elements in the MRD and BRS, respectively. This overhead is much higher for BRS because the `CS` vector stores the column numbers even for some of the off-processor nonzeros. The length of this vector can be reduced by using processor meshes with \( X >> Y \).

#### 8 Runtime Evaluation

This section describes our performance evaluation of the sparse loops of the CG algorithm when parallelized using the VPCS compiler under the BRS and MRD specifications. Our intent was to study the effect of the distribution choice on inspector and executor performance within a data-parallel compiler.

Finally, a manual version of the application was used as a baseline to determine the overhead of a semi-automatic parallelization.

Our platform was an Intel Paragon using the NXLIB communication library. In our experiments, we do not account for the I/O time to read in the matrix and perform its distribution.

### 8.1 Inspector Cost

Fig. 12 shows the preprocessing costs for the sparse loops of the CG algorithm when performing the inspector phase corresponding to the sparse loops of the CG algorithm.

![Fig. 12. Preprocessing cost for each matrix when performing the inspector phase corresponding to the sparse loops of the CG algorithm.](image)

Fig. 13. Executor time for the sparse loops of the CG algorithm, where the communication time has been broken up and highlighted in black.

![Fig. 13. Executor time for the sparse loops of the CG algorithm, where the communication time has been broken up and highlighted in black.](image)
over a larger space. Though it is not shown in the table, our measurements indicate that the BRS inspector is actually faster than MRD for more than 64 processors.

### 8.2 Executor Time

Since both schemes distribute the nonzeros equally across processors, we found that the computational section of the executor scaled very well for both distributions until 32 processors, after which the communication overheads start to reduce efficiency. Fig. 13, which shows the executor time for the sparse loops of the two CG versions, indicates good load balance. In fact, we find some cases of superlinear speedup, attributable to cache effects.

The executor communication time is shown in dark in Fig. 13. The BRS communication overhead remains essentially invariant across all processor sizes. This suggests that the overhead of the extra communication startups is offset by the reduced communication volume, maintaining the same total overhead. For MRD, the communication is much more unbalanced and this leads to much poorer scaling of the communication costs. Indeed, this effect is particularly apparent for BCSSTK29, where the redistribution is extremely unbalanced and becomes a severe bottleneck as the processor size is increased.

### 8.3 Comparison to Manual Parallelization

The efficiency of a sparse code parallelized within the VFCS compiler depends largely on primary factors:

- the distribution scheme selected for the parallelization, either MRD or BRS,
- the sparsity rate of the input matrix,
- the cost of the inspector phase to figure out the access pattern.

On the other hand, we have seen that the parallelization of the sparse loops of the CG algorithm within the VFCS leads to a target code in which the executor does not perform any communication in the *gather/scatter* routines as a consequence of the full locality achieved by the data distribution, its local representation, and the loop partitioning strategy. Apart from the actual computation, the executor only contains the communication for accumulating the local partial products, which is implemented in a reduction routine exactly as a programmer would do. Thus, the executor time becomes an accurate estimation of the efficiency that a smart programmer can attain and the additional cost of using an automatic compilation lies entirely in the preprocessing time (inspector loops plus subsequent runtime calls in Fig. 10).

Fig. 14 tries to explain the impact of the major factors that influence the parallel efficiency while providing a comparison between the manual and the compiler-driven parallelization. Execution times for the compiler include the cost for a single inspector plus an executor per iteration, whereas, for the manual version, no inspector is required.

As far as the distribution itself is concerned, Fig. 14 shows that the BRS introduces a bigger overhead. This is a direct consequence of its more expensive inspector because of the slower global to local translation process. However, even in the BRS case, our overall results are quite efficient through a number of iterations: In practice, the convergence in the CG algorithm starts to exhibit a stationary behavior after no less than 100 iterations. By that time, the inspector cost has already been widely amortized and the total compiler overhead is always kept under 10 percent, regardless of the input matrix, the distribution chosen, and the number of processors in the parallel machine.

---

**Fig. 14.** Influence of the matrix distribution and sparsity in the efficiency of the compiler parallelization against a programmer (CG algorithm). Upper left: MRD distribution and PSMIGI1 input matrix. Upper right: MRD distribution and BCSSTK29 input matrix. Lower left: BRS distribution and PSMIGR1 input matrix. Lower right: BRS distribution and BCSSTK29 input matrix.
With respect to the matrix sparsity, we can conclude that the higher the degree of sparsity in a matrix is, the better is the result produced by a compiler if compared to the manual version. The overall comparison against a manual parallelization also reflects the good scalability of the manual gain for a small number of iterations.

Summarizing, we can say that the cost to be paid for an automatic parallelization is worthwhile as long as the algorithm can amortize the inspector costs through a minimum number of iterations.

The remaining cost of the CG algorithm lies in the multiple loops dealing with dense arrays distributed by CYCLIC. However, the computational weight of this part never goes over 10 percent of the total execution time. Even though the compiler efficiency is expected to be improved for such cases, its influence is minimal and does not lead to a significant variation in the full algorithm.

Additional experiments to demonstrate the efficiency of our schemes have been developed by Trenas [23], who implemented a manual version of the Lanczos algorithm (see Fig. 6) using PVM routines and the BRS scheme.

9 RELATED WORK

Programs designed to carry out a range of sparse algorithms in matrix algebra are outlined in [3]. All of these codes require the optimizations described in this paper if efficient target code is to be generated for a parallel system.

There are a variety of languages and compilers targeted at distributed memory multiprocessors ([27], [9], [15], [17]). Some of them do not attempt to deal with loops that arise in sparse or irregular computation. One approach, originating from Fortran D and Vienna Fortran, is based on indirect data distributions and cannot express the structure of sparse data, resulting in memory and runtime overhead. The scheme proposed in this paper provides special syntax for a special class of user-defined data distributions, as proposed in Vienna Fortran and HPF+ [12].

On the other hand, in the area of automatic parallelization, the most outstanding tools we know (Parafrase [19], Polaris [6]) are not intended to be a framework for the parallelization of sparse algorithms, such as those addressed in our present work.

The methods proposed by Saltz et al. for handling irregular problems consist of endowing the compiler with a runtime library [22] to facilitate the search and capture of data located in the distributed memory. The major drawback of this approach is the large number of messages that are generated as a consequence of accessing a distributed data addressing table and its associated overhead of memory [16].

In order to enable the compiler to apply more optimizations and simplify the task of the programmer, Bik and Wijshoff [5] have implemented a restructuring compiler which automatically converts programs operating on dense matrices into sparse code. This method postpones the selection of a data structure until the compilation phase. Though more friendly to the end user, this approach has the risk of inefficiencies that can result from not allowing the programmer to choose the most appropriate sparse structures.

Our way of dealing with this problem is very different: We define heuristics that perform an efficient mapping of the data and a language extension to describe the mapping in data parallel languages [17], [27]. We have produced and benchmarked a prototype compiler, integrated into the VFCS, that is able to generate efficient code for irregular kernels. Compiler transformations insert procedures to perform the runtime optimizations. The implementation is qualitatively different from the efforts cited above in a number of important respects, in particular, with respect to its use of a new data type (sparse format) and data distributions (our distributed sparse representations) for irregular computation. The basic ideas in these distributions take into account the way in which sparse data are accessed and map the data in a pseudoregular way so that the compiler may perform a number of optimizations for sparse codes. More specifically, the pseudoregularity of our distributions allows us to describe the domain decomposition using a small descriptor which can, in addition, be accessed locally. This saves most of the memory overhead of distributed tables as well as the communication cost needed for its lookup.

In general, application codes in irregular problems normally have code segments and loops with more complex access functions. The most advanced analysis technique, known as slicing analysis [13], deals with multiple levels of indirection by transforming code that contains such references to code that contains only a single level of indirection. However, the multiple communication phases still remain. The SAR technique implemented inside the sparse compiler is novel because it is able to handle multiple levels of indirection at the cost of a single translation. The key for attaining this goal consists of taking advantage of the compile-time information about the semantic relations between the elements involved in the indirect accesses.

10 CONCLUSIONS

In this paper, sparse data distributions and specific language extensions have been proposed for data-parallel languages, such as Vienna Fortran or HPF, to improve their handling of sparse irregular computation. These features enable the translation of codes, which use typical sparse coding techniques, without any necessity for rewriting. We show in some detail how such code may be translated so that the resulting code retains significant features of sequential sparse applications. In particular, the savings in memory and computation which are typical for these techniques are retained and can lead to high efficiency at runtime. The data distributions have been designed to retain data locality when appropriate, support a good load balance, and avoid memory wastage.

The compile time and runtime support translates these into structures which permit a sparse representation of data on the processors of a parallel system.

The language extensions required are minimal, yet sufficient to provide the compiler with the additional information needed for translation and optimization. A number of typical code kernels have been shown in this paper and in [25] to demonstrate the limited amount of effort required to port a sequential code of this kind into an extended HPF or Vienna Fortran.
Our results demonstrate that the data distributions and language features proposed here supply enough information to store and access the data in distributed memory, as well as to perform the key compiler optimizations, which bring great savings in terms of memory and communication overhead.

Low-level support for sparse problems has been described, proposing the implementation of an optimizing compiler that performs these translations. This compiler improves the functionality of data-parallel languages in irregular computations, overcoming a major weakness in this field.

Runtime techniques are used in the context of inspector-executor paradigm. However, our set of low-level primitives differs from those used in several existing implementations in order to take advantage of the additional semantic information available in our approach. In particular, our runtime analysis is able to translate multiple indirect array accesses in a single phase and does not make use of expensive translation tables.

The final result is an optimizing compiler able to generate efficient parallel code for these computations, very close to what can be expected from a manual parallelization, and much faster in comparison to existing tools in this area.

ACKNOWLEDGMENTS

The work described in this paper was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC92-0942, by the Austrian Research Foundation (FWF Grant P8989-PHY), and by the Austrian Ministry for Science and Research (BMWF Grant GZ 308.9281-IV/3/93).

REFERENCES

Emilio L. Zapata received his degree in physics from the University of Granada in 1978 and his PhD in physics from the University of Santiago de Compostela in 1983. From 1978 to 1982, he was an assistant professor at the University of Granada. In 1982, he joined the University of Santiago de Compostela, where he became full professor in 1990. Since 1991, he has been a full professor at the University of Malaga, where he is currently the head of the Computer Architecture Department.

Dr. Zapata has published more than 70 journal and 80 conference papers. Over the past five years, his research group has received several national and international projects, all of them in parallel computation and architectures. His main research topics include mapping of algorithms on multiprocessors, parallel compilation for irregularly structured computations, computer arithmetic and application-specific array processors, and parallel numerical algorithms.

He is a member of the editorial board of the Journal of Parallel Computing and the Journal of Systems Architecture and has been a program committee member of several international conferences.

Barbara M. Chapman received the BSc (Hons) degree in mathematics from the University of Canterbury in 1975 and pursued further studies in mathematics and computer science at both the Victoria University of Wellington, New Zealand, and the University of Bonn, West Germany. She joined the staff of the Department of Computer Science at Bonn University in 1985, teaching compiler construction, parallel systems, and programming parallel computers. In 1990, she took on a position in the Computer Science Department of the University of Vienna, where she worked on the design and implementation of the Vienna-Fortran data-parallel language and on tool support for program analysis and improvement. Since 1995, she has been building the Center for Parallel Computing in Vienna.

Ms. Chapman received the University Junior Scholarship and the Postgraduate Scholarship in New Zealand, the Cook Memorial Prize from the University of Canterbury, and the D.A.A.D. Scholarship from the Government of the Federal Republic of Germany. She is coauthor of the book Supercompilers for Parallel and Vector Computers and of many papers in the field of programming support for parallel machines. Her current research interests are in programming languages and environments for high-performance computing, data distribution, and support for the migration of applications to high-performance computing systems. She is a member of the ACM and IEEE.

Hans P. Zima studied mathematics, physics, and astronomy at the University of Vienna, where he received his PhD degree in 1964. After working for more than eight years for German and U.S. computer manufacturers and software companies, he accepted a research position at the University of Karlsruhe. He was appointed a professor of computer science at the University of Bonn, Germany, in 1975. During his tenure in Bonn, he led the development of SUPERB, the first Fortran parallelization system for distributed-memory machines. Furthermore, he performed research in the field of constraint languages at the IBM San Jose Research Laboratory in 1983-84 and initiated a cooperation with Rice University, where he spent a year in 1988-89. He is currently a professor of applied computer science and head of the Institute for Software Technology and Parallel Systems at the University of Vienna. He guided the development of Vienna Fortran, one of the major inputs for the High Performance Fortran effort and of the Vienna Fortran Compilation System (VFCS).

He is the author of several books and more than 100 publications in the areas of programming languages and compilers for parallel systems. His book, Supercompilers for Parallel and Vector Computers (coauthored with Barbara Chapman), is one of the first coherent treatments of an important emerging discipline of applied computer science. His main research interests are in the field of advanced languages, programming environments, and software tools for massively parallel machines, in particular, automatic parallelization, performance analysis, and knowledge-based transformation systems. He led research efforts in the context of the ESPRIT projects GENESIS, PREPARE, and PPPE, and is currently involved in the ESPRIT IV LTR project HPF+.

He is a member of the ACM, IEEE, the Gesellschaft für Informatik, and the Austrian Computer Society.