Scientific Programming 15 (2007) 45–65
IOS Press

From FORTRAN 77 to locality-aware high productivity languages for peta-scale computing

Hans P. Zima
Institute for Scientific Computing, University of Vienna, Austria and Jet Propulsion Laboratory, CA, USA and Institute of Technology, Pasadena, CA, USA
E-mail: zima@jpl.nasa.gov

Abstract. When the first specification of the FORTRAN language was released in 1956, the goal was to provide an "automatic programming system" that would enhance the economy of programming by replacing assembly language with a notation closer to the domain of scientific programming. A key issue in this context, explicitly recognized by the authors of the language, was the requirement to produce efficient object programs that could compete with their hand-coded counterparts.
More than 50 years later, a similar situation exists with respect to finding the right programming paradigm for high performance computing systems. FORTRAN, as the traditional language for scientific programming, has played a major role in the quest for high-productivity programming languages that satisfy very strict performance constraints. This paper focuses on high-level support for locality awareness, one of the most important requirements in this context. The discussion centers on the High Performance Fortran (HPF) family of languages, and their influence on current language developments for peta-scale computing. HPF is a data-parallel language that was designed to provide the user with a high-level interface for programming scientific applications, while delegating to the compiler the task of generating an explicitly parallel message-passing program. We outline developments that led to HPF, explain its major features, identify a set of weaknesses, and discuss subsequent languages that address these problems. The final part of the paper deals with Chapel, a modern object-oriented language developed in the High Productivity Computing Systems (HPCS) program sponsored by DARPA. A salient property of Chapel is its general framework for the support of user-defined distributions, which is related in many ways to ideas first described in Vienna Fortran. This framework is general enough to allow a concise specification of sparse data distributions. The paper concludes with an outlook to future research in this area.

1. Introduction

In 1954, more than fifty years ago, John Backus and his group at IBM Corporation began their work on "automatic programming systems" that resulted in the first specification of a high-level algorithmic language in 1956, the FORmula TRANslating system, FORTRAN [22]. At that point in time, programming, which was almost exclusively done in machine or assembly language, was considered a "complex, creative art that required human inventiveness to produce an efficient program" [4]. In view of the surprising similarities to the current situation in programming for high performance computing (HPC) systems, which is the main topic of this paper, it is useful to take a look at the motivation and goals of the original FORTRAN project, which are discussed in detail in John Backus' history paper [4].[1]

An important "factor which influenced the development of FORTRAN was the economics of programming in 1954. . . Programming and debugging accounted for as much as three quarters of the cost of operating a computer; and obviously, as computers got cheaper, this situation would get worse". Users were skeptical that efficient programming could be automat-

[1] All citations in this section are from this paper.

ISSN 1058-9244/07/$17.00

ed; as a consequence, the designers asserted "that if FORTRAN, during its first months, were to translate any reasonable scientific source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger". A revolution in programming could only happen if "*both* of the following requirements have been met: (a) a new kind of programming language, far more powerful than those of today, has been developed and (b) a technique has been found for executing its programs at not much greater cost than that of today's programs". As a result, the development of the compiler was considered an integral part of the design of the 'FORTRAN language, with an "emphasis on object program efficiency rather than on language design". This decision was crucial for the success of FORTRAN, and also paved the way for many other programming languages based on the von-Neumann machine model that were developed in subsequent years.

More than 50 years later, a similar situation exists with respect to finding the right programming language for HPC systems, with a yet uncertain outcome. And FORTRAN, as the traditional language for scientific programming, has been playing a major role as a base language for a multitude of approaches discussed over the past 20 years, as this paper intends to demonstrate. An important issue in this context is how to deal with *locality awareness* at a reasonable level of abstraction. This concept refers to features that allow the explicit analysis and control of locality in a programming language. A key characteristic of today's HPC systems is a physically distributed memory, which makes the efficient management of locality essential for taking advantage of the performance enhancements that these architectures can provide. This problem has existed since the mid 1980s, when distributed-memory HPC architectures emerged as a scalable alternative to vector and shared-memory architectures. The necessity for dealing with locality in these early machines, which provided no hardware support for a global memory, resulted from different protocols and huge differences in latencies and bandwidths between accesses to local and non-local data. A processor that needed to access a data item stored in another processor's memory had to *receive* this item from the other processor, which had to explicitly *send* it to the requesting processor. This required a careful synchronization between sender and receiver, complicated by the fact that in general each processor had to play both roles—as a sender and receiver—and that communication could proceed in multiple scopes in parallel, when library routines were used.

FORTRAN was an obvious starting point for most approaches targeting the design of a new programming paradigm addressing this problem. A first major community effort resulted in the *Message Passing Interface* (*MPI*) [25,42,53], which defined a standardized library for message passing, providing the programmer with a means for explicitly managing and synchronizing communication in a processor-centric model for programming and execution. While this approach allows virtually full control of communication – at a level of abstraction that can be compared to that of assembly language – it is commonly understood that it results in complex, brittle, and error-prone programs, due to the way in which algorithms and communication are inextricably interwoven.

It soon became clear that higher-level approaches to the management of locality, based on the concept of *data distributions*, were feasible. Data distributions provide an abstract specification of the partitioning of large-scale data collections across units of uniform memory access, supporting coarse-grain parallel computation and locality of access at a high level of abstraction. The discussion of such approaches – almost all of which were originally based on FORTRAN – is the focus of this paper. The key idea is the sharing of the responsibility for exploiting parallelism between the user and the compiler/runtime system: the user explicitly specifies data distributions and parallel execution of loops and other program sections, while the compiler and runtime system generate a parallel program with explicit communication, usually expressed using MPI. We will discuss the specification and use of data distributions, and their combination with features for the control of affinity between threads and the data they operate upon. The main challenge is to expose the user to enough level of detail regarding parallel execution to grant the effective transfer of problem-specific knowledge, while concealing the unproductive details related to low-level parallel programming, such as communication and the explicit distinction between local and remote memory accesses. And, what was recognized during the design of the original FORTRAN language, has retained validity: a high-level language supporting HPC programming can only be successful if the associated compiler and runtime systems technology is mature enough to deliver target programs that can compete with "hand-coded" programs using explicit message passing.

This paper is not intended to provide a general and complete treatment of programming languages for HPC systems. Rather, its focus is on the high-level manage-

ment of locality and languages that we call the "HPF family", which includes the main predecessors of HPF as well as successors up to modern object-oriented languages. Also, our focus is on the concepts; syntactic details are largely ignored.

The remainder of the paper is structured as follows. In Section 2, we establish a formal notation and terminology for data distribution, alignment, and affinity. This is followed in Section 3 by the specification of a set of standard distributions, which are provided by many HPC languages. In Section 4, we outline the early history of language efforts in this area and provide an overview of the High Performance Fortran (HPF) family of languages. The influence of FORTRAN-related developments on modern language design will be illustrated in Section 5, which describes the approach towards user-defined distributions in the language *Chapel*, which is currently under development in DARPA's *High Productivity Computing Systems* (*HPCS*) program. Chapel is a modern object-oriented language that takes up many of the key concepts of HPF, generalizing and extending them in significant ways. Section 6 presents an outlook to future research and concludes the paper.

2. Basic concepts

In this section we provide a conceptual framework for distributions, which will be used throughout the rest of this paper.

For the *abstract machine* underlying our model, a set of simple assumptions will suffice. It contains a *memory component* and a *processing component*. Each execution of a program on the abstract machine is associated with a region of its memory component, called the *execution locale set*, and an unbounded, dynamically managed set of *threads* in the processing component.

The **execution locale set** is a non-empty finite set of identical **locales** determined at the time program execution begins. Locales are units of uniform memory access to which data and threads can be mapped. Accesses of a thread to data are called **local** if thread and data are mapped to the same locale, else **remote**. In terms of performance metrics, a local access is assumed to incur less overhead than a remote access.

2.1. Domains

A **domain** is a first-class entity, whose principal aspects are:

- An *index set*, which is a finite set of elements for identifying components of arrays. Of special importance for scientific computation are the *arithmetic domains* related to Fortran 90 arrays, whose index sets are Cartesian products of sets of integer numbers that form an arithmetic sequence, with a *rank* – known at compile time – that specifies the number of dimensions. However, this model is more general, including among others sparse index sets and sets of class instances that reference nodes in an unstructured grid.
- A *distribution*, which specifies a global mapping from the domain's index set to locales in the execution locale set, as well as the local arrangement of indices and data within locales.
- A set of associated *arrays*, which are mappings from the domain's index set to component variables of a given type. Arrays are allocated according to the domain's distribution. Due to the generality of index sets and types in Chapel the notion of an array is a more powerful concept than its counterparts in traditional languages.
- *Iterators*, which are functions defined over the index set of a domain. They can be used to control the execution of sequential and parallel loops.

While every domain has a well-defined index set at any time of its existence, its other components are optional. If nothing else is said explicitly, we identify a domain with its index set in the following.

2.2. Index mappings

This section introduces a class of functions defined over finite, non-empty sets. We use them primarily for modeling mappings involving index sets of domains.

Definition: Let X, Y denote finite, non-empty sets.

1. An **index mapping from** X **to** Y is a total function $f : X \to \mathcal{P}(Y)$, where $\mathcal{P}(Y)$ denotes the powerset of Y.
2. f is called **proper** iff $f(x) \neq \phi$ for all $x \in X$.
3. For a proper index mapping, f, from X to Y, the set
$$f(X) := \{y \in Y \mid \exists x \in X : y \in f(x)\}$$
is called the **image of** X under f.
4. Given a proper index mapping, f, from X to Y, the **inverse mapping**, f^{-1}, is the inverse of the relation in $X \times Y$ defined by f: for all $y \in Y$: $f^{-1}(y) := \{x \in X \mid y \in f(x)\}$. If $f(X) = Y$, then f^{-1} is a proper index mapping from Y to X.

5. A proper index mapping is **replication-free** iff $\mid f(x) \mid = 1$ for all $x \in X$. Such a function will be interpreted as a total mapping, $f\colon X \rightarrow Y$, whenever this is more convenient. If f is surjective, then f^{-1} defines a *partition* of X in the mathematical sense, where all elements of X mapped to the same $y \in Y$ belong to the same equivalence class.

2.3. Data distribution, alignment, and affinity

A data distribution is defined for a domain, whose index set is being distributed, and a subset of the execution locale set, which serves as the target of the distribution. The core components of a distribution are two functions referred to as the *global mapping* and the *layout*, both of which are defined over the domain's index set. The global mapping is an index mapping from the domain's index set to the target locales, while the layout maps indices to locations of the locale associated with them via the global mapping, thus allowing the specification of locale-internal data arrangements.

Languages differ in the way how they treat these two kinds of mappings. Most languages present to the user a set of standard distributions with a pre-defined specification of the global mapping combined with a transparent layout defined by the implementation. An important exception is the Chapel language discussed in Section 5, where both mappings can be explicitly defined in a program.

The reader familiar with standard distributions in HPC languages will notice here that index mappings provide a more powerful mechanism than required when modeling, e.g., block or cyclic distributions: for such distributions, the global mapping function always associates *exactly one* locale with each index rather than an arbitrary non-empty set of locales. This is a valid observation; however, our formalism has been chosen such that it can also model replication, in the context of distributions as well as alignments, which requires the more general approach chosen here.

2.3.1. Global mapping

The global mapping of a domain to a set of target locales associates domain indices with locales. In the following definition we assume a domain *D* with index set *I*, which is distributed to a non-empty subset, LOC, of the execution locale set.

Definition:

1. The **global mapping**, δ, of a data distribution for D is a proper index mapping from I to LOC.
2. D is called the **source domain** of the distribution.
3. LOC is called the **target locale set**; its domain is called the **target domain**.
4. The inverse, δ^{-1}, of the global mapping is called the **ownership function**.
5. Given $loc \in LOC$, $\delta^{-1}(loc)$ is called the **distribution segment** of loc under the global map δ.

The distribution segment of a locale loc under the global map δ specifies the set of all indices in the source domain, I, which are mapped to loc via δ. The image, $\delta(I)$, of I under δ is called the **actual target locale set** for the distribution. This must be a non-empty set. However, δ is not constrained to be surjective, which means that $\delta(I)$ may be a proper subset of LOC. In that case there exist empty distribution segments.

Let A denote an array associated with domain D. Then the mapping $i \mapsto \delta(i)$ for all $i \in I$ determines the set of all locales on which the array component variable $A(i), i \in I$, is to be allocated. Each such locale is called a **home** for $A(i)$. For every loc in the actual target locale set, $\delta(I)$, the **local array segment** of loc for A, is the representation of the portion of A, $A(\delta^{-1}(loc))$, which is associated with the distribution segment of loc. The way in which data are stored in the local array segment is determined by the layout and the array's element type.

As has been mentioned above, there are many cases of interest, where the global mapping of a distribution is replication-free. However, there are instances where the mapping to a powerset is required. Examples include the replication of a "small" data structure (such as a scalar) to *all* locales: $\delta(i) = LOC$ for all $i \in I$, the replication of one or more dimensions of a multi-dimensional array in an alignment (Section 2.4), and the generation of copies of array elements in a halo (Section 4.3.2). In all these cases, the decision to replicate is motivated by the goal to reduce communication latency and bandwidth requirements, taking into account the penalty of increased main memory consumption.

2.3.2. Layout

The **layout** of a data distribution specifies the locale-internal representation of distribution segments and array data in the context of a global mapping. Related functions include methods for accessing data and iterating over index sets.

A discussion of the interface presented to the user for specifying layout will be found in Section 5. Here we introduce only one basic function, which we call the *layout mapping*:

Definition: Let D and I respectively represent a domain and its index set, as introduced above, and δ denote a global mapping. The **layout mapping** of a data distribution is an index mapping, λ, from I to the set of indices [2] in the locale determined by the global mapping.

For any array A and index $i \in I$, $\delta(i)$ and $\lambda(i)$ are the key components determining the location in which element $A(i)$ is stored.

2.4. Alignment

An *alignment* establishes a co-location constraint for corresponding elements in two arrays. This can be used for enforcing the locality of operations applied to multiple arguments.

Let the index sets I' and I be associated with two domains, respectively denoted as the *alignee* and the *source*. An **alignment**, α, from alignee to source, is a proper index set mapping from I' to I. Given α and a distribution, δ, from I to a LOC, a distribution, δ', from I' to LOC is determined as follows: for all $i' \in I'$:

$$\delta'(i') := \bigcup_{i \in \alpha(i')} \delta(i)$$

Alignment provides a capability for constructing a new distribution from an already existing one, in particular as a mechanism for enforcing locality. Important examples are *identity alignment*, where $\alpha(i') = \{i'\}$ for all i', and alignments where specific dimensions of the source index set are replicated or collapsed.

2.5. Data/thread affinity

Consider a distributed array, A, with index set I, distribution $\delta : I \rightarrow LOC$, and associated distribution segments $S_1, \ldots, S_n$. Now assume that a function, g, is to be applied to A, and that the effect of g can be achieved by applying it to all elements of A independently in parallel. If there are n threads $T_1, \ldots, T_n$ available to perform this work, then each thread $T_k, 1 \leqslant k \leqslant n$, can apply g to S_k, independently of the actions of all other threads. The association between the T_k and the S_k is said to establish **affinity** between threads and the data they operate upon.

[2] In this context, we speak of *locations*.

3. Standard distribution classes

In this section we apply the notation introduced above to the specification (under somewhat simplifying assumptions) of a set of standard distribution classes that are represented in many existing HPC languages.

3.1. One-dimensional distribution classes

The distributions described in this section are replication-free, mapping one-dimensional regular arithmetic data domains to one-dimensional regular locale domains. They include the block, block-cyclic, general block, and indirect distributions.

In the following, let the source domain be associated with a rank 1 regular arithmetic index set, I, which we specify in the form $I = (r : t : s)$, where r, t, and s are integers. Here, r denotes the first and t the last element in an arithmetic sequence with stride $s \neq 0$. If s is equal to 1, it can be omitted. For example, $(2 : 11 : 3)$ represents the sequence $2, 5, 8, 11$, (1:-5:-1) the sequence $1, 0, -1, -2, -3, -4, -5$, and $(3 : 100)$ stands for $3, 4, 5, \ldots, 99, 100$. The number of elements, or *extent* of the sequence specified by $I = (r : t : s)$ is given as $q = \lfloor \frac{t-r}{s} \rfloor + 1$, and the sequence itself is designated by the linearly ordered set of elements $\mathbf{set}(I) = \{u(0), \ldots, u(q-1)\}$.

Similarly, let the index set of the target domain be given as $I' = (r' : t' : s')$, a regular locale domain of rank 1 with extent $q' = \lfloor \frac{t'-r'}{s'} \rfloor + 1$. Then $\mathbf{set}(I') = \{u'(0), \ldots, u'(q'-1)\}$. The distributions discussed below will specify global index mappings from I to I'.

3.1.1. Block distributions

A *block distribution* partitions a one-dimensional arithmetic index set into contiguous blocks which are mapped to subsequent locales. The sizes of different blocks can differ by at most 1. [3]

The global mapping function, δ of a **block distribution** from I to I' is defined as follows. Let $l := \lfloor \frac{q}{q'} \rfloor$.

1. If $l = q/q'$, then the array is divided into q' blocks of size l, which are mapped to subsequent locales in I': $\delta(u(j)) = \{u'(\lfloor \frac{j}{l} \rfloor)\}$ for all $j, 0 \leqslant j \leqslant q-1$.

[3] Variants of this definition have been used in other languages [30, 54]. A version of block distribution used in HPF allows the explicit specification of block size, resulting in a mapping that may be non-surjective [30].

2. If $l < q/q'$ let $p = (l+1) * p'$: $q = l * q' + p' = (l+1) * p' + l * (q' - p')$, where $0 < p' < q'$. In this case, the source index set is divided into p' blocks of size $l+1$, which are mapped to the first p' elements of the locale index set, and $q' - p'$ blocks of size l, mapped to the remaining locales: The global mapping function of the distribution is given as
   - $\delta(u(j)) = \{\lfloor \frac{j}{l+1} \rfloor\}$ for all j such that $0 \leqslant j \leqslant p-1$
   - $\delta(u(j)) = \{\lfloor \frac{j-p}{l} + p' \rfloor\}$ for all j such that $q' \leqslant j \leqslant q'-1$.

Note that for block distributions, the application of the functions δ and δ^{-1} to their arguments can be determined by simple arithmetic involving q and q' only.

Example: Let $I = (0:31)$ and $I' = (0:3)$. A block distribution from I to I' results in block size $l = 8$ and

- $\delta(i) = \{\lfloor \frac{i}{8} \rfloor\}$ for all $i, 0 \leqslant i \leqslant 31$
- $\delta^{-1}(i') = (8 * i' : 8 * i' + 7)$ for all $i', 0 \leqslant i' \leqslant 3$

3.1.2. Block cyclic distributions

A *block cyclic distribution* generates a round-robin mapping of contiguous blocks of indices in I to subsequent locales in I'.

A **block cyclic distribution** from I to I' with **block length** $k \geqslant 1$ is defined by the global mapping function $\delta(u(j)) := \{MOD(j/k, q')\}$ for all $j, 0 \leqslant j \leqslant q-1$.

For block cyclic distributions, the application of the functions δ and δ^{-1} to their arguments can be determined by simple arithmetic involving k, q and q' only.

Block cyclic distributions with block length $k = 1$ are simply called *cyclic*.

3.1.3. General block distributions

General block distributions provide more flexibility than block distributions at moderate cost, by allowing different blocks to have different block sizes specified by the program. All other properties, in particular the contiguity of the index subdomain associated with a distribution segment, and the mapping of subsequent segments to subsequent locale indices, remain the same as with block distributions.

Let $\mathbf{b} = (b_0, \ldots, b_{q'-1})$ denote a strictly monotonic sequence of indices in I, where each b_p denotes the first index of I to be mapped to $I'(p), 0 \leqslant p \leqslant q'-1$, and $b_0 = u(0)$. A **general block distribution** with begin vector **b**, *gen_block*(**b**), is defined by this global mapping function:

- $\delta^{-1}(p) = (b_p : b_{p+1} - 1)$ for $0 \leqslant p < q' - 1$, and $\delta^{-1}(q'-1) = (b_{q'-1} : q')$
- $\delta(u(j)) := \{p\}$, for all $j, 0 \leqslant j \leqslant q-1$, where p, $0 \leqslant p \leqslant q'-1$, is the smallest number such that $b_p \leqslant u(j)$.

The representation of general block distributions essentially consists of the begin vector, requiring $\mathbf{O}(q')$ memory; access to an element needs time $\mathbf{O}(log_2(q'))$.

Figure 1 gives an example for a general block distributed one-dimensional arithmetic index set $I = (1..12)$, with a locale index set $I' = (1:4)$: The associated distribution segments are given as $\delta^{-1}(1) = (1:5)$, $\delta^{-1}(2) = (6:7)$, $\delta^{-1}(3) = (8:10)$, and $\delta^{-1}(4) = (11:12)$. The begin vector is given as $(1, 6, 8, 11)$.

3.1.4. Indirect distributions

Indirect distributions allow the direct specification of an arbitrary total mapping from an arithmetic index domain to a locale domain using an **indirect-mapping function**.

Let $f : (0 : q-1) \rightarrow (0 : q'-1)$, total, denote an **indirect-mapping function**. The global mapping function associated with an **indirect distribution** is given as $\delta(u(j)) := \{f(u(j))\}$ for all j.

The class of indirect distributions represents the most general replication-free one-dimensional distributions. In most cases where indirect distributions are used in practice, the mapping function is determined at runtime by a partitioning routine applied to an unstructured grid. Subsequently, the array is partitioned based on the mapping function. The implementation cost of indirect distributions can be considerable since the representation of the mapping function may in general require $\mathbf{O}(q)$ memory; furthermore the generally large size of q may result in the need to distribute the representation as well.

3.2. Distributions for multi-dimensional index sets

Distributions from multi-dimensional source domains to multi-dimensional target domains can be classified as either **dimensional** – where each dimension of the source domain is mapped to exactly one dimension of the target domain (or not mapped at all), or **non-dimensional** – which allow an arbitrary mapping from source to target indices, irrespective of dimensions. We discuss only dimensional distributions here.

Let the index set of the source domain be given as a regular arithmetic n-dimensional Cartesian product of

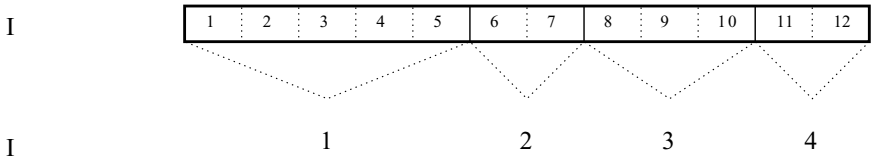


Fig. 1. A general block distribution.

sections, $I = I_1 \times \ldots \times I_n$, with $n \geqslant 1$. Similarly, we assume the index set of the target domain to be an n'-dimensional Cartesian product of sections, $I' = I'_1 \times \ldots \times I'_{n'}$, $n' \geqslant 1$.

A distribution of I to I' is called a **dimensional distribution of rank** m iff the global mapping function, δ, satisfies the following conditions:

1. Each dimension of I is characterized with respect to δ as either **distributed** or **non-distributed**. Let $(d_1, \ldots, d_m)$ denote the **sequence of distributed dimensions** in ascending order, where $0 \leqslant m \leqslant n$.
2. $m \leqslant n'$
3. The distributed source dimensions, $I_{d_1}, \ldots, I_{d_m}$, are one-to-one mapped to the target dimensions $I'_1, \ldots, I'_m$, forming one-dimensional source/ target pairs of index sets. For each such pair, a one-dimensional distribution with global mapping function δ_{d_k} from I_{d_k} to target I'_k must be provided.
4. For each $i = (i_1, \ldots, i_n) \in I$:

$$\delta(u_1(j_1), \ldots, u_n(j_n)) = \{(i'_1, \ldots, i'_{n'}) \in I' \mid \forall k, 1 \leqslant k \leqslant m : i'_k \in \delta_{d_k}(u_{d_k}(j_k))\}$$

Note that for $m < n'$ the $n' - m$ target dimensions $m+1, \ldots, n'$ are universally quantified.

There are two important special cases for dimensional distributions, which are respectively characterized by $m = n$ and $m = 0$. In the first case, all source dimensions are distributed, and the sequence $(d_1, \ldots, d_{n'})$ is identical with $(1, \ldots, n)$. In the second case, no source dimension is distributed. Then $\delta(u_1(j_1), \ldots, u_n(j_n)) = I'$ for all indices of the source domain: I is totally replicated in this case.

4. The High Performance Fortran family of languages

4.1. HPF predecessors

Early work in the field included IVTRAN [43], a language developed for the SIMD machine ILLIAC IV, Connection Machine Fortran [1], Kali [39], and the SUPERB (*Suprenum Parallelizer Bonn*) system [58]. SUPERB was an interactive restructuring tool, developed at the University of Bonn, which translated Fortran 77 programs into message-passing Fortran for a set of early distributed-memory architectures including the SUPRENUM machine [24]. The system supported general block distributions, which were specified via a simple interactive language.

The immediate predecessors of HPF were Fortran D, developed at Rice and Syracuse universities, and Vienna Fortran, developed at the University of Vienna in cooperation with ICASE, NASA Langley Research Center. Both languages were originally based on FORTRAN 77.

Fortran D [23,32] adopted the following approach: it first aligned data arrays to virtual arrays known as *decompositions*; decompositions were then distributed across an implicit set of processors using relative weights for the different dimensions. The language allowed an extensive set of alignments along with block, cyclic, and indirect distributions. All mapping statements were considered executable.

Vienna Fortran [13,59] was the first language to provide a complete specification of mapping constructs in the context of Fortran. Based to a significant extent on Kali, it provided the programmer with a facility to define arrays of virtual processors, and introduced distributions as mappings from multi-dimensional array index spaces to (sub)sets of processor spaces. Special emphasis was placed on support for irregular and adaptive programs: in addition to the regular *block* and *block-cyclic* distribution classes the language provided *general block* and *indirect* distributions. It is important to note that general block distributions, when used in conjunction with reordering of data sets, can efficiently represent partitioned irregular meshes, without the need to use indirect distributions – which are more expensive to implement – for this purpose.

The Vienna Fortran Compilation System extended the functionality of the SUPERB compiler to cover most of the language, while placing strong emphasis on the optimization of irregular algorithms. An overview

of the compilation technology used in this system is presented in [6].

Several other academic as well as commercial projects also contributed to the understanding necessary for the development of data parallel languages and the required compilation technology [3,10,26,27,34,38,41, 44,45,47,49–51,55]. More recently, High Performance Java [31] and a number of parallel Matlab versions [36] have extended their base languages with high-level distribution specifications. These languages largely rely on a set of built-in mechanisms, such as the standard distributions described above in Section 3. A major step for defining more general distribution classes was proposed in the Vienna Fortran language specification [59], which introduced a capability for user-defined mappings from Fortran arrays to a set of abstract processors, and for user-defined alignments between arrays. Distribution classes more general than the standard distributions include the Kelp library [21] and the generalized multipartitioning scheme implemented in Rice University's dHPF compiler [16].

4.2. High Performance Fortran

The definition of High Performance Fortran (HPF) was a community effort: the HPF Forum, a group of about 40 researchers, met at intervals of about six weeks between March 1992 and May 1993, at which time Version 1.0 of HPF was released. After the completion of a corrected and improved HPF Version 1.1 in November 1994, the HPF Forum resumed regular meetings in 1995 and 1996, resulting in a revised and extended version of the language, HPF 2.0, in 1997.

HPF is a set of Fortran extensions designed to allow specification of data parallel algorithms for a wide range of architectures. The user annotates the program with distribution and alignment directives to specify the desired distribution of data. The underlying programming model provides a global name space and a single thread of control. Explicitly parallel constructs allow the expression of fairly controlled forms of parallelism, in particular data parallelism. Thus, the code is specified with no explicit tasking or communication statements. The goal is to allow architecture-specific compilers to generate efficient code, with a clear focus on the support of locality.

The HPF 2.0 language consists of three parts: a) the Base Language, b) the Approved Extensions, and c) Recognized Extrinsic Interfaces. The base language defines the basic HPF features which each HPF compiler must support. The Approved Extensions include advanced features that meet specific needs but are not required to be supported by HPF compilers. The Recognized Extrinsic Interfaces provide interfaces to other programming paradigms and languages.

Below we provide a brief description of the base language and the approved extensions, respectively. A more complete description of the language can be found in the HPF Language specification [29].

4.2.1. The base language

The HPF 2.0 Base Language supports a set of features for the specification of global mappings based on predefined distribution classes, and the expression of parallelism. This language is very close to the original version, HPF 1.0, of HPF.

Global mapping directives. HPF provides directives to specify the mapping of array elements to memory regions associated with "abstract processors". Arrays are first aligned relative to each other and then the aligned group of arrays are distributed onto a rectilinear arrangement of abstract processors. The alignment directives support a rich set of mappings, including identity alignment, alignment with offset and stride, collapsing, embedding, replication and permutation. The distribution directives allow each dimension of an array to be independently distributed using the block and block cyclic distributions.

Data parallel directives. HPF2.0 is based on the Fortran 95 standard. Thus, the array constructs of Fortran 90 can be used to specify the data parallelism in the code. Also, the forall statement and construct (which were introduced in HPF version 1.1 and later adopted in Fortran 95) provide a more general mechanism to specify such parallelism.

HPF contains the **independent** directive to assert that iterations of a loop do not have any loop-carried dependences [57] and thus can be executed in parallel. A **reduction** clause can be used with this directive to identify variables which are updated by different iterations via associative and commutative operators.

Intrinsic and library functions. HPF provides a rich set of new intrinsic functions including inquiry functions referring to the underlying hardware and the mapping of the data structures, and a set of computational intrinsic functions. New library routines were defined so as to provide a standard interface for highly useful parallel operations such as reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

Extrinsic procedures. In order to accommodate other programming paradigms, HPF provides *extrinsic* pro-

cedures. These define an explicit interface and allow codes expressed using a different language, e.g., C, or a different paradigm, such as an explicit message passing, to be called from an HPF program.

4.2.2. HPF Approved Extensions

The Base Language is somewhat restricted, in particular with respect to the functionality provided by its data distributions. Only block and block cyclic are supported; and distributions always target the full set of locales associated with an abstract processor array. Furthermore, there exists no mechanism in the language that supports the specification of affinity. These limitations can result in performance problems, in particular if the language is used to model unstructured meshes. A detailed analysis of these problems led to the HPF 2.0 Approved Extensions as discussed below.

Extensions to global mapping directives. These extensions allow greater control of the mapping of data objects. For example, users can map pointers and components of derived types, and can map objects to subsets of processors directly. New distribution formats include general block and indirect for the support of irregular meshes.

Another important feature is the support for dynamic remapping of data. If an object has been declared **dynamic** then it can be remapped at runtime using the the **realign** or **redistribute** directives.

Extensions to data parallel directives. In addition to mapping data, the **on** directive allows users to map computation onto processors thus allowing the explicit specification of affinity. The **resident** directive allows the specification of information about accesses to data objects within the scope of an associated **on** block.

The **task_region** directive extends HPF beyond the realm of data parallelism by allowing some forms of control parallelism to be expressed within the language. This directive can be used to indicate regions of code that can be executed in parallel on different subsets of processors.

4.2.3. A simple example of an HPF code

The HPF version of the Jacobi iterative procedure which may be used to approximate the solution of a partial differential equation discretized on a grid, is shown in Fig. 2. Such an algorithm, using a five-point stencil, is typical of many CFD applications.

In this code fragment, the data objects are distributed as follows. The array F is aligned with the array U using identity alignment. U is declared to distributed via the distribution clause (*****, **BLOCK**), implying that the first dimension of the array is not distributed, wheras its second dimension is block distributed. That is, the columns of U (and thus those of F because of the alignment) are distributed by block across the processor array P. P has been declared to be an array of abstract processors whose size is determined by the system inquiry function *NUMBER_OF_PROCESSORS*, which returns the size of the execution locale set. The above code can be run on varying numbers of processors without recompilation. The computation is expressed using a **FORALL** construct, where all the right hand sides are evaluated using the old values of U before assignment to the left hand side.

Note that the computation is specified using a global index space and does not contain any explicit data motion constructs such as explicit communication statements. Assume now that the **FORALL** loop is stripmined by the compiler using the *owner computes rule*, where the owner of the data object at the left-hand side of an assignment executes the assignment. Since the underlying arrays are distributed by columns, the edge columns will have to be communicated to neighboring processors. It is the compiler's responsibility to analyze the code and translate it into an explicitly parallel code with the appropriate communication statements inserted to satisfy the data requirements.

A detailed discussion of the application of HPF to advanced applications including multiblock codes and the processing of unstructured grids is provided in [40]. The history of HPF, and its relationship to the evolution of architectures, applications, and other language developments are discussed in [35].

4.3. HPF+ and HPF/JA

While HPF 2.0 represents considerable progress on the path to an expressive HPC language, there are a number of situations, where an equivalent hand-coded program using MPI may display superior performance. For example, the functionality of HPF does not provide for the detailed control of communication that is needed in a finite element code simulating vehicle crashes [28].

HPF+ [5,7,8], developed and implemented in a European project, addresses these problems by providing features that allow explicit control of communication at a high level of abstraction. HPF+, resulting from an analysis of advanced industrial codes, provides a `REUSE` clause for independent loops that asserts reusability of the communication schedule computed during the first execution of the loop. Furthermore, the `halo` construct allows the functional specification of

```
!HPF$ PROCESSORS P(NUMBER OF PROCESSORS())
      REAL U(1:N,1:N), F(1:N,1:N)
!HPF$ ALIGN U::F
!HPF$ DISTRIBUTE U(*,BLOCK)
      FORALL (I=2:N-1,J=2:N-1)
      U(I,J) = 0.25 * (F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J-1)+U(I,J+1))
      END FORALL
```

Fig. 2. HPF version of a Jacobi procedure.

non-local data accesses in processors, and user control of the copying of such data to region boundaries.

In 1999, the Japan Association for High Performance Fortran, a consortium of Japanese companies including Fujitsu, Hitachi, and NEC, released HPF/JA, which included a number of features found in previous programming languages on parallel-vector machines from Hitachi and NEC. An important source contributing to HPF/JA was the HPF+ language. These and other features allowed for better control over locality in HPF/JA programs. For example, the `LOCAL` directive could be used to specify that communication was not needed for data access, and the `REFLECT` directive included in HPF/JA corresponds to HPF+'s `halo` feature. HPF/JA was implemented on the Japanese Earth Simulator [52].

In a sense, the features described above violate one of the principal goals stated at the beginning of this paper – to avoid explicit management of communication in a program. They allow the experienced user to provide information to the program that the compiler or the runtime system are not able to derive automatically. But they are unsafe: a wrong assertion by the programmer in using these features will conceivably result in a program crash. As a consequence, they were expected to be used only in extremely performance-critical sections of the program.

Below we discuss the explicit management of communication schedules and halos in more detail.

4.3.1. Explicit management of communication schedules

Parallel loops that occur in the context of algorithms dealing with semi-structured or unstructured grid computations contain indirect accesses to arrays. This is due to the fact that such grids are generated dynamically, and the neighborhood of objects is not reflected by a similar neighborhood of their indices, as is the case in dense regular computations such as in the Jacobi example discussed above. A typical loop occurring in such a computation may look like the excerpt of a sweep over the edges of an unstructured grid [40], as sketched in Fig. 3.

Data accesses in such a loop cannot in general be analyzed by the compiler. However, the fact that the loop is asserted as being *independent* excludes race conditions and permits runtime optimization, which under certain conditions can dramatically improve performance by exploiting temporal locality. Many of these methods are related to the *inspector/executor* paradigm [37]. The essential ideas are the following:

1. Before beginning the actual execution of the loop, runtime analysis of its access patterns and the associated communication is performed. This is called the *inspector*. The result of runtime analysis is stored in a *communication schedule* for each array accessed in the loop.
2. The loop is restructured in such a way that:
   (a) all read communication is performed before the actual loop execution
   (b) all write communication is performed after completing the loop execution
3. The core execution of the loop, after the transformation of Step 2, is purely local.

The PARTI and CHAOS libraries [46] provide a set of primitives that support the inspector/executor paradigm. They have been integrated into a number of compilers, including the Vienna Fortran Compilation System(VFCS) [6] and the Fortran D system [32].

The inspector phase may be very expensive and represents a significant runtime overhead, which in some cases may dominate the overall execution time [9,46]. It is therefore important to be able to deal with situations in which the schedule computed by the inspector remains invariant over multiple executions of the parallel loop (for example, if the parallel loop is enclosed in a time-step loop). If this is the case, the inspector needs to be executed only when the loop is entered for the first time, suppressing subsequent computations of the communication schedule. Three approaches to this problem have been studied:

```
!HPF$ INDEPENDENT, ON HOME(EDGE(J, 1)), NEW(N1, N2, DELTAX), REDUCTION(V2)
      DO J = 1, M
       ...
       N1 = EDGE(J, 1);     N2 = EDGE(J, 2)
       ...
       DELTAV = F(V1(N1), V1(N2))
       V2(N1) = V2(N1) - DELTAV
       V2(N2) = V2(N2) + DELTAV
      ENDDO
```

Fig. 3. Sweep over edges of an unstructured grid.

– *Inspector hoisting* [17] determines at compile time the set of program statements and data on which the inspector depends. If the compiler can assert invariance, it can modify the code generated for the inspector/executor paradigm to suppress redundant computations of the communication schedule.
  This is difficult for many "real" programs, in which calls to parallel loops tend to occur in a deep nest of procedure calls which may be impossible to analyze statically.
– *Timestamping* denotes a method in which assignments to indirection arrays are monitored at runtime. If, at the time the inspector is activated, the communication schedule has already been computed in a previous iteration and none of the indirection arrays on which the access pattern depends has changed in the meantime, then a recomputation of the communication schedule can be avoided. Experiments with this method have been reported in [46].
– *Explicit schedule control* gives the programmer linguistic mechanisms to explicitly control the computation and application of communication schedules. These language extensions, described in detail in [9], have been included in HPF+ and implemented in VFCS. An example is discussed below.

The code fragment below illustrates the use of *schedule variables* to control the computation of a communication schedule.

```
!HPF+ SCHEDULE:: S
    ...
    DO t = 1, max_time
    ...
!HPF+ INDEPENDENT, ON HOME(C(I)), GATHER (B::S)
     L: DO I = 1, N
          ...
          A(I) = B(IX(I)) + C(I)
          ...
        END DO
          ...
          IF RECONFIGURE(...) THEN CALL
          RECOMPUTE(IX)
!HPF+      RESET S
          END IF
        END DO
```

Here, S is an explicitly declared schedule variable which is initially undefined. Therefore, the first time the parallel loop is encountered, the inspector is fully executed and determines a communication schedule as discussed above. This schedule is assigned to S. Upon subsequent executions of the parallel loop, this schedule is reused, as long as the call to *RECONFIGURE* yields **false**. Once this call yields **true**, S is *reset*, resulting in the disassociation of S from the previously computed schedule, and setting its value to *undefined*. With the next execution of the parallel loop, the inspector has to be run again, and the above cycle is restarted.

4.3.2. Halos

Halos for a given distributed data structure are designed to support the user-directed optimization of communication for that data structure in a region of the program by allowing:

1. the explicit allocation of copies of non-local elements of the data structure in a locale,
2. the explicit management of coherency between such copies and their home, and
3. the replacement of all non-local references to the data structure by local references.

Consider a program region R – typically the body of a function – in which an array A is accessed. A **halo** for A is a user-provided specification of the set (or a superset) of components of A that are remotely accessed from threads executing R. For each component in a halo a local copy is generated in the accessing locale. The user can control communication between local copies and their home via explicit high-level commands, such

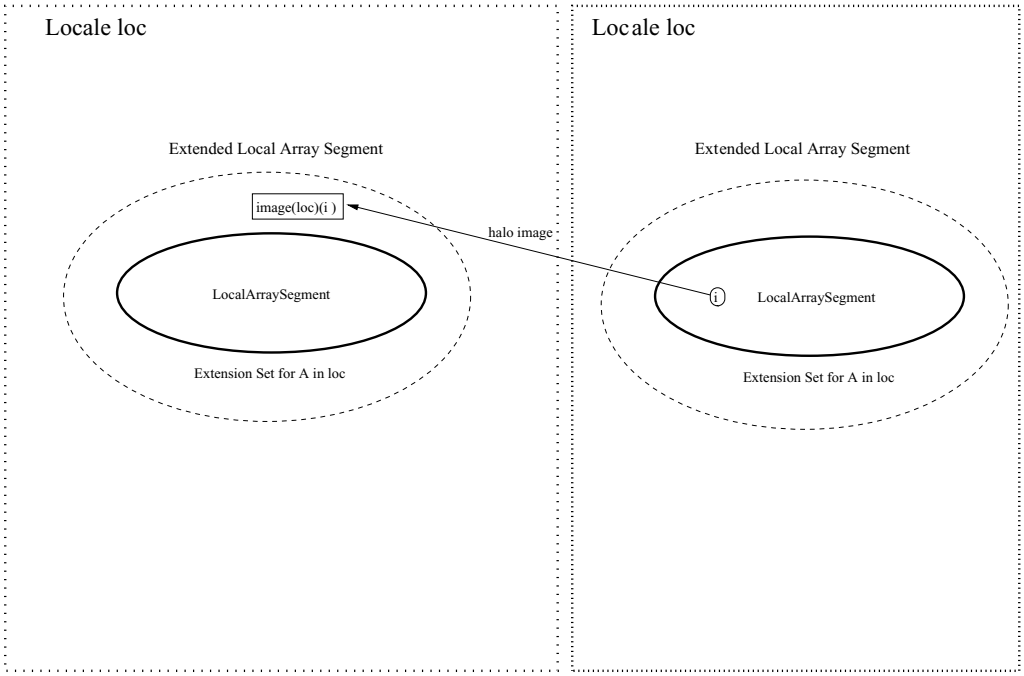


Fig. 4. Halo relationships: $A(i')$, owned by locale loc', is in the halo of loc, and has image $image(loc)(i')$, which extends the local array segment for A in LOC.

as *update_halo*(. . .) and *flush_halo*(. . .). Implicit in the use of halos is the assertion that *all* communication with regard to A in *R* is controlled explicitly by the user, i.e., the standard method of generating communication in the compiler/runtime system is suppressed when halos are used and all accesses to A are interpreted as local.

Key relationships regarding halos are outlined in Fig. 4.

5. Chapel

The Chapel programming language [11,12,48] is being developed in the Cascade project of the DARPA-sponsored High Productivity Computing Systems (HPCS) program. It is a modern object-oriented language designed to promote the development of efficient and reusable parallel code. Key features of Chapel include support for generic programming, type parameterization, collections and iterators, and explicit parallelism in conjunction with user-defined data distributions.

The Chapel programmer reasons about data distribution, affinity and their relationships to data access patterns in an algorithm from a global viewpoint. In this approach, there is no concept of built-in distributions as with all languages discussed so far, but of a distribution class interface which allows the explicit specification of global mapping, layout, and affinity. The result is a concise programming model that separates algorithms from data representation and enables reuse of distributions, allocation policies, and data structures.

From a programmer's perspective, the global mapping is the primary component of a distribution and must be explicitly specified any time a distribution is defined. In contrast, the specification of the layout may be left to the system, which provides a default layout for any global mapping. Explicit specification of the layout is required if highly sophisticated data representation strategies are to be applied, which are beyond the reach of automatic methods, such as for distributed sparse data structures. In such a case, the layout specification allows virtually complete control over the locale-internal allocation policy. This capability can provide a significant performance gain in situations, where there is a strong correlation between data access patterns and internal data structures reflecting properties of an application that cannot be automatically recognized by the compiler.

The approach taken in Chapel towards user-defined data distributions has been strongly influenced by Vienna Fortran and HPF. It is important to note that it

```
class Domain {
  iterator for (): IndexType;                          /* sequential iterator */
  iterator forall (): IndexType;                       /* parallel iterator */
  function GetDistribution(): Distribution;            /* distribution of the domain */
  function GetParent(): Domain;                        /* parent domain */
  function extent(): integer;                          /* size of the index set */
}

class Distribution {
  var source: Domain;                          /* the source domain to be distributed */
  var target: Domain;                          /* the target locale domain */

  function getSource(): Domain;
  function getTargetDomain(): Domain;
  function getTargetLocales(): [target] locale;
  function map(i: index(source)): locale;
  iterator DistSegIterator(loc: index(target)): index(source));
  function GetDistributionSegment(loc: index(target)): Domain;
}

class LocalSegment: Domain {
  var LocalDomain: Domain;              /* local data domain */

  function getLocale(): locale;         /* locale associated with an instance of this class */
  function layout(i: index(source)): index(LocalDomain);
  function setLocalDomain(ld: Domain);
  function getLocalDomain();
}
```

Fig. 5. Base classes of the distribution framework: Public components.

is largely independent of the base language chosen and could be easily formulated based on other existing object-oriented languages.

5.1. *User-defined specification of distributions in Chapel*

The Chapel programmer does not necessarily need insight into the details of the specification and the inner workings of a distribution in order to *use* it successfully in an application. A predefined distribution class can be used by just attaching an instance to a domain or array declaration, as long as the construct is syntactically valid and certain semantic constraints are satisfied. This applies to any distribution, may it be standard – such as *block*, *block-cyclic* or *indirect* – or tuned to a class of applications exploiting their special properties. Apart from the knowledge of the interface the programmer only needs guidance for the effective use of the distribution in view of the array declarations in the program and the related access patterns in algorithms.

However, the sophisticated user will have the opportunity to explicitly define data distributions based on an object-oriented framework provided by the language. There exist no built-in distributions as a part of the language (and therefore known to the compiler) but any distribution that is required in a program must be explicitly defined and – if to be reused in different contexts—stored in a library.

In this section we introduce the interface and the methodology for the specification of user-defined distributions. In a sense, these features can be considered to be at a lower level of abstraction than the rest of the Chapel language since their implementation interacts directly with architectural features that are at a lower level than the abstractions addressed in the Chapel source language. Specifically, the locality properties of a program execution and the required communication depend on the access patterns to arrays and the distributions of their domains. We can think of specialized *distribution writers* (which of course can be the application developers themselves) being in charge of developing sophisticated distribution libraries that reflect properties of applications.

We begin by explaining the interface of the distribution framework in Section 5.1.1. We will see that distributions can be essentially specified at two levels, one of which deals only with the global mapping while the other uses in addition an explicit layout to control in detail the arrangement of data at the locale-internal level. In the first case (Section 5.1.2), the system provides by default all functionality required for the allocation and management of distributions and distributed arrays based on the user-specified global mapping

```
class MyC1: Distribution {
  const ntl: integer=… ;     /* number of target locales */
  function map(i:index(source): locale {return Locales(mod(i-1,ntl)+1);}

  iterator DistSegIterator(loc: index(target)): index(source) {
     const N: integer = getSource().extent;
     const k: integer = locale index(loc);

     for i in k..N by ntl { yield (i); }
  }

  function GetDistributionSegment(loc: index(target)): Domain {
     const N: integer = getSource().extent;
     const k: integer = locale index(loc);

     return (k..N by ntl)
  }
}

const D1C1: domain (1) distributed (MyC1()) on Locales(1..4) = 1..16;

var A1: [D1C1] float;
```

Fig. 6. A cyclic distribution.

(and, in most cases, the specification of its inverse). We illustrate below simple examples for this option. The explicit use of a layout specification will be explained subsequently in Section 5.1.3. by a highly specialized *banded distribution*. We also discuss how it can be applied to a matrix vector product using a distributed sparse matrix represented by a combination of a binary recursive distribution of the dense parent domain with a compressed row storage representation of the locale-internal data structures of the sparse domain.

5.1.1. Main interface classes

The distribution framework provides the distribution writer with tools for supplying application-specific functionality to the compiler and runtime system via a set of predefined public base classes with an overridable interface.

The base classes involved include `Domain`, `Distribution`, and `LocalSegment`, which are shown in Fig. 5 together with public methods. We provide here only a reduced list focusing on the main functionality required in this section, and omitting details that will be used and explained in examples. For example, an arithmetic domain provides methods that determine the extent of its index set, and for each of its dimensions the lower and upper bounds.

The `Domain` class supports the built-in features for domains in the language. Two general iterators are associated with each domain: the sequential `for` and the parallel `forall` iterator. The public method `GetDistribution` provides access to the domain's distribution, while `GetParent` links a subdomain to its parent. A call of the method `extent` yields the number of elements in the domain's index set.

The core component of the framework is the `Distribution` base class. Any user-defined distribution class is a subclass of `Distribution`. The `getSource` method yields the source domain of the distribution, while the `getTargetDomain` and `getTargetLocales` methods respectively yield the target domain and the target locale set.

The `map` method represents the global mapping from source indices to target locales.[4]

The iterator `DistSegIterator(loc)` produces the elements in the distribution segment associated with locale `loc`, as a sequence of source domain indices.

Finally, the function `GetDistribution Segment`, applied to a locale `loc`, yields the domain associated with the distribution segment of `loc`.

The global mapping is the only method that must be always specified by the distribution writer when specifying a new distribution. In addition, the user will in general provide explicit specifications of `DistSegIterator` and `GetDistribution-Segment`, although the system, in principle, can de-

[4] The generic type constructor `index`, when applied to a domain, yields its index type.

termine these functions by automatically inverting the `map` function. In most cases, the specification of these functions by a knowledgable user will significantly enhance performance.

The `LocalSegment` class, a subclass of `Domain`, is a predefined class that provides a default representation of distributions and associated array data in locales. A separate instance of this class is created on every locale in the target locale set of a distribution. This class can be overriden if the user wants explicit control of either mechanism.

The function `getLocale` yields the locale for a specific instance of `LocalSegment`. Let `loc` denote such a locale: we discuss the properties of the particular instance of `LocalSegment` for this locale:

The value of the public variable `LocalDomain` is the domain for the representation of local array data in locale `loc`. At this point we postulate that each array associated with the source domain of the distribution is represented in locale `loc` by a separate array over the domain `LocalDomain`. In order to deal with complex data structures such as sparse matrices the user may need to generate a set of persistent auxiliary data structures in each locale to support an efficient representation of the distribution and the mapping from global to locale-internal indices.

Finally, the `layout` function maps a global source index (that in the given context can be assumed to be in the distribution segment associated with `loc`) to the associated index in `LocalDomain`.

5.1.2. User-specified global mapping

We discuss here two examples for user-defined distributions, which deal with the global mapping and leave the arrangement of locale-internal data structures as well as the definition of the layout mapping to the system. The minimum specification for practical purposes consists of the definition of a subclass of `Distribution`, combined with a definition of the methods `map`, `DistSegIterator`, and `GetDistributionSegment`.

Specification of a Cyclic Distribution. The first example considered here is a variant of the cyclic distribution. The source code is given in Fig. 6. For simplicity we consider here only the case where the block length is 1, and source as well as target domains are arithmetic index sets of rank 1, defining a contiguous interval of integer indices beginning with 1.

The `DistSegIterator` in this specification first determines the number of elements in the source domain. According to our conventions, the source domain is then given as `1`...`N`. Secondly, it determines `k` as the number of the locale to which `DistSegIterator` is applied. The iterator successively generates the arithmetic sequence of all values of the form $k + j * ntl$, where $j = 0, 1, \ldots$ and $k + j * ntl \leqslant N$.

The `GetDistributionSegment` function is similar, except that it produces as its value a domain of rank 1, whose index set is determined by the elements in the arithmetic sequence discussed above.

For the distribution of `D1C1` we obtain:

$$\delta(i) = \mathrm{mod}\ (i-1, 4) + 1 \text{ for all } i, 1 \leqslant i \leqslant 16$$

The distribution segments can be specified as

$$\delta^{-1}(k) = k \ldots 16 \textbf{ by } 4 \textbf{ for } 1 \leqslant k \leqslant 4$$

Specifically,

$$\delta^{-1}(1) = \{1, 5, 9, 13\}$$
$$\delta^{-1}(2) = \{2, 6, 10, 14\}$$
$$\delta^{-1}(3) = \{3, 7, 11, 15\}$$
$$\delta^{-1}(4) = \{4, 8, 12, 16\}$$

5.1.3. User-specified layout

In this section, we illustrate the actions required for user-specified layout through a detailed example describing an irregular banded distribution for a square matrix. The specification determines specialized structures for auxiliary data and the actual array storage in each target locale. We also briefly discuss issues with the specification of a distributed sparse problem.

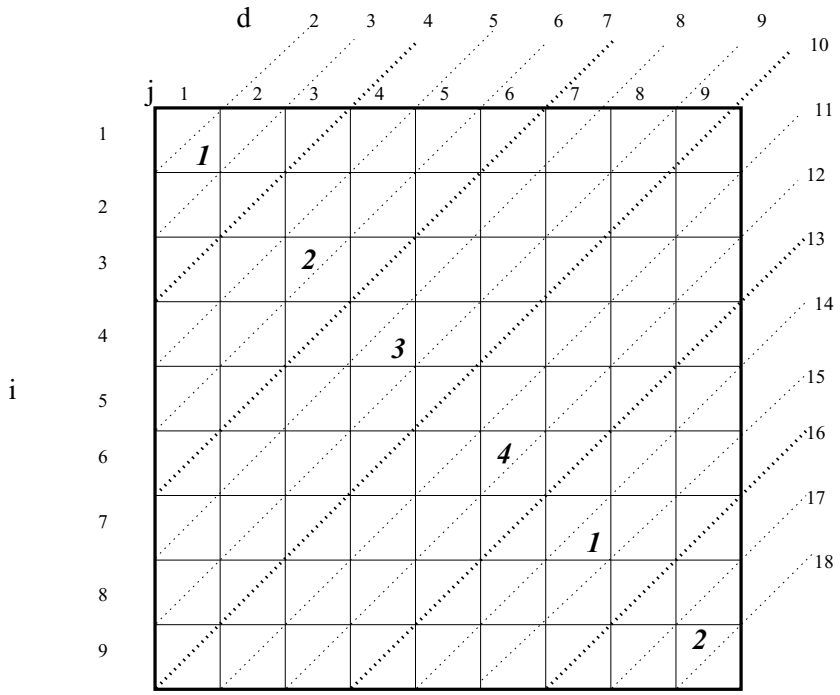
Example: Irregular banded distribution

In this example[5] we study a *banded distribution* for a square matrix of order n. The problem is specified as follows: Let A denote an array associated with arithmetic domain, `D`, of rank 2 with index set [1:n, 1:n]. For the purpose of this example we define for all $d = 2, 3, \ldots, 2 * n$ the *diagonal* $DIAG^d$ as the sequence of all indices (i, j) of D such that $d = i + j$, starting either with an element in the top row or the rightmost column, and proceeding diagonally from top down and from right to left.

In the example of Fig. 7, where $n = 9$, the diagonals are identified as $DIAG^2$, $DIAG^3$, $\ldots$, $DIAG^{18}$, where,e.g.:[6]

[5] This problem has been proposed by Brad Chamberlain from Cray Inc.

[6] A sequence of elements $x_1, \ldots, x_m$ is denoted in Chapel in the form $(/x_1, \ldots, x_m/)$.



Figures inside the matrix denote locale numbers

Fig. 7. A banded distribution.

$DIAG^2 = (/(1,1)/)$
$DIAG^5 = (/(1,4),\ (2,3),\ (3,2),\ (4,1)/)$, and
$DIAG^{16} = /((7,9),\ (8,8),\ (9,7)/)$.

Now assume $b \geqslant 1$ to denote a *bandwidth*, and p the number of target locales in the distribution to be constructed. This distribution is characterized as follows:

1. Starting with $d = 2$, diagonals $DIAG^d$, $DIAG^{d+1}$, ..., $DIAG^{d+b-1}$ are mapped to `Locales(1)`. Similarly, the next b diagonals are mapped to `Locales(2)`, and so on, in a cyclic pattern: after the first $b * p$ diagonals have been mapped to locales `Locales(1)` through `Locales(p)`, the process continues for diagonal $b * p + 1$ with a mapping to `Locales(1)`, etc.
   This results in the following global mapping for index $(i, j) \in$ [1:n, 1:n]:

   $$\delta(i,j) = ((i + j - 2)/b + 1) \bmod p$$

   In the matrix of Fig. 7, we assume $b = 3$ and $p = 4$. Then $\delta(7,6) = 4$ (this element belongs to $DIAG^{13}$), and $\delta(8,9) = 2$. For every $k \in 1 : p$, the distribution segments are determined as:

   $$\delta^{-1}(k) = \{(i,j) \mid (t * p + k - 1) * b + 2 \leqslant i + j \leqslant \min(2 * n, (t * p + k) * b + 1), \text{ where } t = 0,\ 1,\ \ldots\}$$

   In the matrix of Fig. 7, the resulting distribution segments are (locales numbered from 1 to 4):

   - $\delta^{-1}(1)$ contains the elements in $DIAG^2$, $DIAG^3$, $DIAG^4$, $DIAG^{14}$, $DIAG^{15}$, $DIAG^{16}$
   - $\delta^{-1}(2)$ contains the elements in $DIAG^5$, $DIAG^6$, $DIAG^7$, $DIAG^{17}$, $DIAG^{18}$
   - $\delta^{-1}(3)$ contains the elements in $DIAG^8$, $DIAG^9$, $DIAG^{10}$
   - $\delta^{-1}(4)$ contains the elements in $DIAG^{11}$, $DIAG^{12}$, $DIAG^{13}$

2. The layout is specified as follows. Let $k, 1 \leqslant k \leqslant p$, denote the number of an arbitrary locale used in this distribution, and $DIAGS\_k$ a domain, whose index set is the set of all diagonals mapped to k via δ. Then we choose to represent the local domain associated with k as the irregular product domain (see [15, p. 115]):

   [DIAGS_k] **domain** (1)

   For each $d \in DIAGS\_k$ the associated subdomain is defined as $1 : length(d)$, where

```
class Banded: Distribution {
  const b: integer;
  const n: integer = getSource()(1).extent();
  const p: integer = getTargetLocales().extent();
  const ndiags: integer = 2*n-1;
  var firstDiagsInDS: [1..p] domain(1);        /* first in every block of b diagonals */
  var diagsInDS: [1..p] seq(integer)= nil;  /* set of all diagonals in distribution segments */

  constructor Banded() {
   forall k in 1..p {
      firstDiagsInDs(k) = (k-1)*b + 2 .. ndiags by b*p;
      forall d in firstDiagsInDs(k) diagsInDs(k) = diagsInDs(k) # d..min(d+b-1,2*n);
   }
  }

  function map(i, j: integer): locale { return Locales(mod((i + j - 2)/b + 1, p)); }

  /* auxiliary functions: */
  /* first component of first element in diagonal d: */
  function first i(d: index(diagD)): integer{return (if (d<=n+1) then 1 else d-n); }
  /* number of elements in diagonal d: */
  function length(d: index(diagD)): integer){return (if (d<=n+1) then d-1 else 2*n-d+1);}
  /* indices of elements in diagonal d: */
  function diag(d: index(diagD)): seq(integer, integer){
   var AD: seq(integer, integer) = nil;
   for s in 0..length(d) - 1 { AD = AD # (first i(d) + s, d - first i(d) - s); }
   return AD;
  }

  iterator DistSegIterator(loc: locale): index(source) {
     const k: integer = locale index(loc);
     for d in diagsInDS {
       for i in first i(d)..first i(d) + length(d) -1 { yield(i, d-i) };
     }
  }

}

class Banded Layout: LocalSegment {
  const loc: locale=this.getLocale();
  const k: integer = locale index(loc);
  const DIAGS k: domain = Banded.diagsInDS(k); /* diagonals in distribution segment loc */

  /* LocalDomain specifies the local index domain of arrays associated with D. Here we chose an
     irregular product domain that binds each local diagonal to an arithmetic domain of rank 1:*/
  const LocalDomain = [DIAGS k] domain(1);

  constructor Banded Layout() forall d in DIAGS k LocalDomain(d)=1..Banded.length(d);

  function layout(i, j: integer): index(LocalDomain) {
   /* Assume k to be the locale index for (i,j), i.e., Banded.map(i,j)=k. The layout function
      performs the mapping (i,j) → (d,l), where d=i+j is the diagonal to which (i,j)
      belongs, and l is the position of (i,j) in the set of all indices belonging to d: */
     return this.index(i + j, i - first i(i + j) + 1);
  }
}
```

Fig. 8. Specification of global mapping and layout for banded array.

$length(d)$ is the number of indices in diagonal d. Thus, an index (i, j) in diagonal $DIAG^d$ can be locally accessed in k in the form (d, l), where l is the position of (i, j) in $DIAG^d$, and positions are counted starting with 1.
For example, the element $(7, 6)$ belonging to diagonal $DIAG^{13}$ is mapped to locale 4, and can be locally accessed there via the pair $(13, 4)$.

The specifications of the class `Banded`, which defines the global mapping, and the class `Banded_Layout`, which determines the locale-internal data arrangement are shown in Fig. 8.

Figure 9 illustrates a program fragment containing the global declarations in the context of a banded array, where the distribution is only referenced in the domain declaration.

5.1.4. Distributed sparse data structures

In terms of building the distribution, the generation of a sparse structure differs from that of a dense domain

```
type eltType;
const n : integer = ...;                                          /*order of matrix */
const bw: integer = ...;                                  /*bandwidth of distribution */
const p : integer = ...;        /*number of locales used: we assume 1 ≤ p ≤ num locales  */
const D: domain (2) distributed (Banded(b=bw),Banded Layout()) on Locales[1..p]=[1..n,1..n];
const diagD: domain (1)=2..2*n;    /* the indices in this domain identify the diagonals of D */

iterator across diagonal(d: index(diagD)): (integer,integer) {
  for i in Banded.first i(d)..Banded.first (d) + Banded.length(d) -1 { yield (i,d-i) };
}

var A: [D] eltType;
var dx: (integer,integer);
...
forall d in diagD {
...
for dx in across diagonal(d) { ... A(dx)= ... }
...

}
```

Fig. 9. Program using banded array.

in at least the following points:

- It is necessary to deal with two domains and their interrelationship: the algorithm writer formulates the program based on the original dense domain, i.e., indexing data collections in the same way as if they were dense. In contrast, the actual representation of the data and the implementation of the algorithm are based on the sparse subdomain of the dense domain.
- In many approaches used in practice, the distribution is determined in two steps:
  1. First, the dense domain is distributed, i.e., a mapping is defined for *all* indices of that domain, including the ones associated with zeroes. In general, this will result in an irregular partition, reflecting the sparsity pattern and communication considerations.
  2. Secondly, the resulting local segments are represented using a sparse format, such as CRS (compressed row storage).

An approach for dealing with this problem is illustrated in a separate report [18].

6. Conclusion

Finding a programming language for HPC systems that combines a high level of abstraction with target code efficiency close to that of programs using explicit message passing is a difficult research issue. Most of the early work in this field has been based on FORTRAN, as the traditional language for scientific programming, but many important ideas arising in this context were later transferred to other languages such as C, C++, Java, and the more recent high-productivity languages.

This paper focused on the problem of high-level language support for locality awareness. We presented a formal framework, which was used to discuss the important features of the High Performance Fortran family of languages. Furthermore, we illustrated how key concepts of Vienna Fortran and HPF were taken over and generalized in the modern object-oriented high-productivity language Chapel, which is currently being developed in DARPA's HPCS program. The framework for user-defined distributions provided in this language is powerful enough to deal not only with standard distributions but to allow the specification of essentially arbitrary global mappings as well as of locale-internal data arrangements that can express distributed sparse collections of data.

The final outcome of the research described in this paper is still open, depending on the outcome of ongoing efforts. For example, the implementation of Chapel is still in progress at this time. Recent language developments include the class of partitioned global address space (PGAS) languages, including Co-Array

Fortran [19], Unified Parallel C (UPC) [33], and Titanium [56]. These languages each provide a set of abstractions for communicating between multiple instances of a single source text running on distinct processors, and represent a reasonable productivity improvement over lower-level communication libraries like MPI. However, they rely on a processor-centric programming model that requires the user to manually decompose their data structures and control flow into per-processor chunks. The one exception to this is UPC, which has support for block-cyclic distributions of one-dimensional arrays over a one-dimensional set of processors (*threads*), and a stylized upc_forall loop that supports an affinity expression to map iterations to threads.

Closer to the goals represented by Chapel are two languages developed along with Chapel in the HPCS program: X10, designed in the PERCS project led by IBM [14,20], and Fortress [2], developed at SUN Microsystems. X10 and Fortress both provide built-in distributions as well as the possibility to create new distributions by combining existing ones. However, they do not contain a framework for specifying user-defined distributions and layouts such as Chapel.

Acknowledgments

This paper is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039 003. The research described in this paper was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

I would like to thank my collaborators in the design of the Chapel language, David Callahan, Bradford Chamberlain and Roxana Diaconescu, for continuously providing ideas and constructive feedback, and exposing interesting issues related to distributions.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas and Jr. Guy L. Steele. Compiling Fortran 8x array features for the connection machine computer system, in: *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, ACM Press, 1988, pp. 42–56.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr. and S. Tobin-Hochstadt. *The Fortress language specification version 1.0 α*, Technical report, Sun Microsystems, Inc., September 2006.

[3] F. Andre, J.-L. Pazat and H. Thomas, Pandore: A system to manage data distribution, in: *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990, pp. 380–388.

[4] J. Backus, The History of FORTRAN I, II and III, *ACM SIGPLAN Notices* **13**(8) (August 1978), 165–180.

[5] S. Benkner, HPF+: High Performance Fortran for advanced industrial applications, *Lecture Notes in Computer Science* **1401**(797) (1998).

[6] S. Benkner and H.Zima, Compiling high performance Fortran for distributed-memory architectures, *Parallel Computing* (1999).

[7] S. Benkner, E. Laure and H. Zima, HPF+: An Extension of HPF for Advanced Industrial Applications, Technical Report TR 99-01, Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, January 1999.

[8] S. Benkner, G. Lonsdale and H. Zima, *The HPF+ Project: Supporting HPF for Advanced Industrial Applications*, Springer Verlag, Toulouse, France, August 1999.

[9] S. Benkner, P. Mehrotra, J. Van Rosendale and H. Zima, High-Level Management of Communication Schedules in HPF-like Languages, in *Proc. International Conference on Supercomputing 1998* (*ICS'98*), Melbourne, July 1998.

[10] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony and B. Mohr, Implementing a Parallel C++ Runtime System for Scalable Parallel Systems, In *Proceedings of Supercomputing '93*, November 1993.

[11] D. Callahan, B.L. Chamberlain and H.P. Zima, The Cascade High Productivity Language, in: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments* (*HIPS'04*), April 2004, pp. 52–60..

[12] B.L. Chamberlain, D. Callahan and H.P. Zima, Parallel programmability and the Chapel language, *International Journal on HPC Applications*, 2007. Special Issue on High Productivity Languages and Models.

[13] B.M. Chapman, P. Mehrotra and H.P. Zima, Programming in Vienna Fortran, *Scientific Programming* **1**(1) (1992), 31–50.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun and V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: *Conference on Object-Oriented Programming Systems, Languages and Applications*, 2005, pp. 519–538.

[15] Cray Inc, *Chapel Specification 4.0*, February 2005, http://chapel.cs.washington.edu/specification.pdf.

[16] A. Darte, J. Mellor-Crummey, R. Fowler and D. Chavarra-Miranda, Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations, *J Parallel Distrib Comput* **63**(9) (2003), 887–911.

[17] R. Das, J. Saltz and R. von Hanxleden, Slicing analysis and indirect access to distributed arrays, in: *Proc.6th Workshop on Languages and Compilers for Parallel Computing*, Springer Verlag, August 1993, pp. 152–168.

[18] R.E. Diaconescu and H.P. Zima, A new approach to Locality Awareness in High Productivity languages, Technical report, Jet Propulsion Laboratory, California Institute of Technology, May 2006. New Technology Report 44028.

[19] Y. Dotsenko, C. Coarfa and J. Mellor-Crummey, A multi-platform Co-Array Fortran compiler, in: *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2004. IEEE Computer Society, pp. 29–40.

[20] K. Ebcioglu, V. Saraswat and V. Sarkar, X10: Programming for hierarchical parallelism and non-uniform data access, in:

3rd International Workshop on Language Runtimes, ACM OOPSLA 2004, Vancouver, BC, October 2004.

[21] S.J. Fink, S.B. Baden and S.R. Kohn, Efficient run-time support for irregular block-structured applications, *J Parallel Distrib Comput* **50**(1–2) (1998), 61–82.

[22] Programmer's reference manual, the FORTRAN automatic coding system for the IBM 704, Technical report, IBM Corporation, Applied Science Division and Programming Research Department, October 1956.

[23] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng and M.-Y. Wu, Fortran D language specification. Technical Report CRPC-TR90079, Rice University, Center for Research on Parallel Computation, Houston, TX, December 1990.

[24] W.K. Giloi, Suprenum: A trendsetter in modern supercomputer development, *Parallel Computing* (7) (1988), 283–296.

[25] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitsberg, W. Saphir and M. Snir, *MPI—The Complete Reference: Volume 2, The MPI Extensions*, Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, September 1998.

[26] L. Hamel, P. Hatcher and M. Quinn, An Optimizing C* Compiler for a Hypercube Multicomputer, In Joel Saltz and Piyush Mehrotra, editors, *Languages, Compilers, and Run–Time Environments for Distributed Memory Machines*. North–Holland, Amsterdam, The Netherlands, 1992.

[27] P. Hatcher, A. Lapadula, R. Jones, M. Quinn and J. Anderson, A production quality c* compiler for hypercube machines, in: *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991, pp. 73–82.

[28] E. Haug, J. Dubois, J. Clinckemaillie, S. Vlachoutsis and G. Lonsdale, Transport vehicle crash, safety and manufacturing simulation in the perspective of high performance computing and networking, *Future Generation Computer Systems* **10** (1994), 173–181.

[29] High Performance Fortran Forum, High Performance Fortran Language Specification, *Scientific Programming* **2**(1–2) (1993), 1–170. (also available as CRPC–TR92225).

[30] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, Rice University, Center for Research on Parallel Computation, January 1997.

[31] High Performance Java, http://www.hpjava.org, 2004.

[32] S. Hiranandani, K. Kennedy and C.-W. Tseng, Compiling fortran d for mimd distributed-memory machines, *Commun. ACM* **35**(8) (1992), 66–80.

[33] P. Husbands, C. Iancu and K. Yelick, A performance analysis of the Berkeley UPC compiler, in: *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, New York, NY, USA, 2003. ACM Press, pp. 63–73.

[34] K. Ikudome, G. Fox, A. Kolawa and J. Flower, An automatic and symbolic parallelization system for distributed memory parallel computers, in: *Fifth Distributed-Memory Computing Conference*, Charleston, South Carolina, April 1990, pp. 1105–1114.

[35] K. Kennedy, C. Koelbel and H.P. Zima, The rise and fall of High Performance Fortran: An historical object lesson, In *Third ACM SIGPLAN History of Programming Languages Conference* (*HOPL-III*), San Diego, CA, June 2007. in preparation.

[36] J. Kepner and N. Travinin, Parallel Matlab: the next generation, in: *Proceedings of the 2003 Workshop on High Performance Embedded Computing* (*HPEC03*), 2003.

[37] C. Koelbel, P. Mehrotra, J. Saltz and H. Berryman, Parallel loops on distributed machines, in: *Fifth Distributed Memory Conference*, Charleston, SC, April 1990, pp. 1097–1104.

[38] J. Li and M. Chen, Generating explicit communication from shared-memory program references, in: *Proceedings of Supercomputing '90*, New York, NY, November 1990, pp. 865–876.

[39] P. Mehrotra and J.V. Rosendale, Programming distributed memory architectures using Kali, in: *Advances in Languages and Compilers for Parallel Computing*, MIT Press, 1991.

[40] P. Mehrotra, J.V. Rosendale and H.P. Zima, High Performance Fortran: History, status and future, *Parallel Computing* **24**(3) (May 1998), 325–354.

[41] J.H. Merlin, Adapting Fortran 90 array programs for distributed-memory architectures, in: *First International ACPC Conference*, H.P. Zima, ed., Lecture Notes in Computer Science 591, Springer Verlag, Salzburg, Austria, 1991, pp. 184–200.

[42] Message Passing Interface Forum, MPI: A Message–Passing Interface Standard, *International Journal of Supercomputer Applications and High Performance Computing* **8**(3/4) (1994), 165–414. (special issue on MPI, also available electronically via ftp://www.netlib.org/mpi/mpi-report.ps).

[43] R.E. Millstein, Control structures in Illiac IV Fortran, *Commun. ACM* **16**(10) (1973), 621–627.

[44] *MIMDizer User's Guide, Version 7.02*, Los Angeles, CA, 1990.

[45] D. Pase, *MPP Fortran Programming Model*, High Performance Fortran Forum, January 1991.

[46] R. Ponnusamy, J. Saltz and A. Choudhary, Runtime compilation techniques for data partitioning and communication schedule reuse, Technical report, University of Maryland, April 1993. UMIACS-TR-93-32.

[47] A.P. Reeves and C.M. Chase, The Paragon programming paradigm and distributed-memory multicomputers, in: *Compilers and Runtime Software for Scalable Multiprocessors*, J. Saltz and P. Mehrotra, eds, Elsevier, Amsterdam, The Netherlands, 1991.

[48] R.L. Diaconescu and H.P. Zima, An approach to data distributions in chapel, *The International Journal of High Performance Computing Applications*, 2006, Special Issue on High Productivity Programming Languages and Models (in print).

[49] A. Rogers and K. Pingali, Process decomposition through locality of reference, in: *Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989, pp. 69–80.

[50] M. Rosing, R.W. Schnabel and R.P. Weaver, Expressing complex parallel algorithms in Dino, in: *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989, pp. 553–560.

[51] R. Rühl and M. Annaratone, Parallelization of fortran code on distributed-memory parallel processors, in: *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990. ACM Press.

[52] M. Shimasaki and H.P. Zima, eds, Special issue on the earth simulator, November 2004.

[53] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI – The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, Massachusetts, 2 edition, 1998.

[54] L. Snyder, *The ZPL Programmer's Guide*, MIT Press, March 1999.

[55] P.S. Tseng, A systolic array programming language, in: *Fifth Distributed-Memory Computing Conference*, Charleston, South Carolina, April 1990, pp. 1125–1130.

[56] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella and A. Aiken, Titanium: A high-performance Java dialect. in: *ACM 1998 Workshop on Java for High-Performance Network Computing*, ACM, ed., New York, NY 10036, USA, 1998. ACM Press.

[57] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series, 1991.

[58] H. Zima, H.-J. Bast and M. Gerndt, SUPERB: A tool for semi-automatic MIMD/SIMD parallelization, *Parallel Computing* **6** (January 1988), 1–18.

[59] H.P. Zima, P. Brezany, B.M. Chapman, P. Mehrotra and A. Schwald, Vienna Fortran – a language specification. Technical Report 21, ICASE, NASA Langley Research Center, March 1992.