SUPERB AND VIENNA FORTRAN*

Hans P. Zima, Peter Brezany, and Barbara M. Chapman
Institute for Software Technology and Parallel Systems
University of Vienna, Brünner Strasse 72, A-1210 VIENNA, AUSTRIA
E-Mail: zima@par.univie.ac.at

Abstract

Distributed-memory systems are powerful tools for solving large-scale scientific and engineering problems. However, these machines are difficult to program since the data have to be distributed across the processors and message-passing operations must be inserted for communicating non-local data. In this paper, we discuss SUPERB and Vienna Fortran, two related developments with the objective of providing the user with a higher level programming paradigm while not sacrificing target code performance.

The parallelization system SUPERB was developed in the German supercomputer project SUPRENUM from 1985 to 1989. It is based on the Single-Program-Multiple-Data (SPMD) paradigm, allows the use of global addresses, and automatically inserts the necessary communication statements, given a user-supplied data distribution. SUPERB was the first implemented system that translated sequential Fortran 77 into explicitly parallel message-passing Fortran.

As a result of the experiences with SUPERB and related research, the language Vienna Fortran was designed within the ESPRIT project GENESIS, in a joint effort of the University of Vienna and ICASE, NASA Langley Research Center. Vienna Fortran is a machine-independent language extension to Fortran, which includes a broad range of features for the high-level support of advanced application development for distributed-memory multiprocessors. It has significantly influenced the development of High Performance Fortran, a first attempt of language standardization in this area.

Keywords: distributed-memory multiprocessor systems, numerical computation, data parallel algorithms, data distribution, program analysis, optimization.

This paper was published in: Parallel Computing, North Holland, 1994, pp. 1487–1517

---

*The work described in this paper was supported by the Federal Ministry for Research and Technology (BMFT), Federal Republic of Germany, by the Austrian Research Foundation (FWF) and by the Austrian Ministry for Science and Research (BMWF)
1 Introduction

Since the advent of early distributed-memory multiprocessing systems (DMMPS) such as Caltech’s Cosmic Cube and the German supercomputer SUPRENUM less than a decade ago, these architectures have rapidly gained user acceptance and are today offered by most major manufacturers. Current DMMPS include Intel’s hypercubes, the Paragon, the nCUBE, Thinking Machine’s CM-5, and the Meiko Computing Surface. DMMPS are relatively inexpensive to build, and are potentially scalable to large numbers of processors.

However, these machines are difficult to program: the non-uniformity of the memory which makes local accesses much faster than the transfer of non-local data via message-passing operations implies that the locality of algorithms must be exploited in order to achieve acceptable performance. The management of data, with the twin goals of both spreading the computational workload and minimizing the delays caused when a processor has to wait for non-local data, becomes of paramount importance.
When a code is parallelized by hand, the programmer must distribute the program’s work and data to the processors which will execute it. One of the common approaches to do so makes use of the regularity of most numerical computations. This is the so-called **Single Program Multiple Data** (SPMD) or **data parallel** model of computation. With this method, the data arrays in the original program are each partitioned and mapped to the processors. This is known as **distributing** the arrays. A processor is then thought of as **owning** the data assigned to it; these data elements are stored in its local memory. Now the work is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it – this is known as the **owner computes** paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally. Accesses to non-local data must be explicitly handled by the programmer, who has to insert communication constructs to send and receive data at the appropriate positions in the code. The details of message passing can become surprisingly complex: buffers must be set up, and the programmer must take care to send data as early as possible, and in economical sizes. Furthermore, the programmer must decide when it is advantageous to replicate computations across processors, rather than send data.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected. It determines not only where computation will take place, but is also the main factor in deciding what communication is necessary. The communication statements as well as the data distribution are hardcoded into the program. It will generally require a great deal of reprogramming if the user wants to try out different data distributions. This programming style can be likened to assembly programming on a sequential machine – it is tedious, time-consuming and error prone.

Thus much research activity has been concentrated on providing programming tools for DMMPs. One of the first such tools is **SUPERB**[42], an interactive restructurer which was developed in the SUPRENUM project ([42]) starting in 1985. It translates Fortran 77 programs into message passing Fortran for the SUPRENUM machine [18], the Intel IPSC, and the GENESIS machine. SUPERB performs coarse-grain parallelization for a DMMP and is also able to vectorize the resulting code for the individual nodes of the machine. The user specifies the distribution of the program’s data via an interactive language. Program flow and dependence analysis information, using both intraprocedural and interprocedural analysis techniques, is computed and made available to the user, who may select individual transformation strategies or request other services via menus. SUPERB puts a good deal of effort into optimizing the target program, extracting communication from loops whenever possible, and combining individual communication statements (by vectorization and fusion) to reduce the overall communication cost ([16]). Simple reductions are recognized and handled by the system. SUPERB handles full Fortran 77, dealing with common blocks and equivalencing. Its implementation was completed in 1989 and thus it was the first system which compiled code for DMMPs from Fortran 77 and a description of the distribution of data. SUPERB provides special support for handling work arrays, as are commonly used in Fortran codes, for example to store several grids in one array.

The experience and success gained with SUPERB and other experimental parallelization systems for DMMPs led to a new focus of research: the provision of appropriate high-level **language constructs** for the specification of data distributions. **Vienna Fortran** [8, 43], developed within the ESPRIT project GENESIS in joint work by the University of Vienna and ICASE, NASA Langley Research Center, is a machine-independent language extension to Fortran, which includes high-level features for specifying virtual processor structures, distributing data across sets of processors, dynamically modifying distributions, and formulating explicitly parallel loops.

This paper will focus on SUPERB and Vienna Fortran, which are discussed in detail in Sections 3 and 4, after an introduction to the basic notation and terminology (Section 2). The rest of the paper deals with the relationship between Vienna Fortran and HPF (Section 5), an advanced compilation technique for dealing with irregular data accesses (Section 6), and an overview of related work (Section 7), followed by the conclusion.
2 Basic Notation and Terminology

2.1 Source and Target Languages of SUPERB

We will describe SUPERB by specifying a source-to-source translation from a subset of Vienna Fortran to message passing Fortran.\footnote{The original version of SUPERB used a syntactically different interactive input language. The term message passing Fortran stands for a language that can be thought of as a representative for a class of similar message passing dialects used in typical DMMPs.}

Vienna Fortran programs are executed by a DMMP according to the SPMD programming model. Its central concepts are processors and distributions:

- Vienna Fortran allows the explicit specification of processor arrays to define the set of abstract processors used to execute a program. This set will in the following be denoted by $P$. The model abstracts from the actual topology of the physical machine. Any two processors in $P$ communicate by exchanging messages.

- Distributions map arrays to non-empty sets of processors; they can be specified by declaration annotations.

The subset of Vienna Fortran accepted by SUPERB requires a constant number of processors and constant distribution parameters. No facility for dynamic redistribution exists.

In the target language, the basic message passing operations are SEND and RECEIVE. We assume a non-blocking SEND and a blocking RECEIVE operation. Furthermore, we postulate that a sequence of messages sent from a processor $p_1$ to a processor $p_2$ arrives at $p_2$ in the same order in which it was sent by $p_1$.

2.2 Index Domains and the Data Space of a Program

An index domain $I$ of rank (dimension) $n$ is a Cartesian product specifying a rectilinear section \cite{11} of $\mathbb{Z}^n$, where $\mathbb{Z}$ denotes the set of integer numbers.

Let $A$ denote a declared array. Then $A$ is associated with an index domain $I^A$, and a set of elements, $E^A$. There is a one-to-one correspondence between $E^A$ and $I^A$; if $i \in I^A$ is associated with $e \in E^A$, then $i$ is called the index of $e$. The elements in $E^A$ represent the memory locations associated with the array element names during program execution. Similarly, if $R$ is a processor array, then $I^R$ denotes its index domain.

In the following, we consider the data space $A$ of all declared arrays that are accessible in a given scope. $A$ does not include formal parameters. The set of all elements associated with arrays in the data space is denoted by $E$. The scalar objects declared in a program are, for simplicity, considered as specially declared arrays.

We will need an additional class of objects, which are called private variables. Private variables are strictly local to a processor, used only in that processor, and can never cause communication. An example for a private variable is a processor-local buffer. Private variables of a processor $p$ are allocated in the local address space of $p$, together with the variables that are owned by the processor.

2.3 Distributions

A distribution of an array maps each array element to one or more processors which become the owners of the element and, in this capacity, store the element in their local memory. We model distributions by mappings between the associated index domains.
Definition 1 Index Mappings
Let I, J denote two index domains. An index mapping from I to J is a total function \( i : I \rightarrow \mathcal{P}(J) - \{\emptyset\} \), where \( \mathcal{P}(J) \) denotes the powerset of J.

Definition 2 Distributions
Let A denote an array, and R a processor array. An index mapping \( \delta^A_R \) from I^A to I^R is called a distribution function for A with respect to R. □

A distribution function \( \delta^A_R \) – which is a mapping between index domains – induces an associated element-based distribution that maps elements of A to one or more abstract processors. Note that replication can be modeled as a special case of distribution, since every array element can be distributed to an arbitrary (positive) number of processors.

Whenever A or R are determined by the context, they can be omitted.

Definition 3 Local variables
Let the total function \( \lambda : P \rightarrow \mathcal{P}(E) \) be defined as follows: For each processor \( p \in P \), \( \lambda(p) = \{ e \in E \mid p \in \delta(e) \} \). \( \lambda(p) \) is the set of local variables of p; these variables are said to be owned by p.

Furthermore, let for each \( A \in A \) and each \( p \in P \), \( \lambda^A(p) := \lambda(p) \cap E^A \) determine the set of elements of A owned by p.

2.4 Distribution Annotations
SUPERB allows – regular or general – block distributions as well as replication. We illustrate these features and the annotation syntax by means of an example (Figure 1).

The PROCESSORS declaration introduces R1 as a one-dimensional processor array with four elements. The annotations attached to the declarations specify a regular block distribution for A, replication for array B, and a general block distribution for C. The elision symbol “;” is provided to indicate that an array dimension is not distributed: the annotation in the declaration of D specifies a block-wise distribution of its columns. Since in this example R1 is the only processor array specified, it is always implied and can be omitted in the annotation.

- **BLOCK** distributes an array dimension to a processor dimension in evenly sized portions. Array A is partitioned into 4 blocks of length 3. For all \( i, 1 \leq i \leq 12, \delta^A_{R1}(i) = \{\lfloor \frac{i}{3} \rfloor \} \). For example, processor R1(1) owns A(1), A(2), and A(3).
- Array B is (totally) replicated: \( \delta^B(i) = R1 \) for all i. Every processor owns a copy of the whole array.
- **GENERAL_BLOCK** distributes an array dimension to a processor dimension in arbitrarily sized portions. Array C is partitioned into 4 blocks of lengths 1.5, 3, and 1.
- The first dimension of D is not distributed, while the second dimension is distributed by **BLOCK**, with a block length of 25. More precisely: For all \( i \) and \( j, 1 \leq i, j \leq 100, \delta^C_{R1}(i, j) = \{\lfloor \frac{i}{25} \rfloor \} \).

In Figure 2 below, we show a program fragment for an iteration of Jacobi relaxation, together with one possible method for distributing the data: UNEW, U and F are all distributed by **BLOCK** in both dimensions. We will later refer to this example when explaining the phases of SUPERB.

General Block Distributions
In some cases – for example, if the boundary region of a data domain needs more work than the interior – the **BLOCK** distribution may lead to an unbalanced load. The **GENERAL_BLOCK** distributions of SUPERB are designed to deal with this problem: such distributions map single array dimensions to single processor dimensions but allow – in contrast to **BLOCK** distributions – arbitrary partitions of the array dimension. If an array A has a general block distribution, then for every processor \( p \), \( \lambda^A(p) \) is an arbitrary contiguous rectilinear section of A that is called the distribution segment.
associated with \( A \) and \( p \). We require that segments associated with different processors are either disjoint or identical. The data space \( \mathcal{A} \) can be represented as the union of two disjoint sets, the set of unpartitioned, and the set of partitioned arrays. Each unpartitioned array is replicated.

3 The Parallelization Strategy of SUPERB

The translation from Vienna Fortran to message passing Fortran will be described as a sequence of phases, each of which specifies a translation between two source language levels. We use the Jacobi code shown in the previous section to illustrate some of the transitions involved.

Assume that initially a Vienna Fortran source program \( Q^0 \) is given. We transform \( Q^0 \) into a message passing Fortran program \( Q^4 \) in four conceptually distinct consecutive phases:\footnote{This method of presentation has been chosen in order to present the major steps as clearly as possible; in contrast, a real system will frequently use shortcuts.}

\[
\begin{align*}
\text{PARAMETER} & (M=2,N=16) \\
\text{PROCESSORS} & R(M,M) \\
\text{REAL} & \text{ UNEW}(1:N,1:N), U(1:N,1:N), F(1:N,1:N) \text{ DIST ( BLOCK, BLOCK )} \\
\text{DO} & J = 2, N-1 \\
\text{DO} & I = 2, N-1 \\
\text{S: UNEW}(I,J) = 0.25 \ast (F(I,J) + U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1)) \\
\text{ENDDO} \\
\text{ENDDO} \\
\end{align*}
\]
3.1 Phase 1: Front End

**Vienna Fortran Source Program** $Q^0 \rightarrow \text{Vienna Fortran Normalized Program } Q^1$

The source program $Q^0$ is processed by a **Front End**, which performs the following three tasks: it transforms the source code into an *internal representation* suitable for further compiler manipulation; it performs the *initial analysis* of the program, and it *normalizes* the code. This analysis may have to be updated or recomputed during subsequent transformation phases.

Thus Phase 1 includes syntactic and semantic analysis, control flow analysis, data flow analysis, data dependence analysis and the construction of the call graph. The call graph is annotated with information needed in Phase 2. Further, some standard transformations such as constant propagation and dead code elimination may be applied at this stage.

Normalizations have the purpose of simplifying subsequent compiler operations. They include transformations to eliminate certain program constructs (such as statement functions, ENTRY statements or alternate RETURNS), to position specific kinds of statement at a prescribed place in the code (e.g. FORMAT or DATA statements), and to transform other statements so that they always appear in a standard form. The number of different kinds of statements affecting control flow may be considerably reduced by such transformations.

Typically, the normalization transformations performed include *loop normalization*, *if conversion* and *subscript standardization*.

We do not discuss the details of the Front End analysis and transformations in this paper: they are essentially the same as those employed for automatic vectorization and shared-memory parallelization, and as such have been discussed extensively in the literature [44]. The resulting program, $Q^1$, is called the **normalized program**.

3.2 Phase 2: Splitting

**Vienna Fortran Normalized Program** $Q^1 \rightarrow (\text{Host Program } Q^H, \text{ Node Program } Q^2)$

$Q^1$ is split into a **host program**, $Q^H$, and a **node program**, $Q^2$. The host program will, after compilation, be executed on the host computer or a specially designated host node of the target system, as the **host process**. It performs global management tasks, such as requesting resources, loading code, and terminating program execution; it also handles all input/output.

Hence the I/O and termination statements of the original program, and all control statements which these depend on, will remain in the resulting host program. Functions and subroutines which cause I/O also appear in the host program in a rudimentary form: both are converted to subroutine calls without arguments.

The node program, $Q^2$, which is still a sequential Vienna Fortran program, contains the actual computation; input/output is represented by communication with the host program. All I/O statements are removed from the node program: reads require a SEND statement in the host and a RECEIVE in the node program; writes require a RECEIVE in the host and a SEND in the node program.

Program control flow is modeled in both codes: hence the node program may have to communicate with the host to provide values of loop bounds, the result of conditionals, or values required in I/O operations. Error or end-of-file specifications in I/O statements also influence control flow, and require communication of values from the host to the node program. Optimizations are necessary to ensure that only those values are transmitted which are actually required by the recipient to perform its tasks in the correct order.

The host and node programs resulting from this phase are loosely coupled; all synchronization necessary between them at run time will be achieved through message passing.

3.3 Phase 3: Initial Parallelization

**Node Program** $Q^2 \rightarrow \text{Defining Parallel Program } Q^3$
This transformation is performed by processing the data distribution specified in the original Vienna Fortran program in two steps, which are referred to as masking and communication insertion:

- **Masking** enforces the owner computes paradigm by associating a boolean guard, called the mask, with each statement, in accordance with the ownership implied by the distributions: each statement $S$ is replaced by the masked statement

  $$\text{IF } \text{mask}(S) \text{ THEN } S \text{ ENDIF}$$

  where

  - $\text{mask}(S)$ is $\text{OWNED}(A(x))$ if $S$ is an assignment statement of the form $A(x) = \ldots$, where $A$ is partitioned and $x$ is an associated list of subscript expressions. \(^\text{3}\)
  - $\text{mask}(S) \equiv \text{true}$ in all other cases. Clearly, then the masked statement associated with $S$ can be immediately replaced by $S$.

  Note that $\text{mask}(S) \equiv \text{true}$ for all statements which are not assignments (in particular, control statements) and for all assignments to replicated variables.

- For all non-local data accesses, **communication insertion** generates communication statements which copy non-local data items to private variables of the processor. For this, the exchange primitive of level 0, EXCH0, as defined in Figure 3, is used: For every statement $S$ with $m = \text{mask}(S)$ and every right-hand side reference $\text{ref}$ to a partitioned array in $S$, CALL EXCH0$(m,\text{ref},\text{temp})$ is inserted before $S$. In $S$, $\text{ref}$ is replaced by the private variable $\text{temp}$. Each execution of a $\text{SEND} \ \text{ref} \ \text{TO} \ \text{p'}$ in a processor $\text{p}$, which is caused by a call to EXCH0$(m,\text{ref},\text{temp})$, corresponds to exactly one execution of $\text{RECEIVE} \ \text{temp} \ \text{FROM} \ \text{p} \ \text{in} \ \text{p'}$, caused by a call to the same occurrence of EXCH0$(m,\text{ref},\text{temp})$, in the same state. The order of different EXCH0 statements associated with a statement $S$ is irrelevant; however, their private variables must be distinct.

/* EXCH0$(m,\text{ref},\text{temp})$

The algorithm below specifies the effect of executing a call to EXCH0$(m,\text{ref},\text{temp})$. Here, MVAL$(\text{mask}(S),\text{p})$ yields true iff the value of $\text{mask}(S)$ in the current state is true for processor $\text{p}$. In the case where a data item $\text{ref}$ is owned by more than one processor, MASTER$(\text{ref})$ determines the processor responsible for organizing the communication with respect to $\text{ref}$. */

```plaintext
IF OWNED(ref)
  THEN temp := ref;
    IF (MY_PROC = MASTER(ref))
      THEN FOR EVERY $p$ SUCH THAT MVAL$(m,p) \land (\text{ref} \notin \Lambda(p))$ SEND temp TO p ENDFOR
    ENDIF
  ELSE
    IF $\lnot$ OWNED(ref) $\land$ MVAL$(m,\text{MY_PROC})$
      THEN RECEIVE temp FROM MASTER(ref)
    ENDIF
  ENDIF
ENDIF
```

Figure 3: The EXCH0 primitive.

Note finally that, according to the SPMD execution model, the compiler does not generate separate node programs for each processor. Instead, each will execute the same program, receiving its parameters and

\(^3\) OWNED$(U)$, executed in process $p$, yields true iff $U$ is owned by $p$. 
initial data from the host program.

We now show the effect of initial parallelization on the program of Figure 2 (I/O and splitting have been ignored). The resulting program is given in Figure 4.

PARAMETER (M=2,N=16)
PROCESSORS R(M,M)
REAL UNEW(1:N,1:N), U(1:N,1:N), F(1:N,1:N) DIST (BLOCK, BLOCK)
PRIVATE REAL TEMP1, TEMP2, TEMP3, TEMP4
...
DO J = 2, N-1
  DO I = 2, N-1
    CALL EXCH0(OWNED(UNEW(I,J)),U(I-1,J),TEMP1)
    CALL EXCH0(OWNED(UNEW(I,J)),U(I+1,J),TEMP2)
    CALL EXCH0(OWNED(UNEW(I,J)),U(I,J-1),TEMP3)
    CALL EXCH0(OWNED(UNEW(I,J)),U(I,J+1),TEMP4)
    IF OWNED(UNEW(I,J)) THEN
      S: UNEW(I,J) = 0.25 * (F(I,J) + TEMP1 + TEMP2 + TEMP3 + TEMP4)
    ENDIF
  ENDDO
ENDDO
...

Figure 4: Jacobi relaxation code after initial parallelization.

3.4 Phase 4: Optimization and Target Code generation

Defining Parallel Program $Q^3 \mapsto$ Optimized Target Program $Q^4$

The defining parallel program – as produced by Phase 3 – specifies exactly the work distribution and communication required by the input program. However, it would be very inefficient to actually compile and execute such a program on a parallel computer since communication involves only single data items and in general each processor has to evaluate the mask of a statement for all instances of that statement.

In Phase 4, the defining parallel program is transformed into an optimized parallel message passing Fortran target program, $Q^4$. Communication and masking are improved: Communication statements are moved out of loops and combined to perform aggregate communication where possible; the strip mining of loops [44] across the processors can be achieved in many cases by propagating the information in masks to the loop bounds. A prerequisite for many optimizations is precise flow and data dependence information, as gathered in Phase 1, and overlap analysis, which detects certain simple regular communication patterns and re-organizes communication based upon this information, as explained below. Overlap analysis also helps determine the minimum amount of storage which must be reserved for each partitioned data array in the memory of a node processor.

3.4.1 Overlap Analysis

Overlap analysis is performed in the compiler to determine which non-local elements of a partitioned array are used in a processor. For many regular computations, the precise pattern of non-local accesses can be computed; this information can then be used both to determine the storage requirements for the array and to optimize communication.
For each partitioned array $A$ and processor $p \in P$, the distribution segment $\lambda^A(p)$ is allocated in the local memory associated with $p$. In the defining parallel program, communication is inserted each time a potentially non-local element of $A$ is referenced in $p$ – and private variables are created to hold copies of the original non-local values. Overlap analysis is used to allocate memory space for these non-local values in locations adjacent to the local distribution segment. More precisely, the overlap area, $OA(A,p)$, is the smallest rectilinear contiguous area around the distribution segment of a process $p$, containing all non-local variables accessed (see Figure 5). The union of the distribution segment and the overlap area is called the extension segment associated with $A$ and $p$. This description can be used to significantly improve the organization of communication; it facilitates memory allocation and the local addressing of arrays. The relevant analysis is described in [14, 45].

The overlap area for an array $A$ is specified by its overlap description, $OD(A)$, which is determined by the maximum offsets for every dimension of the distribution segments, over all processors. If $n$ is the rank of $A$, this takes the form

$$OD(A)=[dl_1 : du_1, \ldots, dl_n : du_n]$$

Here, $dl_i$ and $du_i$ denote the offsets with respect to the lower and upper bound of dimension $i$. Finally, the overlap description of a statement $S$ with respect to a right-hand side reference $ref$, $OD(S,ref)$, is defined as the contribution of $S$ and $ref$ to $OD(A)$.

**Example 1** Consider Figure 4. The overlap descriptions satisfy $OD(F) = [0 : 0, 0 : 0]$ and $OD(U) = [1 : 1, 1 : 1]$; while the overlap area of array $F$ is empty, that of array $U$ consists of an area of depth 1 around the distribution segment. For example, $\lambda^U(R(1, 2)) = U(1 : 8, 9 : 16)$. The associated extension segment is $U(0 : 9, 8 : 17)$, and the overlap area is $U(0 : 9, 8 : 16) \cup U(9, 9 : 16) \cup U(1 : 8, 8) \cup U(1 : 8, 17)$. □

![Figure 5: Overlap area for array $U$ in Jacobi code ($M \geq 4$)](image)

The overlap concept is not an appropriate approach if there is any form of indirection in the code. In such a case, alternative methods must be employed. For a further discussion of this, see Section 6.

The overlap concept can be used to organize communication: the exchange statement $EXCH0$ is replaced by $EXCH$, which is similar except that it refers to overlap descriptions $OD(S,ref)$ rather than statement masks.

The new version of Jacobi is given in Figure 6, the local declarations are omitted.
DO  J = 2, N-1
DO  I = 2, N-1
   CALL EXCH(U(I-1,J),[I:0:0:0])
   CALL EXCH(U(I+1,J),[0:1:0:0])
   CALL EXCH(U(I,J-1),[0:0:1:0])
   CALL EXCH(U(I,J+1),[0:0:0:1])
   IF OWNED(UNEW(I,J)) THEN
      S: UNEW(I,J) = 0.25 * (F(I,J) + U(I-1, J) + U(I+1, J) + U(I, J-1) + U(I, J+1))
   ENDIF
ENDDO
ENDDO

Figure 6: Program Jacobi with EXCH-based communication

3.4.2 Optimization of Masking and Communication

The modified exchange statement forms the basis for a more advanced optimization of communication: we
generalize EXCH to an aggregate communication primitive that moves blocks of data rather than single
objects. The communication for a data item can be moved out of a loop if no true dependence is violated.
Loop distribution and vectorization [44] can then be applied to generate aggregate communication. Further
optimization of communication includes fusion and elimination of redundant communication, as
described in the literature [16, 14, 27, 45]. A more general approach – orthogonal to the overlap concept – is
discussed in ([27]).

After initial masking, all processors execute the same masked statement sequence. For each masked
statement, each processor first evaluates the mask and executes the corresponding (unmasked) statement
instance if the mask yields true. The following transformations optimize the handling of masks in loops and,
in many cases, lead to the strip-mining [44] of the loop across the processors by partitioning the iteration
space:

- iteration elimination deletes irrelevant loop iterations for a process, and
- mask simplification eliminates statement masks that are true for each instance of each process.

A more general discussion of mask optimization and the associated work distribution can be found in
[17]. We again illustrate these optimizations as well as the target code generation with the example program.

Figure 7 gives the final version of the Jacobi iteration. The program is parameterized in terms of the
executing processor, p. It is assumed that \lambda = U(\$L1(p) : \$L2(p) : \$U1(p) : \$U2(p)). Hence, for example,
\$L2(R(1,2)) = 9 and \$U2(R(1,2)) = 16. The local declarations reserve space for the extension segment of
U. To simplify matters, we assume here that the target language can handle expressions in array bound
declarations, so that global addressing is possible.

The execution of the first exchange statement, CALL EXCH(U(1:N-2,2:N-1),[1:0:0:0]), in a processor \(p\)
has the following effect: first, all elements of \(U(1:N-2,2:N-1)\) that are owned by \(p\) and belong to the
corresponding overlap area of another processor \(p'\), are sent to \(p'\). Secondly, all elements of \(U(1:N-2,2:N-1)\)
that are in \(OA(U,p)\) are received from the respective owner. For example, \(R(2,1)\) sends \(U(9:14,8)\) to \(R(2,2)\), and
\(R(1,2)\) receives \(U(1:8,8)\) from \(R(1,1)\). These communication operations can be executed in parallel.
PARAMETER (N=16)
REAL UNEW($L1(p)$,$U1(p)$,$L2(p)$,$U2(p)$)
REAL U($L1(p)$,$U1(p)$+$1$,$L2(p)$,$U2(p)$+$1$)
REAL F($L1(p)$,$U1(p)$,$L2(p)$,$U2(p)$)
...
CALL EXCH(U(1:N-2,2:N-1),[1:0,0:0])
CALL EXCH(U(1:N,2:N-1),[0:1,0:0])
CALL EXCH(U(2:N-1,1:N-2),[0:0,1:0])
CALL EXCH(U(2:N-1,3:N),[0:0,0:1])
DO J = MAX(2,$L1(p)$),MIN(N-1,$U1(p)$)
   DO I = MAX(2,$L2(p)$),MIN(N-1,$U2(p)$)
      $S$: UNEW(I,J) = 0.25 * (F(I,J) + U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1))
   ENDDO
ENDDO
...

Figure 7: Jacobi – Final Version: Code for Processor p.

For each $p$, $S$ is executed exactly for those values of $I$ and $J$ that satisfy $\text{MAX}($$L1(p)$,2) \leq I \leq \text{MIN}($$U1(p)$,N - 1) and $\text{MAX}($$L2(p)$,2) \leq J \leq \text{MIN}($$U2(p)$). For these iterations, the mask can be eliminated; all other iterations can be eliminated for $p$.

3.5 Interprocedural Distribution Propagation

In SUPERB, the distributions of formal parameters are inherited from the corresponding argument distributions. The SPMD paradigm is applied to procedures in an obvious way: a procedure call in the node program is executed by all processors allocated to the program; masking and communication is performed as described above. Note that different incarnations of the same procedure may associate the same formal parameter with arrays that have different distribution characteristics. The code implementing the accesses to a formal array parameter has to take into account all possible cases for the corresponding argument. The quality of the code generated for procedures then depends critically on how much information is available at compile time about the distributions of formal parameters in different incarnations of the procedure, and on how efficiently run-time information can be organized.

SUPERB performs an interprocedural optimization strategy, based on the call graph of the program. For each incarnation of a procedure, a distribution vector is computed that establishes a binding between the formal parameters of the procedure and the distributions of the associated arguments. Conflicts are resolved by cloning: if the same formal parameter is bound to replicated as well as partitioned arguments in different iterations, copies of the procedure are automatically generated.

Details of this method are given in [15, 14, 45].

4 Vienna Fortran

We have already introduced the subset of Vienna Fortran on which SUPERB is based (Section 2). In this section, we provide an informal overview of the whole language, mainly based on examples. For a complete and precise description, see [43]. The reader is also referred to [7] for further examples of the use of these extensions and demonstration of their expressiveness.

---

4 The nodes and edges of the call graph respectively represent the procedures of the program and their calling relationship.
4.1 The PROCESSORS Statement and Processor References

The user may declare and name one or more processor arrays by means of the PROCESSORS statement. The first such array is called the primary processor array; others are declared using the keyword RESHAPE. They refer to precisely the same set of processors, providing different views of it: a correspondence is established between any two processor arrays by the column-major ordering of array elements defined in Fortran 77. Expressions for the bounds of processor arrays may contain symbolic names, whose values are obtained from the environment at load time. This allows the program to be parameterized by the number of processors. For example:

```
PROCESSORS MYP3(NP1, NP2, NP3) RESHAPE MYP2(NP1, NP2*NP3)
```

Processor arrays may be referred to in their entirety by specifying the name only. Array section notation, as introduced in Fortran 90, is used to describe subsets of processor arrays; individual processors may be referenced by the usual array subscript notation. Dimensions of a processor array may be permuted.

The number of processors on which the program executes may be accessed by the intrinsic function $\&NP$. A one-dimensional processor array, $\&P(1:$\&NP$)$, is always implicitly declared and may be referred to. This is the default primary array if there is no processor statement in a program. The index of an executing processor in $\&P$ is returned by the intrinsic function $\&MY\_PROC$.

4.2 Distribution Annotations

Distribution annotations may be appended to array declarations to specify direct and implicit distributions of the arrays to processors. Direct distributions consist of the keyword DIST together with a parenthesized distribution expression, and an optional TO clause. The TO clause specifies the set of processors to which the array(s) are distributed; if it is not present, the primary processor array is selected by default. A distribution expression consists of a list of distribution functions. There is either one function to describe the distribution of the entire array, which may have more than one dimension, or each function in the list distributes the corresponding array dimension to a dimension of the processor array. If there are fewer distributed dimensions in the data array than there are in the processor array, the array will be replicated to the remaining processor dimensions. Both intrinsic functions and user-defined functions may be used to specify the distribution of an array dimension.

The intrinsic functions provided by Vienna Fortran for the most common kinds of distributions are BLOCK, CYCLIC, and GENERAL\_BLOCK, all of which map a dimension of an array to a dimension of a processor array. CYCLIC maps array elements in a round-robin fashion to the corresponding dimension of the processor array. The following are further examples of Vienna Fortran array declarations annotated by a distribution:

```
PROCESSORS P2(NP,MP)
REAL XX(1000,100) DIST( CYCLIC(50), BLOCK )
REAL YY(10000) DIST( BLOCK ) TO $P$
INTEGER KK(500,50,5) DIST( BLOCK, CYCLIC, : ) TO P2/2,1/ 
```

Arrays XX and KK are distributed to P2; however, the dimensions have been permuted in the second case, so that the first dimension of KK is distributed by block to the second dimension of P2, and the second dimension of KK is scatter distributed to the first dimension of P2. YY is distributed to $\&P$, which has NP*MP elements in this case. Remember that the standard ordering of array elements defined in Fortran 77 may be applied to processor arrays, so that there is a well-defined relationship between the elements of $\&P$ and those of P2.

Implicit distribution, or alignment might be used, for example, to parallelize the following kernel as shown:
PARAMETER ( N = ...)
REAL ZX(N+12) DIST ( BLOCK )
REAL X(N),Y(N) ALIGN $ (I) WITH ZX(I + 10)
REAL Q, R, T
DO 11 K = 1, N
   X(K) = Q + Y(K)* ( R*ZX(K+10) + T* ZX(K+11) )
11 CONTINUE

The elements of arrays X and Y are aligned with the elements of array ZX in the example above: for each I from 1 through N, X(I) is mapped to the processor that owns element ZX(I+10). The $ symbol is merely a placeholder, indicating that multiple arrays are being aligned. Note that the scalar variables are replicated.

In practice, alignments can be used whenever there is a fixed relationship between two arrays that is of a very specific nature. In other situations it will generally suffice, or be more appropriate, to specify that data items are to be distributed “in the same way”. In the above, for example, the distribution of X and Y could have been expressed by giving them the same distribution function as ZX:

REAL X(N), Y(N) DIST (=ZX)

This distributes X and Y by block, with the appropriate block sizes. In this case, X and Y would be distributed evenly by block across the processors. Since they have fewer elements than ZX, the length of their blocks may be slightly smaller than the length of the blocks of ZX. When they are aligned with ZX as above, then the lengths of the first blocks of X and Y will be identical to those of ZX. However, the last processor will contain fewer elements of these arrays. For example, if N = 100 and the data is distributed to 4 processors, then the second distribution would distribute 25 elements of X and Y to each processor, whereas the alignment with ZX would result in the mapping of 28 elements of X and Y to the first three processors, and only 16 elements to the last of them. Thus the elements of X and Y are not spread evenly over the processors. It will depend very much on the nature of our computation which of these distributions performs better.

Note that if we choose to distribute X and Y in the same way as ZX, we could actually distribute them all by one single declaration in this case. But that would not be true in a subroutine: when, say, ZX is a dummy argument whose distribution is not known. Both alignment and the referral to the distribution of other arrays are important in subroutines where information on the distribution of dummy arguments is incomplete.

Rather more complex distributions and alignments are required in many real applications. Many of them, such as arbitrary rectilinear block distributions are useful to the programmer and can be efficiently implemented.

The INDIRECT distribution intrinsic function enables the specification of a mapping array which allows each array element to be distributed individually to a single processor. The mapping array must be of the same size and shape as the array being distributed. The values of the given array are processor numbers (in $P):

INTEGER IAPROCS(1000)
REAL A(1000) DIST ( INDIRECT(IAPROCS) )

Thus, for example, the value of IAPROCS(60) is the number of the processor to which A(60) is to be mapped. Note that IAPROCS must be defined before it is used to specify the distribution of A, and that each element of A can be mapped to only one processor.
4.3 Dynamic Distributions and the DISTRIBUTE Statement

By default, the distribution of an array is static. Thus it does not change within the scope of the declaration to which the distribution has been appended. The keyword DYNAMIC is provided to declare an array distribution to be dynamic. This permits the array to be the target of a DISTRIBUTE statement. A dynamically distributed array may optionally be provided with an initial distribution in the manner described above for static distributions. A range of permissible distributions may be specified when the array is declared by giving the keyword RANGE and a set of explicit distributions. If this does not appear, the array may take on any permitted distribution with the appropriate dimensionality during execution of the program. Finally, the distribution of such an array may be dynamically connected to the distribution of another dynamically distributed array in a specified fixed manner. This is expressed by means of the CONNECT keyword. Thus, if the latter array is redistributed, then the connected array will automatically also be redistributed.

```
REAL F(200,200) DYNAMIC, RANGE({ BLOCK, BLOCK}, { CYCLIC(5), BLOCK })
```

The distribute statement begins with the keyword DISTRIBUTE and a list of the arrays which are to be distributed at runtime. Following the separator symbol “;”, a direct, implicit or indirect distribution is specified using the same constructs as those for specifying static distributions. It has an optional NOTRANSFER clause; if it appears, then it specifies that the arrays to which it applies are to be distributed according to the specification, but that old data (if there is any) is not to be transferred. Thus only the access function is modified. For example:

```
DISTRIBUTE A, B :: { CYCLIC(10) } NOTRANSFER(B)
```

In the above statement, both arrays A and B are redistributed with the new distribution CYCLIC(10), however for the array B only the access function is changed, the old values are not transferred to the new locations. Whenever an array is redistributed via a distribute statement, then any arrays connected to it are also automatically redistributed to maintain the relationship between their distributions.

Distribution Queries and The DCASE Construct  The DCASE construct enables the selection of a block of statements for execution depending on the actual distribution of one or more arrays. It is modeled after the CASE construct of Fortran 90. The keywords “SELECT DCASE” are followed by one or more arrays whose distribution functions are queried. The individual cases begin with the keyword “CASE” together with a distribution expression for each of the selected arrays. The distribution expressions consist of one or more distribution functions (which may contain arguments such as a length), or a “**” which matches any distribution. The distribution of an array is matched only if it is matched in all dimensions. The first case which satisfies the actual distributions of the selected arrays is chosen and its statements executed. No more than one case may be chosen.

```
SELECT DCASE (A, B)
  CASE ( BLOCK), ( BLOCK)
    CALL BLOCKSUB(A,B,N,M)
  CASE ( BLOCK), ( CYCLIC)
    ...
  CASE DEFAULT
    ...
END SELECT
```

The distributions of two different arrays may be compared in a similar manner within an IF statement.
4.4 Common Blocks

Common blocks in which no data is explicitly distributed have the same semantics as in Fortran 77. The common block storage sequence is defined for them. Individual arrays which occur in a named common block may also be explicitly and individually distributed just as other arrays are. However, they may not be dynamically distributed. Once storage space has been determined for a named common block, then it may not change during program execution. Note that, in accordance with Fortran 90, allocatable arrays may not be in common blocks.

4.5 Using Subroutines in Vienna Fortran

We discuss the main issues which arise when subroutines\(^5\) are invoked with distributed arguments by, again, looking at a very simple example. This permits us to ignore the computational problem and concentrate on the situations a programmer will need to be able to deal with.

It is common practice to write subroutines for such operations as matrix multiplication, which are used frequently. In this section we consider how this is done in Vienna Fortran.

When a distribution annotation is appended to a declaration in Vienna Fortran, then that distribution has the same scope as the declaration itself. In a subroutine, both local arrays and dummy array arguments may be given an explicit distribution when they are declared. As we will see below, this makes the mechanism of appending distribution annotations to array declarations a very powerful tool, enabling a controlled redistribution of data.

One version of a subroutine to multiply matrices in Vienna Fortran is as follows:

```fortran
SUBROUTINE MATMUL(A,B,C,N,M,L)
REAL A(N,M),B(M,L), C(N,L) DIST(*)

DO 30 I = 1, N
   DO 30 J = 1, L
      C(I,J) = 0.0
   DO 30 K = 1, M
      C(I,J) = C(I,J) + A(I,K)*B(K,J)

30 CONTINUE
RETURN
END
```

In this routine we employ the additional method for specifying distributions which can be used for dummy array arguments only. If a "*" is used to specify the distribution, then the dummy argument inherits the distribution of the actual array. This means that each time the above routine is called, the actual arguments may be distributed differently to the processors. Interprocedural distribution analysis will often reveal the distribution functions which reach the subroutine, and the compiler is then able to generate code based on that information. This is a flexible way to write subroutines. But an unfortunate consequence of using inherited distributions is that the compiler may not always have precise (or, if it is separately compiled, any) information on the actual distributions which may reach the dummy arguments. In cases where this analysis fails, there is a way of providing extra help. If the user knows that only a few distributions will occur, then this information may be provided in a RANGE clause which is appended to the distribution. For example, the specification:

```fortran
REAL A(N,M) DIST(*), RANGE(( BLOCK, BLOCK),( BLOCK, CYCLIC(100)))
```

\(^5\)We will not examine functions separately; they can be written similarly.
declares that only the distributions, \((BLOCK,BLOCK)\) and \((BLOCK,CYCLIC(100))\) are allowed for the dummy argument \(A\).

Further, the efficiency of the computation within the subroutine may depend very heavily on the actual distributions of the arguments, thus yielding good performance in some cases and very poor performance in others.

An alternative implementation might explicitly enforce the distribution of a dummy argument. We may write, for example:

```fortran
SUBROUTINE MATMUL(A,B,C,N,M,L)
REAL A(N,M), C(N,L) DIST(BLOCK) TO $P
REAL B(M,L)
DO 30 I = 1, N
...
```

Now this subroutine also has three dummy argument arrays, two of which, \(A\) and \(C\), are distributed by block in the first dimension to all available processors whereas the third, \(B\), is replicated. The dummy arguments are explicitly distributed in order to eliminate communication during the computation of the result. However, the actual arguments may not have the same distribution as the dummy arguments with which they are associated. When their distributions differ, they must be redistributed on entry to the subroutine to match the specified distribution. In general, their original distribution must also be restored on exit from the subroutine. Thus the efficient implementation of the computation within the subroutine has a price: the redistribution of actual arguments may sometimes be very costly.

We have seen the apparent difficulty in resolving two legitimate demands of a general purpose subroutine: that it handle a variety of different arguments, which may be differently distributed, on the one hand, and that it handle them efficiently on the other hand. Redistribution may be costly, yet we may want to implement the routine in a way that is handled optimally on the target machine. In this situation, the DCASE construct may be used:

```fortran
SUBROUTINE MMUL(A,B,C,N,M,L)
PARAMETER (MAXSIZE = ...)
REAL A(N,M),B(M,L), C(N,L) DIST(*)
INTEGER LEN, LSUB

SELECT DCASE (C,A):
  CASE (BLOCK, :),( BLOCK, :)
    IF (M*L .LE. MAXSIZE) THEN
      CALL MATMUL(A,B,C,N,M,L)
    ELSE
      LEN = L / $NP
      DO 45 J = 1, $NP
        CALL MATMUL1 (A,B,C,N,M,L,LEN,J)
      CONTINUE
    ENDIF
  CASE (BLOCK, BLOCK),( BLOCK, *)
    ...
  CASE DEFAULT
    ...
END SELECT
```

16
In the above, the matrix operation is handled in a specific way depending on how the actual argument arrays are distributed. In this way, we can insert appropriate code or call further subroutines as required. The compiler has precise information on the distribution functions of the selected arrays for the block of statements within the cases. Only one of the case alternatives is executed; if none of the other specifications match, then the default (if present) is selected. Here, the cases are examined in the order in which they occur textually. The first distribution expression is compared with the actual distribution of C, and the second with that of A. If C is distributed by block in the first dimension and not at all in the second, and A likewise, then the first case is selected and its code executed. Otherwise, the distribution of C is then compared with the next case: if it is distributed by block in both dimensions, then if A is distributed by block in the first dimension, this case is selected. An ‘*’ matches any distribution whatsoever.

4.6 The FORALL Loop

The FORALL loop enables the user to assert that the iterations of a loop are independent and can be executed in parallel. A precondition for the correctness of this loop is the absence of loop-carried dependences. There is an implicit synchronization at the beginning and end of such a loop. Private variables are permitted within forall loops; they are known only in the forall loop in which they are declared and each loop iteration has its own copy. The iterations of the loop may be assigned explicitly to processors if the user desires, or they may be performed by the processor which owns a specified datum. This can be done through the optional on clause specified in the forall loop header.

```
FORALL I = 1,N ON MASTER(C(K(I)))
  Y(K(I)) = X(I) + C(K(I))^2 Y(K(I))
END FORALL
```

```
FORALL I = 1,N ON $P(NOP(I))
  REAL T
    ...
END FORALL
```

(a) Parallel loop with indirect accesses   (b) Parallel loop with a private variable

Figure 8: Parallel loops.

The on-clause in the example shown in Figure 8a specifies that the i-th iteration (1 ≤ i ≤ N) of the loop is executed on processor MASTER(C(K(i))). The processor may also be specified explicitly, such as in ON R1(I), where R1 is a processor array. In the second parallel loop (Figure 8b) the on-clause directly refers to the implicit processor array, with the i-th iteration assigned to the k-th processor where k is the current value of the array element which is denoted by NOP(i). T is declared private in the forall loop. Logically there are N copies of the real T, one for each iteration of the loop. Thus assignments to such variables do not cause loop-carried dependences.

A reduction statement may be used within forall loops to perform such operations as global sums (cf. ADD below); the result is not available until the end of the loop. The user may also define reduction functions for operations which are commutative and associative in the mathematical sense. The intrinsic reduction operators provided by Vienna Fortran are ADD, MULT, MAX and MIN. The forall loop in Figure 9a results in the values of the array A being summed and the result being placed in the variable X. In each iteration of the forall loop in Figure 9b, elements of D and E are multiplied, and the result is used to increment the corresponding element of B. In general, all arrays B, D, E, X, Y, and Z can be distributed.

4.7 Input/Output

Files read/written by parallel programs may be stored in a distributed manner or on a single storage device. We provide a separate set of I/O operations to enable individual processors access to data stored across several

---

6Note that the forall loop, as introduced here, is not the forall loop proposed during the development of Fortran 90 and in HPF.
FORALL \( I = 1, N \) ON OWNER(A(I))

\[ \cdots \]

REDUCE (ADD, X, A(I))

\[ \cdots \]

END FORALL

FORALL \( I = 1, N \) ON OWNER (B(X(I)))

\[ \cdots \]

REDUCE(ADD, B(X(I)), D(Y(I))*E(Z(I)))

\[ \cdots \]

END FORALL

(a) Summing values of a distributed array

(b) Accumulating values onto a distributed array

Figure 9: Applying reduction statements.

devices. These operations can be classified into three groups: data transfer, inquiry and file manipulation. They can be used by the programmer to provide information which will allow the compiler and runtime environment to optimize the transfer of data to and from secondary storage. Details can be found in [5, 43].

5 Vienna Fortran and HPF

Recently, a consortium of researchers from industry, government labs and academia formed the High Performance Fortran Forum to develop a standard set of extensions for Fortran 90 which would provide a portable interface to a wide variety of parallel architectures. The forum has produced a draft proposal for a language, called High Performance Fortran (HPF) [21], which focuses mainly on issues of distributing data across the memories of a distributed memory multiprocessor.

The main concepts in HPF have been derived from a number of predecessor languages, including mainly DINO [35], CM Fortran [39], Kali [28], Fortran D [13], and Vienna Fortran, with the last two languages having the largest impact.

The basic elements of HPF’s language model are – similarly to Vienna Fortran – abstract processors, distributions, and alignments. In addition, HPF has introduced the concept of a template, which is essentially a named index domain that can be used as an alignment base. The implications of this construct which significantly complicates the underlying semantic model are discussed in [9].

HPF follows Vienna Fortran closely in a number of features. This includes in particular

- abstract processor arrays
- direct distribution and alignment of arrays
- distinction between static and dynamic distributions
- definition of the procedure interface, in particular inherited and enforced distributions
- FORALL loops (called independent loops in HPF)

On the other hand, a number of advanced concepts of Vienna Fortran have not been included in HPF. Among them are

- different processor views
- distribution of arrays to processor sections
- GENERAL, BLOCK and INDIRECT distributions
- user-defined distribution functions
- return of distributions from a procedure call
• parallel I/O

These omissions, in particular the absence of language features for the formulation of arbitrary distribution functions, significantly impairs the applicability of HPF to advanced algorithms using, for example, irregular or adaptive grids.

Table 1 below compares Fortran D, Vienna Fortran, and HPF with respect to a selected set of language features. We use the terminology introduced in this paper.

6 FORALL Loops and Run-Time Analysis

Compilers for DMMPs cannot generate efficient code for loops with irregular accesses as they arise in sparse and unstructured problems, using only compile-time techniques.

For example, in Figure 8a, the array K serves as a pointer array, whose values are determined at run time. As a consequence, the access pattern associated with the use of C and Y cannot be statically analyzed. The compiler can not generate communication statements nor can it determine which processor must execute a specific iteration of the loop. Moreover, since it cannot be decided whether or not separate loop iterations assign values to the same element of array Y, this loop would be executed sequentially if written as a standard DO loop. Thus an explicit language construct or directive is essential to enable parallel execution of code containing one or more levels of indirection. For this aim, Vienna Fortran provides the forall loop construct. Several different kinds of explicitly parallel loop have hitherto been proposed. Their semantics differ; however, they do share the common purpose of giving the programmer a means of enforcing parallel execution of loop iterations.

We describe below a strategy for handling a forall loop which can deal with those cases where compile time analysis fails to provide sufficient information to construct iteration and communication sets. First, the set of iterations which will be executed on each processor must be determined, in accordance with the specification in the ON clause. All non-local values read by a processor may be communicated prior to actual loop execution, since the loop will not modify them. Hence communication may be extracted from the loop and combined. A processor may also assign to non-local variables within a forall loop: hence temporary storage must be reserved for these, either in an overlap area or a separate buffer. The new values may then be communicated to the processors which own them once the loop has terminated.

The strategy discussed here generates two code phases for forall loops, known as the inspector and the executor. The inspector analyzes the communication patterns of the loop, while the executor performs the actual communication and executes the loop iterations. Hence in this context, the term inspector-executor paradigm is frequently used.

We base our treatment of this topic on work by Saltz, Mehrotra, and Koelbel, in particular on Koelbel’s Ph.D. Thesis [23], which provides a detailed discussion. Similar techniques have been applied in the Oxygen compiler ([36]), which also has implemented run-time dependence analysis to handle arbitrary loops containing irregular accesses.

In the following, we denote by SENDS(p, p') the set of elements that must be sent from processor p to processor p' as a consequence of non-local uses in p'. We let RECEIVES(p, p') be the set of elements that must be received by p from p' as a consequence of non-local uses in p. Clearly, SENDS(p, p') = RECEIVES(p', p) for each pair of processors. The set of iterations to be performed on processor p will be called ITS(p). We use the term local iteration for each iteration which does not use any non-local values; the set of these for a processor p is LITS(p). The remaining iterations are termed non-local iterations.

6.1 The Inspector

The inspector implements the first phase in the execution of a forall loop by analyzing the loop and computing the communication and iteration sets. Figure 10 specifies the code of the inspector executed in a
<table>
<thead>
<tr>
<th>LANGUAGE FEATURE</th>
<th>Fortran D</th>
<th>Vienna Fortran</th>
<th>HPF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processors</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstract processor declarations</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>processor views</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>distribution to processor sections</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Distribution and Alignment Mechanism</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direct distribution of arrays</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>direct alignment of arrays</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>templates</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>static distributions</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>dynamic distributions</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>dynamic distribution control</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Distribution Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block distributions</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>cyclic distributions</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>total replication</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>INDIRECT</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>user-defined distribution functions</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Procedure Interface</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inherited distributions</td>
<td>U</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>enforced distributions</td>
<td>U</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>procedures returning distributions</td>
<td>U</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Other Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORALL (independent) loops</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SIMD FORALL loops</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>parallel I/O</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>pure procedures</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>extrinsics</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>task parallelism</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Note:** “+” and “-” respectively indicate the presence or absence of a language feature, “U” stands for undefined.

Table 1: A comparison of selected language features in Fortran D, Vienna Fortran, and HPF
processor \( p \). We assume that the sets \( ITS(p) \) have been determined.

The inspector consists of two steps. The first step performs a dynamic loop analysis: a simplified version of the loop is executed, in which only those components are analyzed that are relevant for the generation of communication. For each iteration in \( ITS(p) \), it is determined whether or not it is a local iteration in the above sense. For each non-local iteration, the non-local uses and their associated processors are determined; this yields the sets \( RECEIVES(p,p') \) for all \( p' \in P \). The second step uses the relationship \( SENDS(p,p') = RECEIVES(p',p) \) to compute, as a result of a global communication phase, the sets of values which must be transmitted prior to execution of the loop.

Although it is superior to individual communications of values, the inspector phase nevertheless represents a significant run time overhead. It is therefore important to recognize situations in which one execution of the inspector suffices to organize the communication for multiple executions of the executor. This may happen when the same \( ON \) clause is used for several different loops. If arrays are not redistributed, then repeated executions of the same forall loop (at each time step, for example) will use the same iteration sets and the same communication structure; hence, the inspector phase need only be executed once.

<table>
<thead>
<tr>
<th>Step 1: Loop Analysis</th>
<th>Step 1: Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform a dynamic analysis of the loop to determine the local and non-local iteration sets, and, for all ( p' ), ( RECEIVES(p,p') ).</td>
<td>Whenever ( SENDS(p,p') \neq \phi ):</td>
</tr>
<tr>
<td></td>
<td>SEND ( SENDS(p,p') ) TO ( p' )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2: Computation of SENDS sets</th>
<th>Step 2: Execute Local Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>In a global communication phase, compute all sets ( SENDS(p,p') ) from the receive sets determined in Step 1.</td>
<td>For each ( p' \in P ) such that ( RECEIVES(p,p') \neq \phi ):</td>
</tr>
<tr>
<td></td>
<td>RECEIVE temps FROM ( p' )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3: Receive</th>
<th>Step 4: Execute Non-local Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each ( p' \in P ) such that ( RECEIVES(p,p') \neq \phi ):</td>
<td>Execute all remaining iterations</td>
</tr>
</tbody>
</table>

Figure 10: Inspector and executor code for processor \( p \).

### 6.2 The Executor

The **executor** is the second phase in the execution of the forall statement; it performs the actual computation. When the executor commences, all communication and iteration sets have been computed in the inspector. The executor consists of four steps (see Figure 10). The first of these sends all local data that are required by other processors. Then all local iterations are executed. In parallel (if the target machine permits this), non-local data required in one processor is received from the others and stored in a temporary buffer. Finally, the executor performs all iterations with at least one non-local use, accessing the corresponding buffer elements.

We have not shown the final stage of a forall loop: if values have been assigned to non-local data, then there must be a concluding communication phase which transmits these to all processors which own them.

### 6.3 Implementation

In the Vienna Fortran Compiler, implementation of the above inspector/executor strategy is based on the **Parti** routines ([38, 41]). The compiler automatically inserts them into the code when compiling programs with forall loops. Details can be found in [5]. Parti routines represent a concrete system that addresses the specific problems related to the fact that either distributions of data or data dependence patterns are determined by program data which is not known until run time. A library of primitives have been constructed to support the handling of irregularly distributed arrays, whose distributions are specified via mapping arrays (see Section 4.2).
Parti routines receive and transmit irregularly distributed non-local data, and perform reduction operations on them. The strategy adopted generalizes the inspector-executor paradigm discussed above.

In order to deal with array elements which are arbitrarily assigned to processors, a translation table is constructed from the mapping array, recording the owner of each datum and its local index. It is generally itself distributed in a regular fashion across the executing processors. In Step 1 of the inspector, communication has to be performed to find out which processor a data item is stored on, and to get the local index. In view of the expense of these operations, a good deal of work has gone into optimizing the handling of these arrays: local copies of non-local data are stored using hash tables in a so-called hashed cache. This helps eliminate duplicate fetches. Methods to keep track of the location of non-local data and reuse values already received have been developed on the basis of incremental communication schedules, which fetch only those data items which are not already available locally – this often proves useful for sequences of loops where reuse of data items is not uncommon.

To support efficient implementation of irregular computations that deal with arrays distributed regularly, special Parti primitives have been developed.

7 Related Work

An early attempt to provide higher-level language constructs for the specification of numerical algorithms on DMMPs is DINO [34, 35]. DINO is explicitly parallel, providing a set of C language extensions. Non-local data may be read and written; thus DINO does not conform to the owner computes paradigm. Remote accesses are marked by the user. DINO has been fully specified and implemented.

The description of SUPERB in [42] is the first journal publication in the area of compiling Fortran for DMMPs. Callahan and Kennedy propose a similar compilation approach in [6].

The concept of defining processor arrays and distributing data to them was first introduced in the programming language BLAZE [25] in the context of shared memory systems with non-uniform access times. This research was continued in the Kali programming language [28] for distributed memory machines, which requires that the user specify data distributions in much the same way that Vienna Fortran does. It permits both standard and user-defined distributions. The design of Kali has greatly influenced the development of Vienna Fortran. In particular, the parallel FORALL loops of Vienna Fortran were first defined in Kali and implemented with the inspector-executor paradigm as described in Section 6.

The Parti routines and the ARF compiler ([41, 38]), developed by Saltz and co-workers at ICASE, represent techniques developed to handle the kind of codes written for sparse and unstructured problems in scientific computing. They are designed to handle the general case of arbitrary data mappings, and efficient techniques were developed for a number of subproblems.

A commercially available system is the MIMDizer ([30]) which may be used to parallelize sequential Fortran programs according to the SPMD model. The MIMDizer takes a similar approach to SUPERB; it deals with a number of specific Fortran issues, including a very flexible handling of common blocks.

The programming language Fortran D [13] proposes a Fortran language extension in which the programmer specifies the distribution of data by aligning each array to a decomposition, which corresponds to an HPF template (see Section 5), and then specifying a distribution of the decomposition to a virtual machine. These are executable statements, and array distributions are dynamic only. A subset of Fortran D – roughly corresponding to SUPERB – has been implemented for the iPSC/860 [26].

The source language for the Crystal compiler built by Li and Chen at Yale University ([26]) is the functional language Crystal, which includes constructs for specifying data parallelism. Thus there is a cer-
tain amount of parallelism explicit in the original code. Experimental compilers have been constructed for
the iPSC hypercube and the nCUBE; they place particular emphasis on an analysis of the communication
requirements to generate efficient communication.

Dataparallel C ([19]) is a SIMD extension of the C language which is a slightly modified version of
the original C* for the Connection Machine. Like DINO, it is explicitly parallel and requires the user to
specify a local view of computations. Dataparallel C compilers have been constructed for both shared and
distributed memory machines.

Cray Research Inc. has announced MPP Fortran [32], a set of language extensions to Cray Fortran which
enable the user to specify the distribution of data and work. They provide intrinsics for data distribution
and permit redistribution at subroutine boundaries. Further, they permit the user to structure the executing
processors by giving them a shape and weighting the dimensions. Several methods for distributing iterations
of loops are provided.

In the Cray programming model, many of the features of shared memory parallel languages have been
retained: these include critical sections, events and locks. New instructions for node I/O are provided.

Other systems include AL, which has been implemented on the Warp systolic array processor [40],
Pandore, a C-based system [2], Id Nouveau, a compiler for a functional language [33], Oxygen [36],
ASPAR [22], Adapt, developed at the University of Southampton [29], and the Yale Extensions [10]. In
a few systems, dynamic data distributions have been implemented within narrow constraints [3, 2].

The systems described above are not the only efforts to provide either suitable language constructs
for mapping code onto DMMPs or to generate message passing programs from higher-level code. Other
important approaches include Linda [1], Strand [12], and Booster [31].

8 Conclusion

In this paper, we have outlined the compilation strategy of the parallelization system SUPERB, illustrated
the main features of Vienna Fortran, and described a compilation strategy handling irregular data accesses
by employing run-time analysis.

There are still many challenges facing the builders of compilation systems for DMMPs. Much improve-
ment is needed with regard to functionality, the efficiency of the generated code, and the quality of the
interaction with the user. In particular:

- **The functionality of systems must be improved to**
  - handle “real” codes
  - support automatic data distribution
  - deal efficiently with dynamic and irregular data and work distributions
  - handle parallel I/O
  - provide task parallelism

- **New compilation methodologies must be developed**, including
  - global program transformation strategies
  - knowledge-based techniques
  - rule-based compilation

- **Integrated programming environments must be built**. Their elements include – in addition to
  the **transformation component** as present in conventional systems – tools for
– performance analysis and prediction
– high-level user interfaces
– graphics support
– high-level test and debugging support

At many research sites around the world, work is underway in all these areas.

Finally, the current HPF language [21] is a first attempt for standardization in the area of languages for DMMPs. However, as has been pointed out in Section 5, only modest goals have been reached by now. We believe that Vienna Fortran and the associated implementation experience will provide a good starting point for the continuation of the standardization process that started with HPF.

References


