High Performance Fortran: Current Status and Future Directions *

Barbara Chapman[a] *Piyush Mehrotra*[b] *Hans Zima*[a]

[a]Institute for Software Technology and Parallel Systems,
University of Vienna, Brünner Strasse 72, A-1210 Vienna, Austria
E-Mail: {barbara, zima}@par.univie.ac.at

[b]ICASE, MS 132C, NASA Langley Research Center
Hampton VA. 23681 USA
E-Mail: pm@icase.edu

Abstract

High Performance Fortran (HPF) was defined with the objective of providing support for the development of efficient data parallel programs for distributed-memory architectures. We believe that the current version of the language has failed to reach this goal to a sufficient degree. While the basic distribution functions offered by the language can support regular numerical algorithms, advanced algorithms such as particle-in-cell codes or unstructured mesh solvers cannot be expressed adequately. This paper discusses some of the data distribution and alignment issues, and outlines possible future paths of development. Emphasis will be placed on language features that support dynamic and irregular distributions, and related implementation strategies developed in cooperation between Joel Saltz's group at the University of Maryland and the Vienna Fortran group.

1 Introduction

High Performance Fortran (HPF) [5] provides language extensions for Fortran 90 to support data parallel programming on a range of parallel architectures. The language contains

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF Grant P8989-PHY), by the Austrian Ministry for Science and Research (BMWF Grant GZ 308.9281-IV/3/93), and by the ESPRIT project 6643 PPPE – Portable Parallel Programming Environment. This research was also supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

directives for specifying alignment and distribution of a program's data. These enable the programmer to influence the locality of computation by controlling the manner in which the data is mapped to processors. Other major extensions include data parallel constructs, such as the **FORALL** statement and construct and the **INDEPENDENT** directive, and a number of library routines. Even though the defined extensions provide the first steps towards a portable programming interface, it is our contention that they fall short of the overall goal.

In this paper we show that there is some important functionality which is missing from the current HPF language definition. We will focus on features for data distribution and alignment.

Much of HPF consists of constructs which may be used to specify the mapping of data in the program to an abstract set of processors. This is achieved via a two-level mapping: first, data arrays are aligned with other objects, and then groups of objects are distributed onto an array of abstract processors. These processors are then mapped to the physical processors of the target machine in an implementation-dependent manner.

The **ALIGN** directive is used to align elements of data arrays to other data arrays or *templates*. Templates are abstract index spaces which can be used as a target for alignment and may then be distributed in the same way as arrays. The **DISTRIBUTE** directive is provided to control distribution of the dimensions of arrays or templates onto an abstract set of processors. The distribution of a dimension may be described by selecting one of a set of predefined primitives which permit block, cyclic or block-cyclic mapping of the elements. The rules for alignment are more flexible: a linear function may be used to specify the relationship between the mappings of two different data arrays. Mechanisms are also provided to enable dynamic modification of both alignment and distribution; these are the **REALIGN** and **REDISTRIBUTE** directives, respectively.

These language constructs suffice for the expression of a range of numerical applications operating on regular data structures. However, more complex applications pose serious difficulties. For example, modern codes in such important application areas as aircraft modeling and combustion engine simulation often employ multiblock grids. Even if these grids are to be distributed by block to processors, the constructs of the current HPF language specification do not permit an efficient data mapping for these programs: this requires distributions to sections of the processor array in general [3]. HPF is even less equipped to handle advanced algorithms such as particle-in-cell codes, adaptive multigrid solvers, or sweeps over unstructured meshes. Many of these problems need more complex data distributions if they are to be executed efficiently on a parallel machine. Some of them may require the user to control the execution of major do loops by specifying which processor should perform a specific iteration. These are not provided by HPF.

Some programs will require that data be distributed onto processor arrays of different dimensions: the current language definition does not permit the user to prescribe or assume any relationship between such processor arrays.

Language extensions which provide the required functionality include

- distribution to processor subsets

- processor views
- general block distributions
- indirect distributions
- user-defined distribution functions, and
- on-clauses for the control of the work distribution in an **INDEPENDENT** loop.

In this paper, we indicate how such features might be added to the language and give a number of examples to illustrate their use. The set of all features that we propose for inclusion into the current version of HPF will be informally subsumed under the name **HPF**$^+$. Our prime consideration is functionality and semantics; we do not attempt to provide a full definition of the proposed features, and sometimes use an ad-hoc syntax. In a few places, we use syntax from Vienna Fortran [2, 11], which already provides solutions for some of the problems discussed in this paper.

We assume throughout that the reader is familiar with the basic mechanisms of HPF for mapping data. The full details are to be found in the HPF Language Specification [5].

The next section deals with irregular data distributions, introducing indirect distributions (Section 2.1) and user-defined distribution functions (Section 2.2) as two different approaches. In Section 3, we discuss on-clauses in relationship with **INDEPENDENT** loops. The two subsequent sections illustrate two approaches – on different levels of abstraction – to the problem of formulating sweeps over unstructured meshes. Concluding remarks are to be found in Section 6.

2 Irregular Distributions

Dimensions of data arrays or templates are mapped in HPF by specifying one of a small number of predefined distributions, possibly with an argument. There are a number of applications, for example particle-in-cell codes, for which these mappings are inadequate since they do not provide adequate balancing of the work load. One possible solution for this problem is a generalization of the block distribution.

General block distributions, as initially implemented in SUPERB [10] and Vienna Fortran [2, 11], are similar to the regular block distributions of HPF in that the index domain of an array dimension is partitioned into contiguous blocks which are mapped to the processors; however, the blocks are not required to be of the same size. Thus, general block distributions provide more generality than regular blocks while retaining the contiguity property, which plays an important role in achieving target code efficiency. An example illustrating their use for the solution of a particle-in-cell code is given in [3].

General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of an unstructured mesh are partitioned prior to execution and then appropriately renumbered, then the resulting distribution can be described in this manner. However, this approach is not appropriate for each irregular

problem. For example, a data distribution as shown in Figure 1 – which may be the outcome of a dynamic partitioner – cannot be represented in this way. A system based on this kind of distribution was developed by Baden [1].

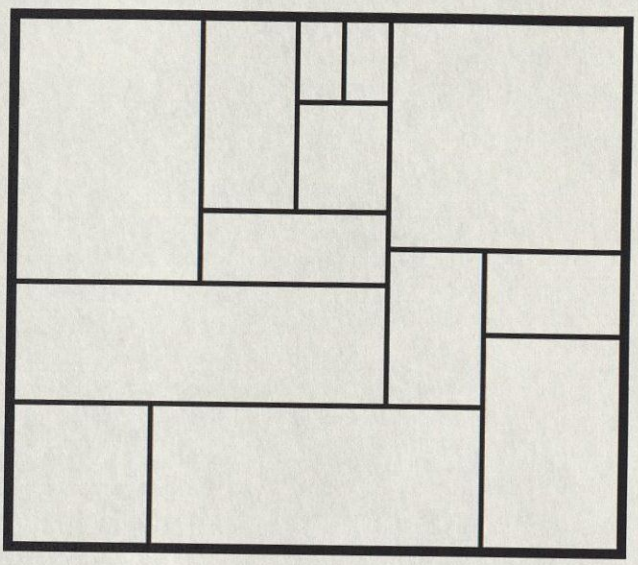

Figure 1: An example for an Irregular Block Distribution

Rather than proposing a special syntax for this kind of distribution, we will in the following subsections deal with a range of mechanisms, at different levels of abstraction, to handle arbitrarily complex data distributions.

We begin with *indirect distribution functions*, which allow the specification of a distribution via a mapping array (Section 2.1), and continue with *user-defined distribution functions* (Section 2.2). After discussing an extension of HPF's **INDEPENDENT** loop concept (Section 3), we give an example of a sweep over an unstructured mesh, based on all three extensions (Section 4). We conclude with the discussion of language features that directly support the binding of partitioners to **INDEPENDENT** loops (Section 5).

2.1 Indirect Distributions

Indirect distribution functions can express *any* distribution of an array dimension that does not involve replication. Consider the following program fragment in HPF+:

```
!HPF$ PROCESSORS R(M)
      REAL A(N)
      INTEGER MAP(N)
      ...
!HPF$ DYNAMIC, DISTRIBUTE (BLOCK) :: A
!HPF$ DISTRIBUTE (BLOCK) :: MAP
      ...
!  Compute a new distribution for A and save it in the mapping array MAP:
```

```
! The j-th element of A is mapped to the processor whose number is stored in MAP(j)

      CALL PARTITIONER(MAP, A,...)
      ...
! Redistribute A as specified by MAP:
!HPF$ REDISTRIBUTE A(INDIRECT(MAP))
      ...
```

Array A is dynamic and initially distributed by block. MAP is a statically distributed integer array that is of the same size as A and used as a **mapping array** for A; we specify a reference to an indirect distribution function in the form *INDIRECT(MAP)*. When the reference is evaluated, all elements of MAP must be defined and represent valid indices for the one-dimensional processor array R, i.e., they must be numbers in the range between 1 and M. A is then distributed such that for each $j, 1 \leq j \leq N$, $A(j)$ is mapped to $R(MAP(j))$. In this example, MAP is defined by a **partitioner**, which will compute a new distribution for A and assign values to the elements of MAP accordingly. (This distribution will often be used for a number of arrays in the program).

Indirect arrays were introduced in [7, 9]. They must be supported by a runtime system, which manages the internal representation of the mapping array and handles accesses to the indirectly distributed array. The mapping array is used to construct a *translation table*, recording the owner of each datum and its local index. Note that this representation has **O**(N) elements, on the same order as the size of the array. Most codes require only a very small number of indirect mappings (this is usually between 1 and 3 distinct mappings). The PARTI routines [9] represent a runtime library which directly supports indirect distribution functions, and has been integrated into a number of compilers.

2.2 User-Defined Distribution Functions

Indirect distribution functions incur a considerable overhead both at compile time and at runtime. A difficulty with this approach is that when a distribution is described by means of a mapping array, any regularity or structure that may have existed in the distribution is lost. Thus the compiler cannot optimize the code based on this complex but possibly regular distribution.

User-defined distribution functions (UDDFs) provide a facility for extending the set of intrinsic mappings defined in the language in a structured way. The specification of a distribution function introduces a class of *distribution types* by establishing mappings from (data) arrays to processor arrays.

UDDFs were first defined in Kali [7] and Vienna Fortran [11].

Syntactically, UDDFs are similar to Fortran functions; however, their activation results in the computation of a distribution rather than in the computation of a value. Apart from this, no side effects may occur as a result of executing these functions. UDDFs have two implicit formal arguments, representing the data array to be distributed and the processor array

to which the distribution is targeted. Specification statements for these arguments can be given using the keywords **TARGET_ARRAY** and **PROCESSOR_ARRAY**, respectively. Other local data structures may be declared as well. UDDFs may contain Fortran executable statements along with at least one *distribution mapping statement* which maps the elements of the target array to the processors.

UDDFs constitute the most general mechanism for specifying distributions: any arbitrary mapping between array indices and processors can be expressed, including partial or total replication. We illustrate their use by an example, representing indirect distributions. A more elaborate application, describing a skewed distribution, can be found in [3].

Example: A UDDF Specifying an Indirect Distributions

The distribution function *INDIRECT*, as introduced in the previous section, can be easily expressed by a UDDF as shown below. For simplicity we assume that A and MAP have the same shape.

```
!HPF$ DFUNCTION INDIRECT(MAP)
!HPF$ TARGET_ARRAY A(*)
!HPF$ PROCESSOR_ARRAY R(:)
!HPF$ INTEGER MAP(*)

!HPF$ DO I=1,SIZE(A)
!HPF$   A(I) DISTRIBUTE TO R(MAP(I))
!HPF$ ENDDO
!HPF$ END DFUNCTION INDIRECT
```

A facility similar to UDDFs can be provided for **user-defined alignment**; the (implicitly transferred) processor array of the UDDF must be replaced in this case by the source array to which the target array is to be aligned [11].

3 Extensions of the INDEPENDENT Loop Concept

Whenever a do loop contains an assignment to an array for which there is at least one indirect access within the loop, then the compiler will not be able to determine whether the iterations of the loop may be executed in parallel. Since such loops are common in irregular problems, and may contain the bulk of the computation, the user must assert the independence of the do loop's iterations.

For this, HPF provides the **INDEPENDENT** directive, which asserts that a subsequent do loop does not contain any loop-carried dependences [12], allowing the loop iterations to be executed in parallel. The **INDEPENDENT** directive may optionally contain a **NEW** clause which introduces *private* variables that are conceptually local in each iteration, and therefore cannot cause loop-carried dependences.

There are two problems with this feature:

- There is no language support to specify the **work distribution** for the loop, i.e., the mapping of iterations to processors. This decision is left to the compiler/runtime system.
- **Reductions**, which perform global operations across a set of iterations, and assign the result to a scalar variable, violate the restriction on dependences and cannot be used in the loop (note that HPF and Fortran 90 provide intrinsics for some important reductions).

The first problem can be solved by extending the **INDEPENDENT** directive with an **ON** clause that specifies the mapping, either by naming a processor explicitly or referring to the *owner* of an element. The concept of *on clause* was first introduced in Kali [7] and later adopted in Fortran D [4] and Vienna Fortran [2]; a similar proposal was discussed in the HPF Forum but not included in the language (see [5], Journal of Development, Section 11). For example, in

```
!HPF$ PROCESSORS R(M)
      ...
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), ...
      DO I = 1, NEDGE
       ...
```

iteration *I* of the loop is executed on the processor that owns the array element *EDGE(I,1)*. If we assume that *R(F(I))* is this processor, the above example could also be written in the form

```
!HPF$ PROCESSORS R(M)
      ...
!HPF$ INDEPENDENT, ON R(F(I)), ...
      DO I = 1, NEDGE
       ...
```

The second problem can be solved by extending the language with a reduction directive – which is permitted within such loops – and imposing suitable constraints on the statement which immediately follows it. It could be augmented by a directive specifying the order in which values are to be accumulated. Note that simple reductions could be detected by most compilers.

For example, in the code fragment below, the contributions *INCR(I)* of different iterations are accumulated in the variables *Y(EDGE(I,1))* (where *EDGE(I,1)* may yield the same value for different values of *I* – see Section 4 for an example with a geometric interpretation):

```
!HPF$ PROCESSORS R(M)
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), ...
      DO I = 1, NEDGE
```

```
      ...
!HPF$ REDUCTION
      Y(EDGE(I,1)) = Y(EDGE(I,1)) + INCR(I)
      ...
```

Vienna Fortran provides a language extension for reduction operations which is more general, but which is not a directive.

4 Sweep Over An Unstructured Mesh

We now illustrate some of the language features introduced above by reproducing a section of code from a two-dimensional unstructured mesh Euler solver.

The mesh for this code consists of triangles; values for the flow variables are stored at their vertices. The computation is implemented as a loop over the edges: the contribution of each edge is subtracted from the value at one node and added to the value at the other node.

Figure 2 illustrates one solution to this problem. The mesh is represented by the array *EDGE*, where $EDGE(I,1)$ and $EDGE(I,2)$ are the node numbers at the two ends of the Ith edge. The arrays X and Y represent the flow variables, which associate a value with each of the *NNODE* nodes.

Consider the distribution of the data across the one-dimensional array of processors, $R(M)$. Each of the arrays has to be dynamically distributed, since the mesh is to be distributed at runtime, in order to balance the computational load across the processors.

The array X is declared to be dynamically distributed with an initial block distribution. Further below, this array is distributed indirectly, using the mapping array *MAP*. The user-specified routine *PARTITIONER*, whose code has been omitted from the example, will generate a mesh partition and store it in *MAP*.

Y is also declared with the keyword **DYNAMIC** and is aligned to X. Whenever X is redistributed, Y is automatically redistributed with exactly the same distribution function.

Consider now how the array *EDGE* is used in the algorithm. Since the elements of *EDGE* are pointers to flow variables – in iteration I, $X(EDGE(I,1))$, $X(EDGE(I,2))$ and the corresponding components of X and Y are accessed – we relate the distribution of *EDGE* to the distribution of X and Y in such a way that *EDGE(I,:)* is mapped to the same processor as *X(EDGE(I,1))*.

This kind of relationship between data structures occurs in many codes, since a mesh is frequently described in terms of elements, whereas values are likely to be accumulated at the vertices. It can be simply expressed if we extend the **REDISTRIBUTE** directive as shown in the example.

The computation is specified using an extended **INDEPENDENT** loop. The work distribution is specified by the **ON** clause: the Ith iteration is to be performed on the processor that owns $EDGE(I,1)$.

```
      PARAMETER (NNODE = ...)
      PARAMETER (NEDGE = ...)
!HPF$ PROCESSORS R(M)
      ...
      REAL      X(NNODE), Y(NNODE)
      INTEGER   MAP(NNODE)
      REAL      EDGE(NEDGE,2)
      INTEGER N1, N2
      REAL   DELTAX
      ...
!HPF$ DYNAMIC :: X,Y,EDGE
!HPF$ DISTRIBUTE (BLOCK) :: X, MAP
!HPF$ ALIGN WITH X :: Y
!HPF$ DISTRIBUTE (BLOCK,*) :: EDGE
      ...
      CALL PARTITIONER(MAP,EDGE)
      ...
!HPF$ REDISTRIBUTE X(INDIRECT(MAP))
!HPF$ REDISTRIBUTE EDGE(I,:) ONTO R(MAP(EDGE(I,1)))
      ...
!HPF$ INDEPENDENT, ON OWNER (EDGE(I,1)), NEW (N1, N2, DELTAX)
      DO I = 1, NEDGE
       ...
       N1 = EDGE(I,1)
       N2 = EDGE(I,2)

       ...
       DELTAX = F(X(N1), X(N2))

!HPF$ REDUCTION
       Y(N1) = Y(N1) - DELTAX
!HPF$ REDUCTION
       Y(N2) = Y(N2) + DELTAX
      END DO
      ...
     END
```

Figure 2: Code for Unstructured Mesh in HPF$^+$

The variables $N1$, $N2$ and $DELTAX$ are private, so conceptually each iteration is allocated a private copy of each of them. Hence assignments to these variables do not cause flow dependencies between iterations of the loop.

For each edge, the X values at the two incident nodes are read and used to compute the contribution $DELTAX$ for the edge. This contribution is then accumulated into the values of Y for the two nodes. But since multiple iterations will accumulate Y values at each node, different iterations may write to the same array elements. As a consequence, we have indicated that these are reductions.

The dominating characteristic of this code, as far as its compilation is concerned, is that the values of X and Y are accessed via the edges, hence a level of indirection is involved. In such situations, either the mesh partition must be available to and exploitable by the compiler, or runtime techniques such as those developed in the framework of the *inspector-executor paradigm* [6, 9] are needed to generate and exploit the communication pattern.

5 Sweep Over Unstructured Mesh: Revisited

The code for the unstructured Euler solver discussed in the previous section represents a low-level approach to parallelization, in which the programmer assumes full control of data and work distributions, using the **ON** clause, and indirect distribution functions; a user-defined partitioning routine explicitly constructs a mapping array which can then be referred to.

This process may be further automated. Recent developments in runtime support tools and compiler technology, such as the CHAOS system developed at the University of Maryland [8] and integrated in the Vienna Fortran Compilation System, show how a higher level language interface may be provided in which control over the data and work distributions in an **INDEPENDENT** loop can be delegated to a combination of compiler and runtime system. We illustrate this approach by the example in Figure 3 which uses an ad hoc notation as follows.

The *use-clause* of the **INDEPENDENT** loop enables the programmer to select a partitioner from those provided in the environment (in the example, this is SPECTRAL_PART) and the arrays (in the example, X) to which it is to be applied. SPECTRAL_PART is called with implicit arguments specifying the iteration space of the **INDEPENDENT** loop and the data flow pattern associated with the use of X in the loop. The call has two effects: First, a new distribution is computed for X, and X along with its associated secondary array, Y, is redistributed accordingly. Secondly, a new work distribution is determined for the **INDEPENDENT** loop, based on the combined objectives of minimizing load imbalances and maximizing locality.

The actions described above represent an extension of the *inspector* in the inspector-executor paradigm [6, 9] and are performed *before* the actual execution of the loop. Note that X and Y have been implicitly redistributed as a result of the execution of the **INDEPENDENT** loop, and will retain that distribution after the loop has completed execution.

Other constructs may be useful in conjunction with the specification of a partitioner:

```
      PARAMETER (NNODE = ...)
      PARAMETER (NEDGE = ...)
!HPF$ PROCESSORS R(M)
      ...
      REAL      X(NNODE), Y(NNODE)
      REAL      EDGE(NEDGE,2)
      INTEGER   N1, N2
      REAL      DELTAX
      ...
!HPF$ DYNAMIC :: X,Y,EDGE
!HPF$ DISTRIBUTE (BLOCK) :: X
!HPF$ ALIGN WITH X :: Y
!HPF$ DISTRIBUTE (BLOCK,*) :: EDGE
      ...
!HPF$ INDEPENDENT, NEW (N1, N2, DELTAX), USE (SPECTRAL_PART(X))
!HPF$ DO I = 1, NEDGE
      ...
      N1 = EDGE(I,1)
      N2 = EDGE(I,2)

      DELTAX = F(X(N1), X(N2))

!HPF$ REDUCTION
      Y(N1) = Y(N1) - DELTAX
!HPF$ REDUCTION
      Y(N2) = Y(N2) + DELTAX
    END DO
    ...
    END
```

Figure 3: Code for Unstructured Mesh: Version 2

for example, the user may wish to specify the array whose usage within the loop should form the basis of the loop's work distribution. It may sometimes be desirable to restore the distribution of one or more of the newly partitioned arrays after the loop has executed. If the partitioner is to be invoked at intervals throughout the program's execution, a condition for its invocation may be needed; this may depend on the value of the loop variable or some other program variable. Finally, it may be be necessary to combine this approach with low-level control, in which case some means of accessing the map array implicitly constructed for irregularly partitioned arrays such as X should be provided.

Note that rather than attaching such attributes to a number of **INDEPENDENT** loops individually, language features can be defined that associate a partitioner with the whole program or a set of loops. We do not discuss their syntax here.

6 Conclusion

In this paper, we have discussed the capabilities of High Performance Fortran for expressing data parallel programs in an efficient manner. We claimed that current HPF does not provide sufficient functionality to implement these problems in an efficient manner. The paper proposes a range of language features, with a particular emphasis on the problem of distributing data and work to the processors of a machine.

References

[1] S. Baden. Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors. SIAM J. Sci. and Stat. Computation,12(1), January 1991.

[2] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming* 1(1):31-50, Fall 1992.

[3] B.Chapman,P.Mehrotra, and H.Zima. Extending HPF for Advanced Data Parallel Applications. IEEE Magazine on Parallel and Distributed Technology, Special Issue on High Performance Fortran. Also: Technical Report TR 94-7, Institute for Software Technology and Parallel Systems, University of Vienna (May 1994).

[4] 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.

[5] 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.

[6] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[7] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pp. 364–384. Pitman/MIT-Press, 1991.

[8] R. Ponnusamy, J. Saltz, A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Technical Report, UMIACS-TR-93-32, University of Maryland, April 1993.

[9] 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.

[10] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[11] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[12] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.