Journal of Universal Computer Science, vol. 7, no. 8 (2001), 736-753
submitted: 18/5/01, accepted: 21/8/01, appeared: 28/8/01 © Springer Pub. Co.

Data Distribution Specification for High Performance Computing

Hans P. Zima
(Institute for Software Science, University of Vienna,
Liechtensteinstr. 22, A-1090 Vienna, Austria
zima@par.univie.ac.at)

Abstract: High performance computing (HPC) architectures are specialized machines which can reach their peak performance only if they are programmed in a way which exploits the idiosyncrasies of the architecture. An important feature of most such architectures is a physically distributed memory, resulting in the requirement to take data locality into account independent of the memory model offered to the user. In this paper we discuss various ways for managing data distribution in a program, comparing in particular the low-level message-passing approach to that in High Performance Fortran (HPF) and other high performance languages. The main part of the paper outlines a method for the specification of data distribution semantics for distributed-memory architectures and clusters of SMPs. The paper concludes with a discussion of open issues and references to future work.

Key Words: parallel architectures, distributed memory, high performance computing, data distribution.
Category: C.4

1 Introduction

Programming languages define the level of abstraction at which a user interacts with a machine. On the one hand, a high level of abstraction provides expressivity and easy readability as well as verifiability of programs, but may result in a performance penalty caused by the gap between language and architecture. On the other hand, low-level languages oriented towards the properties of a given architecture may allow the full exploitation of a machine, but at the cost of increased program complexity and reduced portability.

For von-Neumann machines, which were originally programmed in machine or assembly languages, the 1950s and 1960s brought about a fundamental change with the emergence of high-level languages which were generally accepted because the relatively small performance penalty associated with their use was far outweighed by the advantages of increased reliability and programmer productivity. For parallel architectures, this situation is dramatically different. The continued demand for increased computing power led to the development of highly parallel scalable multiprocessing systems with distributed-memory (*DMMPs*) since the mid 1980's. Such machines are potentially scalable to large numbers of processors, but their hierarchical memory scheme requires sophisticated data management strategies in order to maximize the locality of accesses; at the same time, the workload has to be distributed evenly across the processors. The standard programming model for these architectures has become the data parallel *Single-Program-Multiple-Data (SPMD)* paradigm in which parallelism is achieved by distributing the data across the processors of the machine, with each processor executing a parameterized copy of the same program, essentially working on its “own” data and communicating with other processors if access to nonlocal data items is required.

For many years no high-level software infrastructure did exist for these architectures, forcing users to adopt a low-level programming paradigm based upon a standard sequential programming language (typically Fortran or C), augmented with message passing constructs. In this paradigm, the user deals with all aspects of the distribution of data and work to the processors, and controls the program’s execution by explicitly inserting message passing operations. MPI [8] is the current programming standard for this approach.

Early research into high-level programming support focused on the development of compilation and runtime technology with the aim of automating some aspects of program development within the SPMD framework [18]. The idea is to enable the user to write code using global data references, as for shared memory, but require the specification of a data distribution, which is then used to guide the process of restructuring the code into an explicitly parallel SPMD message passing program for execution on the target machine. This work paved the way for the development of the first generation of *High Performance Languages*, including Fortran D and Vienna Fortran [7, 10, 19, 4], and leading to the de-facto standard *High Performance Fortran (HPF)* [11, 12]. HPF offers high-level features for the declaration of abstract processor sets and the specification of data distribution, data alignment, and explicitly parallel loop constructs.

In recent years, clusters of symmetric shared-memory machines have become increasingly important. Such architectures require a hybrid programming paradigm, combining the coarse-grain distributed-memory level with shared-memory parallelism inside each node. Even if an architecture provides a NUMA shared address space interface to the user, locality at the coarse grain level is still an issue and must be taken into account. The generalization of high level data distribution features as discussed above to such architectures is relatively straightforward.

This paper is organized as follows. In the next section we will discuss a motivating example, comparing two formulations – one based on MPI and one on HPF – for a parallel Jacobi relaxation code. The idea is to demonstrate the complexity of the message-passing approach even in the case of a very simple problem, and to illustrate the power and potential simplicity of a high-level data distribution specification. The following Section 3, which contains the main part of the paper, introduces a formal model for data distribution semantics in the context of DMMPs and applies it to a range of regular and irregular distributions used in current programming language extensions. Section 4 outlines a generalization of this method to clusters of SMPs and NUMA shared address space architectures. The paper concludes with a short overview of open problems (Section 5) and final remarks in Section 6.

2 A Motivating Example

The emergence of distributed-memory architectures brought the necessity of controlling locality into sharp focus. One way is to use an explicitly parallel approach, e.g., C or Fortran coupled with message passing. As a simple example, consider a parallel version of a Jacobi relaxation code using Fortran extended by MPI [8] library calls as shown in Fig. 1. The matrices A, B are partitioned by block across the columns (Fig.2). The core of the algorithm consists of the two-level loop updating the current approximation at a grid point $B(I, J)$ by computing a weighted average of the values at the neighboring grid points. A considerable organizational effort is necessary to insert the required communication: any two processors owning adjacent blocks of columns must exchange the columns at their boundary (marked by the dashed lines in Fig. 2), whereas special rules apply to the processors owning the first and last block of columns.

```
REAL, ALLOCATABLE A(:,:), B(:,:)
   ...
CALL  MPI_COMM_SIZE (c,s)   ! compute number of processes
CALL  MPI_COMM_RANK (c,myrank)   ! compute my process id
 ! compute local size M – number of columns in each process
ALLOCATE (A(0:N+1,0:M+1), B(N,M))
...
DO WHILE  ( .NOT. converged)
  DO  J=1,M
     DO  I=1,N
         B(I,J) = 0.25*(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
     END DO
  END DO
  A(1:N,1:M) = B

IF  (MOD(myrank,2)  .EQ.  1)  THEN
   CALL  MPI_SEND (B(1,1),N, MPI_REAL ,myrank-1,tag,c)
   CALL  MPI_RECV (A(1,0),N, MPI_REAL ,myrank-1,tag, c, status)
   IF  (myrank  .LT.  s-1)  THEN
     CALL  MPI_SEND (B(1,m),N, MPI_REAL ,myrank+1,tag,c)
     CALL  MPI_RECV (A(1,M+1),N, MPI_REAL ,myrank+1,tag, c, status)
   END IF
ELSE
   IF  (myrank  .GT.  0)  THEN
     CALL  MPI_RECV (A(1,0),N, MPI_REAL ,myrank-1,tag, c, status)
     CALL  MPI_SEND (B(1,1),N, MPI_REAL ,myrank-1,tag,c)
   END IF
   IF  (myrank  .LT.  s-1)  THEN
     CALL  MPI_RECV (A(1,M+1),N, MPI_REAL ,myrank+1,tag, c, status)
    CALL  MPI_SEND (B(1,M),N, MPI_REAL ,myrank+1,tag,c)
   END IF
END IF
...
END DO
```

Figure 1: Parallel Jacobi using MPI

The idea underlying HPF is to delegate this organizational work to the compiler, guided by a user-specified directive expressing the data distribution in an abstract notation. The core of an HPF-based algorithm specifying the kind of

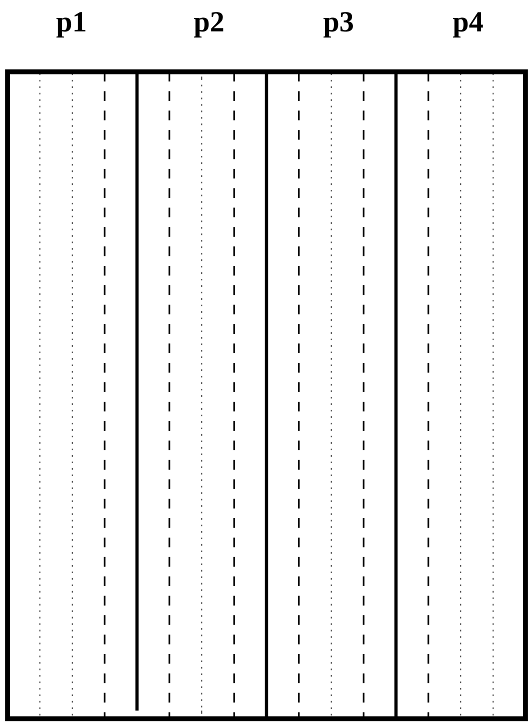

Figure 2: Block column distribution of A and B

data distribution shown in Fig. 1 above is given in the program fragment in Fig. 3.

P is declared as an array of abstract processors whose size is determined by the system inquiry function *NUMBER_OF_PROCESSORS*, which returns the number of processors being used to execute the program. This code can be run on varying numbers of processors without recompilation. A static analysis of the loop shows that no dependences exist, thus it can be parallelized by the compiler.

Note that in HPF the computation is specified using a global index space and does not contain any explicit data motion constructs. It is the compiler's responsibility to analyze the code and translate it into an explicitly parallel code (such as the MPI code given in Fig. 1) with the appropriate communication statements inserted to satisfy the data requirements.

This example – which discusses a regular code fully analyzable by the compiler – illustrates a situation where the high-level notation provided by HPF – which is more concise and much simpler than the MPI-based formulation – does not incur any performance drawbacks. In fact, there exist methods for the automatic creation of the HPF directives in cases such as this one [14, 13]. In general the problem of generating efficient code for HPF programs is more difficult however (Section 3.4).

```
!HPF$ PROCESSORS  P(NUMBER_OF_PROCESSORS())
REAL, ALLOCATABLE A(:,:), B(:,:)
!HPF$ DISTRIBUTE (*, BLOCK) ONTO P :: A, B
ALLOCATE(A(0:N+1,0:N+1), B(0:N+1,0:N+1))
  ...
DO WHILE (.NOT. converged)
  DO J=1,N
    DO I=1,N
       B(I,J) = 0.25*(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
    END DO
  END DO
 A(1:N,1:N) = B(1:N,1:N)
...
END DO
```

Figure 3: Parallel Jacobi with HPF

3 A Data Distribution Model for Distributed-Memory Systems

3.1 Distributed Memory Systems and the SPMD Paradigm

A **Distributed-Memory Multiprocessing System (DMMP)** is a homogeneous multiprocessing system in which each processor has a separate address space, its local memory. Each processor can *directly* access its local memory, whereas access to the address space of another processor must be managed via message passing communication. More precisely, we characterize a DMMP (excluding I/O) by a triple $(\mathbf{P}, \mathbf{M}, L)$, where

1. $\mathbf{P} = \{p_1, \ldots, p_r\}$: set of **processors**
2. $\mathbf{M}$: **system memory**
   $\mathbf{M}$ is partitioned into r equal-sized subsets $\mathbf{M}_1, \ldots, \mathbf{M}_r$, where $\mathbf{M}_i$ is the address space associated with processor $p_i, 1 \leq i \leq r$.
3. $L : \mathbf{P} \rightarrow \mathcal{P}(\mathbf{M})$: **memory mapping**
   L describes the mapping from processors to their address spaces: $L(p_i) = \mathbf{M}_i$ for each i.

We will classify a memory reference in a processor, p, as **local** iff it targets $L(p)$, else as **nonlocal**. Our model assumes that the cost of nonlocal acccesses in terms of latency and bandwidth is significantly higher than that for local accesses, thus emphasizing the importance of spatial locality. In the data parallel **Single-Program-Multiple-Data (SPMD)** programming paradigm, the data domain of the sequential program is partitioned in such a way that each processor "owns" a segment of data that is allocated in its local memory. The processors execute a parameterized version of the same program, with each processor running a single thread. [1] This organization results in a balance of the

[1] This binding of processors to threads remains invariant during execution of the program allowing the informal identification of processors and threads in this section.

workload across all processors on the one hand; on the other hand, processors can operate in parallel on their local data as long as no dependences exist that require communication.

In contrast to the strictly synchronous, lockstep *Single-Instruction-Multiple-Data (SIMD)* model, processors participating in an SPMD scheme operate asynchronously until the need for communication arises, resulting in a characterization of this method of execution as "loosely synchronous".

In a real DMMP system, each processor can communicate with every other processor, but the "distances" between processors may differ depending on the network topology and the relative locations of the processors in the network. This may affect communication latency and bandwidth. Our model ignores these differences, taking into account only the dichotomy local/nonlocal: in this sense it has become common to call the processors in $\mathbf{P}$ "abstract". The mapping of abstract processors to the physical processors of the real machine – which we assume to be one-to-one – is transparent.

3.2 Data Distribution

3.2.1 Basic Concepts

We assume here a sequential Fortran program to be executed on a DMMP using the SPMD paradigm. The logical address space of the program is represented by its data declarations; only arrays are considered as objects for distribution (see Section 5).

Let the DMMP be given as discussed above: $(\mathbf{P}, \mathbf{M}, L)$. Following the conventions in most HPC programming languages we represent a set of processors, $\mathbf{P}$, as an array, R. For example, $R(16, 16)$ defines the set of abstract processors as $\mathbf{P} = \{R(i,j) \mid 1 \leq i,j \leq 16\}$. Note that according to the remarks at the end of the previous section this does *not* imply any information regarding the topology of the underlying physical processors, such as a mesh.

Each array A declared in a program is associated with an **index domain**, denoted by $\mathbf{I}^A$. Similarly, for a processor array, R, $\mathbf{I}^R$ denotes the associated index domain. In the example above, $\mathbf{I}^R = [1:16] \times [1:16]$.

Definition 1 *Let A denote an array and $\mathbf{P}$ a non-empty set of processors. A* **distribution** *of A, δ^A, is a total function*

$$\delta^A : \mathbf{I}^A \to \mathcal{P}(\mathbf{P}) - \{\phi\}$$

δ^A is called **replication-free** iff for all $\mathbf{i} \in \mathbf{I}^A, \mid \delta^A(\mathbf{i}) \mid = 1$. $\square$

Definition 2 Distribution Segments
Let A, $\mathbf{I}^A$, and δ^A be defined as above. The **distribution segment***, $\lambda^A(p)$, of A and processor p with respect to δ^A is given as follows:*

$$\lambda^A(p) := \{\mathbf{i} \in \mathbf{I}^A \mid p \in \delta^A(\mathbf{i})\}.$$

An array element $A(\mathbf{i})$ is said to be **owned** *by processor p iff $\mathbf{i} \in \lambda^A(p)$.* $\square$

If A is an array and δ^A a distribution for A, then each index $\mathbf{i} \in \mathbf{I}^A$ is mapped to all processors in the set $\delta^A(\mathbf{i})$. The semantics underlying this mapping is that for each $p \in \delta^A(\mathbf{i})$, a copy of the element $A(\mathbf{i})$ is to be allocated in $L(p)$. If $\delta^A(\mathbf{i})$ contains more than one processor, we speak of *replication*. The compilation/runtime system must keep the values of replicated copies consistent. Although replication is important under certain circumstances – for example, scalar values are usually replicated to all processors in order to reduce communication –, we will in the rest of this paper deal with replication-free distributions if nothing else is said explicitly.

The function λ^A inverts δ^A by specifying for each processor, p, the set of all indices in $\mathbf{I}^A$ which are mapped to p using δ^A. Each element in $\lambda^A(p)$ is allocated a unique address in $L(p)$.

If δ^A is replication-free, and $\overline{\mathbf{P}} = \delta(\mathbf{I}^A)$, then $\lambda^A : \overline{\mathbf{P}} \rightarrow \mathcal{P}(\mathbf{I}^A)$ defines a *partition* of $\mathbf{I}^A$ (in the mathematical sense), with all elements mapped to the same processor belonging to one class.

Ownership is the key to translate references to distributed data: if processor p owns a data item $A(\mathbf{i})$, then a reference to it in p can be translated to a reference to the associated location in its local address space $L(p)$. Otherwise, a buffer for the element must be provided in $L(p)$, and message passing is required to perform the transfer between local and nonlocal memory. For example, if a nonlocal item $A(\mathbf{i})$ is to be read in p, and p' is the owner of this item, then the value of $A(\mathbf{i})$ is retrieved via message passing and transferred to the local buffer.

We noted above that for each element $A(\mathbf{i})$ with $p \in \delta^A$ an instance of $A(\mathbf{i})$ is allocated in $L(p)$. The actual local adress determined at this allocation is required in addition to the ownership information in order to complete the translation from a global array reference $A(\mathbf{i})$ to a physical address in $\mathbf{M}$. For the purpose of this paper, the details of this mapping are not relevant so we will ignore it.

The **owner computes** rule is a special version of the SPMD paradigm which in early work was used in almost all systems. According to this rule, any modification of data (via input or assignment) is executed on the processor which owns this data. Owner computes provides a very simple (although not always efficient) scheme for transforming sequential programs to SPMD programs [15].

3.2.2 Dimensional Distributions

Assume data array A and processor array R to be given, and δ^A is a distribution from A to R. δ is called a **dimensional distribution** if it specifies a mapping for exactly one dimension of $\mathbf{I}^A$ to exactly one dimension of $\mathbf{I}^R$. One or more dimensional distributions involving disjoint dimensions of $\mathbf{I}^A$ and $\mathbf{I}^R$ can be combined into a **multidimensional distribution**; alternatively, dimensions of A may also remain **undistributed**. The associated formalism has been specified in detail in [19].

Most distributions actually used in HPC languages are dimensional. For their definition it is sufficient to consider one-dimensional data and processor arrays.

3.2.3 Alignment

Alignment is a method to determine the distribution of an array, the *alignee*, from the known distribution of a *target array* and an *alignment function*.

Definition 3 *Let A, B denote arrays. An* **alignment** *of* alignee B *with* target array A *is modeled by means of an alignment function,* α_A^B*:*

$$\alpha_A^B : \mathbf{I}^B \to \mathcal{P}(\mathbf{I}^A) - \{\phi\},\ total\ \Box$$

An alignment maps each index of the alignee to one or more indices of the target array. Given an alignment function and a distribution of the target array, this leads to an obvious construction of a distribution for the alignee:

Definition 4 *Let A and B respectively denote a target array and an alignee, α_A^B an associated alignment function, and δ^A a distribution of the target array.*
Then the distribution, δ^B, of the alignee is defined as follows. For all $\mathbf{i} \in \mathbf{I}^B$:

$$\delta^B(\mathbf{i}) := \bigcup_{\mathbf{j} \in \alpha_A^B(\mathbf{i})} \delta^A(\mathbf{j}) \Box$$

Alignment is one of the most important ways which allow the construction of new distributions from already given ones. Examples include *identity alignment*, where $\alpha(\mathbf{i}) = \{\mathbf{i}\}$ for all $\mathbf{i}$, as well as replicating and collapsing specific dimensions in the target array. More general functions, as allowed in languages such as HPF, may lead to serious implementation problems [2]. A practically useful definition of alignment requires the general version of data distribution which includes replication.

3.3 Regular Dimensional Distributions

The two elementary types of regular dimensional distributions are **block** and **cyclic**. *Block* partitions the array into contiguous equal-sized regions mapped to subsequent processors, whereas *cyclic* generates a round-robin mapping of indices to processors.

Assume array A with a one-dimensional index domain $\mathbf{I}^A = [1 : n]$, and $\mathbf{I}^R = [1 : m]$ a one-dimensional index domain for the set of processors.

1. **Block Distribution**
   Assume $m \mid n$. Then the *block size*, q, is defined as $q := n/m$, and
   $$\delta^A(i) := \{\lceil \tfrac{i}{q} \rceil)\} \text{ for all } i, 1 \leq i \leq n.$$
2. **Cyclic Distribution**
   A cyclic distribution is defined by
   $$\delta^A(i) := \{MOD(i-1, m) + 1\} \text{ for all } i, 1 \leq i \leq n.$$

In general, the assumption $m \mid n$ is not required for the block distribution, leading to the need for introducing two different block sizes. A variant of block distribution used in HPF allows the explicit specification of block size, resulting in a mapping that may not involve all processors. Finally, the *block-cyclic* distribution combines block and cyclic by generating a round-robin mapping of contiguous segments of the index domain with a predefined length.

Example 1 *Let $n = 32$ and $m = 4$: A block distribution results in a block size $q = 8$ and generates a mapping*

$$\delta^A(8*(p-1)+l) := \{p\} \textit{ for all } p, 1 \leq p \leq 4, \textit{ and all } l, 1 \leq l \leq 8.$$

The distribution segment for a processor $p, 1 \leq p \leq 4$, *is given as*

$$\lambda(p) = [8*(p-1)+1 : 8*p]$$

Example 2 *Let* $A(32,16)$ *and the processor array be defined as* $R(4,4)$. *Further assume that the first dimension of A is to be distributed by block, and the second dimension cyclically. This results in a mapping (see previous example)*

$$\delta^A(4*k+l, j) := \{R(k+1, MOD(j-1,4)+1)\} \textit{ for all } k, 0 \leq k \leq 7, \textit{ all } l, 1 \leq l \leq 4, \textit{ and all } j, 1 \leq j \leq 16.$$

For example, $A(17,7)$ *is mapped to processor* $R(3,3)$. □

From the above discussion it can be easily concluded that the representation of block and cyclic distributions in the compiler and runtime system needs only a few parameters, and the mapping from indices to processors (δ) as well as the inverse mapping (λ) can be performed relatively easily and with small runtime effort. Also, scaling with respect to array (dimension) size or processor number is not a problem.

3.4 Irregular Dimensional Distributions

The block and cyclic distributions introduced in the previous section provide a simple and easily implementable mechanism which efficiently supports single structured grids. Many "real" problems, however, require irregular collections of regular grids (so-called *multiblock problems*) or unstructured grids. Regular distributions, when applied to such grids, may result in severe load imbalances and large communication overheads. The two types of irregular dimensional distributions introduced in this section provide efficient support for many classes of irregular grid structures, albeit at an increased cost in memory and time [1, 16, 15, 9].

As before we assume array A with index domain $\mathbf{I}^A = [1:n]$, and processor array R with $\mathbf{I}^R = [1:m]$.

3.4.1 General Block Distributions

In essence, general block distributions generalize the regular block distributions as introduced above by allowing different block sizes. All other properties, in particular the contiguity of the index subdomain associated with a distribution segment, and the mapping of subsequent segments to subsequent processors, remain the same.

Definition 5 *Let* $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ *denote a strictly monotonic sequence of array indices, where* b_p *denotes the first index of A to be mapped to processor* $R(p)$, $1 \leq p \leq m$, *and* $b_1 = 1$. $\mathbf{b}$ *is called the* **begin vector**.

The associated general block distribution is given as follows:

For all $i, 1 \leq i \leq n$: $\delta^A(i) := \{R(p)\}$, *where* p, $1 \leq p \leq m$, *is the smallest number such that* $b_p \geq i$.

The distribution segment associated with a processor p can then be determined as

$$\lambda(p) = [b_p : b_{p+1} - 1] \text{ for } 1 \leq p < m, \text{ and}$$
$$\lambda(m) = [b_m : n] \ \Box$$

General block distributions provide more flexibility than block distributions at moderate cost. In combination with array element reordering they can be used to deal with unstructured grids. Their representation requires $\mathbf{O}(m)$ memory size; access to an element requires time $\mathbf{O}(log_2(m))$. Thus, memory consumption of the representation can be a concern for systems with a very large number of processors.

3.4.2 Indirect Distributions

Indirect distributions are dimensional distributions that allow an arbitrary mapping from an index domain to a processor domain controlled by a **mapping function**.

Definition 6 *Let $f : [1 : n] \to [1 : m]$ denote a* **mapping function**. *f is total but not necessarily injective or surjective. The associated indirect distribution is given as follows:*

$$\text{For all } i, 1 \leq i \leq n\text{: } \delta^A(i) := \{f(i)\} \ \Box$$

The class of indirect distributions represents the most general dimensional distributions which are replication-free. In most cases where indirect distributions are used in practice, the mapping function is determined at runtime by a partitioning routine; subsequently, the array is partitioned based on the mapping function. Indirect distributions provide an elegant means for dealing with unstructured grids, their implementation is, however, expensive: the mapping function, which must be stored at runtime, requires $\mathbf{O}(n)$ memory. Since the mapping function will, because of the generally large size of n, be distributed itself in most cases (normally using a regular block distribution), access to an array element $A(i)$ may require two communication steps.

3.5 Nondimensional Distributions

Nondimensional distributions allow arbitrary mappings from the set of indices to the set of processors. Typically, such a distribution may be required when dealing with a particle-in-cell problem or a sparse matrix. We illustrate the latter case by an example which introduces a special, problem-oriented method for representing distribution segments. This representation supports the efficient parallelization of matrix-vector operations (which is not further discussed in this paper – see [17]).

	1						
						2	
3							*4*
					5		
			6				
				7			
						8	
				9			*10*
	11	*12*		*13*			
14			*15*			*16*	

Figure 4: Undistributed sparse matrix A

3.5.1 Sparse Matrix CRS Representation

Consider a sparse matrix $A(1:N, 1:M)$ with q non-zero elements as given in Fig. 4: the elements whose value is 0 are omitted there; the non-zero elements ($q = 16$) are explicitly indicated (for simplicity, we assume their values identical with their order in a row-oriented enumeration).
In the **Compressed Row Storage (CRS)** format, A is represented by three vectors, D, C, and R:

- the **data vector**, $D(1:q)$, stores the sequence of nonzero elements of A, in the order of their enumeration;
- the **column vector**, $C(1:q)$, contains in position k the column number, in A, of the k-th nonzero element in A; and
- the **row vector**, $R(1:N+1)$, contains in position i the number of the first nonzero element of A in that row (if any); else the value of R(i+1). $R(N+1)$ is set to $q+1$.

Based upon this representation, the core loop of the sequential algorithm can be formulated in Fortran as shown in Figure 5.

3.5.2 Distributed Sparse Representation

The first step in developing a parallel version of the algorithm consists of defining a *distributed sparse representation* of A. This essentially combines a data distribution with a sparse format such as CRS. More specifically, a data distribution is interpreted as if A were a dense array. The distributed sparse representation is then obtained by representing the submatrices constituting the distribution segments in the CRS format.

```
INTEGER :: C(q), R(N+1)
REAL :: D(q), B(M), S(N)
INTEGER :: I, J
 DO I = 1,M
     S(I)=0.0
     DO K = R(I), R(I+1)-1
         S(I) = S(I) + D(K)*B(C(K))
     ENDDO K
 ENDDO I
```

Figure 5: Sparse matrix vector multiply: core loop of sequential algorithm

A number of data distributions have been used for this purpose, including *Multiple Recursive Decomposition (MRD)* and cyclic distributions [17]. MRD partitions A into NN rectangular distribution segments, $A\_u, 1 \leq u \leq NN$, where NN is the number of available modules. These segments are constructed by a recursive algorithm aiming at associating approximately the same number of nonzero elements with each segment. Fig.6 illustrates an irregular distribution of our example matrix, A, into four rectangular distribution segments, and their associated representations.

Based upon a distributed sparse representation using MRD, a data-parallel algorithm for the sparse matrix vector product can be derived relatively easily [17].

4 Data Distribution for Clusters of SMPs

In recent years, **clusters of SMPs (CSMPs)** have become popular platforms for high performance computing. The essential features of such systems can be sketched as follows:

1. The basic components are symmetric shared-memory architectures (SMPs). Each SMP, s, consists of a local memory and a set of identical processors with equal and direct random access to the local memory of s.
2. A CSMP is a system of SMPs connected by a network which communicate via message passing.

The step from our DMMP model as introduced in Section 3.1 to a model for CSMPs can be performed by replacing the role of the processors in DMMPs by SMPs. More precisely, a CSMP can be characterized by a triple $(\mathbf{S}, \mathbf{M}, L)$, where

1. $\mathbf{S}$ is a set of SMPs $s_1, \ldots, s_r$. The SMPs constituting the set $\mathbf{S}$ are also called the **nodes** of the system. Each $s \in \mathbf{S}$ contains a set, $\mathbf{P}^s$, of t processors, where t is a system-constant.
2. $\mathbf{M}$ is the set of memory locations in the system. $\mathbf{M}$ is partitioned into r equal-sized subsets $\mathbf{M}_1, \ldots, \mathbf{M}_r$.
3. $L : \mathbf{S} \rightarrow \{\mathbf{M}_i \mid 1 \leq i \leq r\}$ is a bijective function that associates each SMP, s, with its **local memory** $L(s)$.

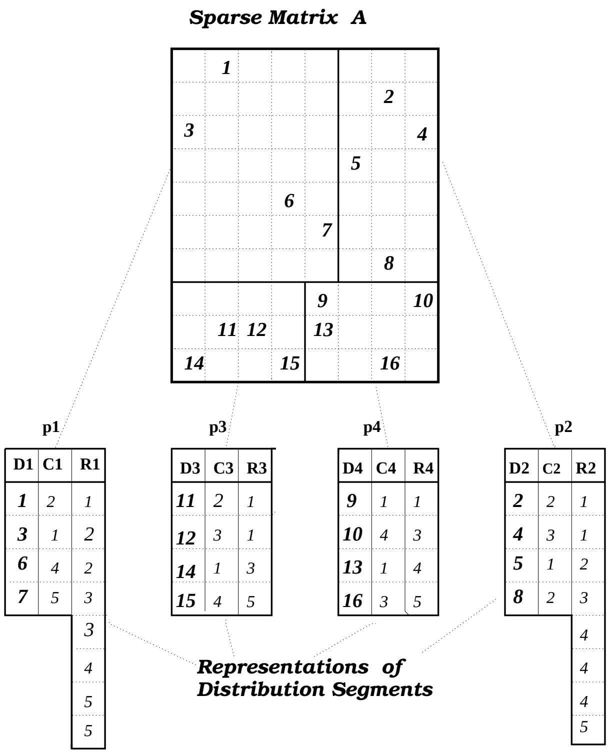

Figure 6: Distributed representation of sparse matrix A

L can be extended to define a mapping for processors as well: for any $s \in \mathbf{S}$ and all $p \in \mathbf{P}^s$, we define $L(p) := L(s)$. Local and nonlocal accesses are characterized in a similar way as for DMMPs: any access within p to an object in $L(p)$ is local, whereas accesses from p to objects in a memory $L(p')$, where $p \neq p'$, are nonlocal. As with DMMPs, nonlocal accesses carry a higher cost than local accesses.

There are a number of ways for generalizing data distribution and alignment from DMMPs to CSMPs [6, 3]. The simplest and most obvious way is to map indices to nodes rather than processors. Thus a distribution in a CSMP takes the form

$$\delta^A : \mathbf{I}^A \to \mathcal{P}(\mathbf{S}) - \{\phi\}$$

The concept of the distribution segment can be defined analogously, as well as alignment and other concepts introduced in the context of DMMPs.

The extension of the SPMD programming paradigm to CSMPs is also straightforward: a CSMP $(\mathbf{S}, \mathbf{M}, L)$ can be considered a DMMP $(\mathbf{P}, \mathbf{M}, L)$, where $\mathbf{P} = \{p \mid p \in \mathbf{P}^s$ for some $s \in \mathbf{S})$ and all processors belonging to the same SMP are mapped to the associated memory unit. The DMMP programming model now has only to be changed such that an access from p to an object in $L(p')$, where $p \neq p'$, is treated as local iff p and p' belong to the same SMP.

Virtual shared memory architectures include a range of architectures from conventional systems such as the SGI Origin'2000 all the way to massively parallel Processor-in-Memory (PIM) architectures such as the Gilgamesh machine [21]. In most cases such systems can be described as NUMA clusters of SMPs (modules) with a global virtual address space. The concepts developed above can be applied to such systems as well. One significant difference is the necessity to rearrange the *logical* address space of an array when applying distribution to it.

5 Related Issues

This section raises a diverse set of issues related to the topics dealt with in the main part of the paper. We discuss an object-based view of distributions, extend the notion of distribution to general data structures, and outline the relationship of (data) distributions to the distribution of work.

5.1 Distributions as Objects

Our discussion up to now considered distributions only in the context of arrays. However, the two concepts involved – data structure (array) and distribution – can be separated. More specifically, data distributions can be defined in an object-based framework. We outline a suitable approach below whose essential components were proposed in the Vienna Fortran language [19].

1. Distributions can be introduced as functions $\delta : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{J})$, where $\mathbf{I}$ and $\mathbf{J}$ are index domains which, in a particular instantiation, must be respectively associated with an array index domain and a processor index domain.
   Thus, a given distribution function can be used for different array and processor configurations, and a given distribution object can be associated with different data and/or processor arrays.
2. Assume a distribution, δ, as sketched above. An application of the method **distribute** (A, R, δ) creates a distribution object based on δ and the index domains $\mathbf{I}^A$ and $\mathbf{I}^R$.
3. The basic intrinsic methods applicable to a distribution object are the ones discussed in Section 3.2: the *ownership* function and the *processor-to-index* mapping λ.
   Furthermore, a range of additional methods can be defined for inquiries, prefix, and reduction functions.
4. Alignment, as discussed in Section 3.2.3, can be seen in this context as a specific distribution constructor function. Examples for other such constructors may include incremental redistribution operators as often required in irregular algorithms.

5.2 Dynamic Distribution Management

An implicit assumption made until now was the existence of a fixed association between a data structure (array) and a distribution. Howver, there are many situations involving regular or irregular distributions where the binding between a variable and a distribution can be modified at runtime. HPC languages usually offer two mechanisms for redistribution: (1) redistribution via explicit statements specifying a new distribution for an array, and (2) redistribution via a procedure call involving an array argument whose formal parameter assumes a new distribution.

Redistribution is an expensive action which requires global communication of all involved processors (SMPs); the required effort may be amortized by a reduction in the cost of accesses to the elements of a data structure.

5.3 Data Structures

Distributions can be applied to more general data structures than arrays. We outline the main idea below [21].

In the first step, the atomic components of a data structure can be uniquely identified. We do this by associating with the data structure a mapping, $D : \mathbf{I} \to \Omega$, where $\mathbf{I}$ is a generalized *index domain*, and Ω is some "universal" set of values [2]. The idea here is that we can always decompose a data structure into its "atomic" components, which designate elementary values such as fixed-point or floating-point numbers, logical values, or pointers, and that each of these components can be 1-1 mapped to a unique "name" in $\mathbf{I}$. For example, if D is a simple numerical variable, then we can choose for the index domain the singleton set $\mathbf{I} = \{1\}$. If $D(1:n, 1:m)$ is a two-dimensional Fortran array, then we define $\mathbf{I} = [1:n] \times [1:m]$. If D is an n-ary tree of height m, then each leaf can be uniquely identified by a string $i_1.i_2 \ldots .i_k$, where $k \leq m$ and all i_j are integers between 1 and n. In this way, we can represent data structures associated with arbitrary graphs if we include $\mathbf{I}$ as a subset of Ω in order to be able to deal with pointers.

The index domain, generalized as shown above, can now be used to define distributions in exactly the same way as discussed in earlier sections of this paper.

Example 3 Distribution of a tree structure
Assume D is a small binary tree with index domain $\mathbf{I} = \{1, 1.1, 1.2, 1.2.1, 1.2.2\}$, and $\mathbf{P} = \{p_1, p_2, p_3\}$ a set of processors for a DMMP. A possible distribution, δ^D, could be defined as $\delta_1^D(1) = \delta_1^D(1.1) = \{p_1\}$, $\delta_1^D(1.2) = \delta_1^D(1.2.1) = \{p_2\}$, and $\delta_1^D(1.2.2) = \{p_3\}$. □

5.4 Work Distributions

In an earlier section we mentioned the *owner computes* rule as one particular way to handle the distribution of work in an SPMD parallel program. This is not always the best way to organize the work since it may lead to excessive communication that can be avoided under a more flexible scheduling strategy.

Consider for example the execution of an independent loop in HPF, i.e., a loop whose iterations can all be executed in parallel since there exists no loop-carried dependence [20]:

[2] For the purpose of discussing distributions we take this limited view, not dealing explicitly with such information as the topology of the data structure.

```
!HPF$ INDEPENDENT
      DO I=1,N
         B(I)=A(I)*A(I)/K...
         ...
      ENDDO
```

The standard method in which code is generated for such a loop is based on a **work distribution** which specifies a mapping from processors to loop iterations. If enough processors are available, all iterations can be executed in parallel. Note however that communication is required for all non-local reads and all non-local writes executed in iterations of the loop. The read communication can be performed before the actual loop execution begins; similarly, the write communication can be performed after its end. Since the amount of communcation required can strongly depend on the work distribution, it is useful to include work distributions and their explicit manipulation in the model. Quite often the work distribution can be tied to a data distribution. For example, the **on-clause ON HOME** (A(I)), when attached to the first line of the above example, expresses a work distribution which requires iteration I to be executed by processor $\delta^A(I)$, where δ^A is a replication-free distribution for array A.

This concept can be also applied to other statements (in particular, array statements) and be generalized to allow more flexibility in the on clause [12, 5].

6 Conclusion

In this paper we discussed the formal specification of data distributions supporting a high-level language approach. The motivation for the explicit modeling of distributions comes from the observation that data distributions play a key role in the formulation of algorithms for modern architectures which are generally characterized by complex memory hierarchies. Whereas present compilation and runtime technology may render some levels of that hierarchy transparent it is the belief of the author that coarse-grain data distribution should be explicitly supported in a high-level language, which in turn means that a suitable semantic model is to be developed.

Although many existing HPC languages allow the specification of data distributions, no generally accpeted semantic model has been developed until now. This paper attempts a classification of some basic properties which could guide the development of a more complete approach. Many issues, in particular those related to the efficient implementation of a suitable class of distributions remain open at this time. Compilers for languages such as Vienna Fortran and HPF can provide actual guidance in improving current languages (which may mean generalization as well as simplification of some concepts) as well as applying these features to newly developed architectures such as processor-in-memory systems.

Acknowledgements

The work described in this paper was partially supported by the Special Research Program F011 "AURORA" of the Austrian Science Fund (FWF).

References

1. G. Agrawal, A. Sussman and J. Saltz. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747-754, July, 1995.
2. S.Benkner and H.P.Zima. Compiling High Performance Fortran for Distributed-Memory Architectures. In: Trystram,D.(Ed.): *Parallel Computing 25 (1999), Special Anniversary Issue*, pp.1785-1825.
3. J.Bircsak,P.Craig,R.Crowell,Z.Cvetanovic,J.Harris,C.A.Nelson, and C.D.Offner. Extending OpenMP for NUMA Machines. *Proc.SC2000*, November 2000, Dallas, Texas.
4. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming* 1(1):31-50, Fall 1992.
5. B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data-Parallel Applications. *IEEE Parallel & Distributed Technology* 2(3):59-70, Fall 1994.
6. B.Chapman,P.Mehrotra,and H.P.Zima. Enhancing OpenMP with Features for Locality Control. In: Zwieflhofer,W. and Kreitz,N. (Eds.): *Proc. Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology "Towards Teracomputing"*, pp.301-313, Reading, England (November 1998). World Scientific, 1999.
7. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90079, Rice University, March 1991.
8. W.Gropp, E.Lusk, A. Skjellum. Using MPI – Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Cambridge, 1997.
9. E. Haug, J. Dubois, J. Clinckemaillie, S. Vlachoutsis, G. Lonsdale. Transport Vehicle Crash, Safety and Manufacturing Simulation in the Perspective of High Performance Computing and Networking. *Future Generation Computer Systems*, Vol.10, pp. 173-181, 1994.
10. S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.
11. High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report, Rice University, Houston, TX, May 3, 1993. Also available as Scientific Programming 2(1-2):1-170, Spring and Summer 1993.
12. High Performance FORTRAN Forum. *High Performance FORTRAN Language Specification, Version 2.0*, January 1997.
13. K.Kennedy and U.Kremer. Automatic Data Layout for Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems* 20(4), 869-916, July 1998.
14. J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.
15. P.Mehrotra,J.Van Rosendale, and H.P. Zima. High Performance Fortran: History, Status and Future. In: Zapata,E. and Padua,D.(Eds.): *Parallel Computing, Special Issue on Languages and Compilers*, Vol.24,No.3-4,pp.325–354 (1998)
16. J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.
17. M.Ujaldon, E.L.Zapata, B.Chapman, and H.Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(11), November 1997.
18. H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

19. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. Internal Report 21, ICASE, Hampton, VA, March 1992.
20. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.
21. H.Zima and T.Sterling. Macroservers: An Object-Based Programming and Execution Model for Processor-in-Memory Arrays. *Proc.International Symposium on High Performance Computing (ISHPC2K)*, Tokyo, Japan, October 2000.