The Development of Data-Parallel Programming

Bryan Carpenter
NPAC at Syracuse University
Syracuse, NY 13244
dbc@npac.syr.edu

December 6, 1999

Abstract

This lecture reviews the historical development of programming languages for *data-parallel programming*, and introduces *High Performance Fortran*

Contents

1 Introduction **3**

2 Early data-parallel languages **6**
2.1 ILLIAC IV CFD 6
2.2 ICL DAP Fortran 9

3 Connection Machine Fortran **10**
3.1 Fortran 90 array processing 10
3.2 Array mapping 12
3.3 FORALL 14
3.4 Related languages 16
3.4.1 MasPar Fortran 16
3.4.2 *LISP 16
3.4.3 C* 17

4 High Performance Fortran **18**
4.1 Multi-processing and distributed data 18
4.2 The Processor Arrangement 21
4.3 The Template and distribution 22
4.4 Data alignment 25
4.5 Other alignment options. 27

1 Introduction

A few years ago a discussion about parallel computing might start with words to the effect that "today it seems that serial processors are approaching their technological limits ...", and infer that therefore parallelism would be needed imminently to reach higher performance[1]. In the mean time, Moore's law, which predicts that that sequential processors will double in performance about every eighteen months, has carried on relentlessly. It is a commonplace observation is that today's personal computers have as much processing power as many of the parallel supercomputers we worked on a decade or so ago.

A school of thought that was influential in the supercomputing field for some time held that the most promising parallel computers were those with a modest number of powerful processing nodes. Such a computer can provide results several times faster than its individual sequential nodes. However it can't do much better than that. An obvious question is whether people will invest effort in parallelizing their software for this kind of platform, when they need only wait a few years for Moore's law give them the same speedup. The answer seems to be "not many". As long as Moore's law operates for sequential processors it seems likely that parallelism must offer orders of magnitude improvement in performance, otherwise it is hardly worth the investment needed to exploit it. This suggests that successful parallelism better be *massive* parallelism, involving many nodes approximately as powerful as, or more poweful than, contemporary PCs.

In this climate there are various places where parallel computing can be successfully applied—for example in large dedicated clusters of commodity computers (government labs, Pixar RenderFarm, ...) or by harvesting cycles of many processors over the Internet. Some of these applications most likely involve large-scale "task farming", applicable when a computation can be divided into a large number of completely independent computational tasks. The other prevalent form of massive parallelism is called "data parallelism". The term is usually reserved for the situation where a computation involves some large data-structures, typically arrays, that are split across nodes. Each node performs similar computations on a different part of the data structure. The computations at each node may require some intermediate results from peer nodes. For data parallelism to work best the volume of communicated values should be small compared with the volume of locally computed results. Pretty much all succesful applications of massive parallelism can be characterized as either task farming or data parallel.

In the case of task-farming the level of parallelism is normally *coarse-grained*. The individual task is a sequential program operating on local storage. A few parameters must be read from a remote controller at the start of the task and a few results returned at the end. This style of programming fits naturally in the framework of conventional sequential programming languages: a task can be abstracted as a function or method; the complex, problem-independent

[1] This argument has been made since the late 60s, according to [2], p67.

infrastructure for communication and load-balancing can be abstracted as a library.

In the data-parallel case the situation is slightly different. While it is possible to write data-parallel programs for modern parallel computers using ordinary sequential languages supported by communication libraries, there is a long and quite successful history of special languages for data parallel computing. Why is data-parallel programming a special case—why should it warrant its own languages?

Historically a strong motivation for the development of data parallel languages came from the association of this sort of parallelism with *Single Instruction Multiple Data* (SIMD) computer architectures. In a SIMD computer a single controlling processor reads the program code. The controller broadcasts every instruction to large number of compute nodes, and each executes the same instruction on its local data. The content of an individual instruction depends on the detailed architecture, but a typical example might be an instruction to add together all locally held elements of two vectors distributed over the compute nodes, one element per node.

For much of the time SIMD computers were in vogue (roughly the 1970s to the early 90s) standard sequential languages like Fortran 77 had no features to support this model of execution. Probably the best one could do in a conventional language would be to map the special parallel instructions to individual procedure calls, but that would be tantamount to writing parallel assembly code. To make these machines programable, it was very desirable to extend the sequential languages available at the time.

The early data parallel computer languages developed for machines like the Illiac IV and the ICL DAP were quite elegant, popular with the scientific programmers who used them, and successful in the sense that they allowed the machines to be programmed efficiently by people who would not willingly use a parallel assembly language. They introduced the new programming-language concept of a distributed or parallel array. Typically the set of semantic operations allowed on a distributed array was somewhat different to the operations allowed on a sequential array, but this wasn't really a problem because the compiler could check for abuse. A more serious problem that each data parallel language had features tied to a particular manufacturer's parallel computer architecture. A particularly interesting series of languages (*LISP, C*, CM Fortran) was developed by Thinking Machines Corporation to support their Connection Machine series of computers.

In the 1980s and 90s microprocessors grew in power and availability, and fell in price. Building SIMD computers out of simple but *specialized* compute nodes gradually became less economical than putting a *general purpose* commodity microprocessor at every node. Eventually SIMD computers were displaced almost completely by *Multiple Instruction Multiple Data* (MIMD) parallel computer architectures. In a MIMD architecture every compute node directly executes its own program text. Different nodes can be executing different parts of the same program (or different programs altogether) at the same time.

The asynchronous operation of MIMD computers makes them extremely flexible. For example they are very well suited to the task-farming kind of parallelism, which is barely feasible at all on SIMD computers. But the same asynchrony makes general programming of MIMD parallel computers hard. In SIMD programming issues of synchronization is rarely an issue—every collective step is trivially synchronized by the action of the controller. Programming a MIMD computer on the other hand typically requires some level of mastery of *concurrent programming*, a hard discipline. A major concern is the explicit use of synchronization primitives to control *nondeterministic* behaviour. Nondeterminism arises almost inevitably in a system that has multiple independent threads of control, for example through so-called *race conditions*. Careful synchronization can control nondeterminism. Unfortunately careless synchronization easily leads to a new problem: *deadlock*.

A common style of writing data parallel programs for MIMD computers soon emerged. This style has some similarities to programming a SIMD computer. There is no universally accepted, formal definition of this style, but it is associated with phrases like *Single Program Multiple Data* (SPMD) programming, and the *Loosely Synchronous Model*. In this style, although there is no central controller, the worker nodes carry on doing *essentially* the same thing at *essentially* the same time. Instead of central copies of control variables stored on the control processor of a SIMD computer, control variables (iteration counts and so on) are usually stored in a replicated fashion across MIMD nodes. Each node has its own local copy of these global control variables, but every node updates them in an identical way. There are no centrally issued parallel instructions, but communications usually happen in well-defined collective phases. These data exchanges occur in a rendezvous that explicitly or implicitly[2] synchronizes the peer nodes. The situation is something like an orchestra without a conductor. There is no central control, but each individual is playing from the same script. The group as a whole stays in lockstep. This loosely synchronous style has some similarities to the Bulk Synchronous Parallel (BSP) model of computing introduced by the theorist Les Valiant in the early 1990s. The restricted pattern of collective synchronization is easier to deal with than the complex synchronization problems of general concurrent programming.

A natural assumption was that it should be possible and not too difficult to capture the SPMD model for programming MIMD computers in data-parallel languages, along lines similar the successful SIMD languages. Various research prototype languages attempted to do this, with some success. By the 90s the value of portable, standarized programming languages was universally recognized, and there seemed to be some consensus about what a standard language for SPMD programming ought to look like. The High Performance Fortran (HPF) standard was born.

The remainder of this lecture will review some of these steps on the road to HPF in more detail. Several earlier languages that had HPF-like features will be described. There have been many data-parallel languages. The selection here is somewhat arbitrary, but was partly motivated by the desire to present practical

[2]Depending on whether the platform presents a shared memory or message-passing model.

languages that appear to have been widely used by "real programmers". Many important research contributions are missing.

2 Early data-parallel languages

2.1 ILLIAC IV CFD

In the early 60s Daniel Slotnick worked at Westinghouse on a project called *Solomon*. If it had been completed, the project would have produced the first massively parallel computer. In fact the project folded, and Slotnick moved to the University of Illinois and started the *ILLIAC IV* project with Burroughs, in the latter half of the 60s.

Amongst its technological innovations, ILLIAC IV was the first large system to employ semiconductor primary memory. Development of the system was a very large project for a university, with a final cost of around $30 million. This cost meant the machine had to to be made available to a large community of users, and Slotnick observes [10] that development of ILLIAC IV and ARPANET (which ultimately evolved into the Internet) became closely linked. After many problems, including political instabilities associated with the Vietnam war that caused the hardware to be transferred from its campus base to NASA Ames, the first successful runs occurred in 1973. The machine was not fully operational until 1975. Between that time and 1981 it was the world's fastest computer.

The ILLIAC IV was a SIMD computer for array processing. It consisted of a control unit (CU) and 64 processing elements (PEs). Each processing element had two thousand (2K) 64-bit words of memory associated with it. The CU could access all 128K words of memory through a bus, but each PE could only directly access its local memory. An 8 by 8 grid interconnect joined each PE to 4 neighbours. The CU interpreted program instructions scattered across the memory, and broadcast them to the PEs (Figure 1). Neither the PEs nor the CU were general-purpose computers in the modern sense—the CU had quite limited arithmetic capabilities.

In general software was problematic for the ILLIAC IV. Various ambitious, high-level programming languages were proposed. We will discuss in detail one of the more pragmatic proposals, which seems to have been successfully implemented and used. This language made the architectural features of the ILLIAC IV very apparent to the programmer, but it also contained the seeds of some *practical* programming language abstractions for data-parallel programming.

CFD [11] was a language developed in the early 70s at the Computational Fluid Dynamics Branch of Ames Research Center. CFD was a "FORTRAN-like" language, rather than a FORTRAN dialect in the modern sense (it did not strictly include normal FORTRAN as a subset). The language design was extremely pragmatic. No attempt was made to hide the hardware peculiarities from the user; in fact, every attempt was made to give programmers access and control of all of the Illiac hardware so they could construct an efficient programs.

CFD had five basic datatypes: CU INTEGER, CU REAL, CU LOGICAL,

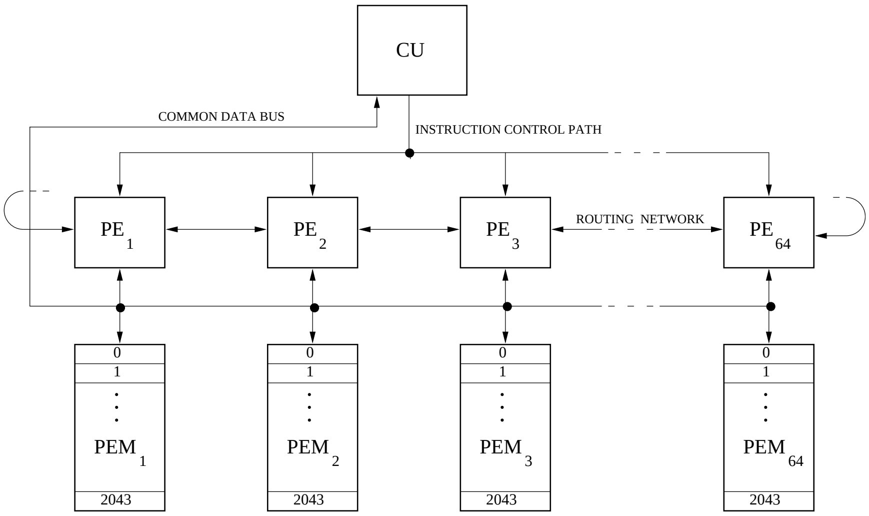

Figure 1: Architecture of the ILLIAC IV. CU = Control Unit, PE = Processing Element, PEM = PE Memory module. The routing network also directly connected PEs at distance 8.

PE REAL, and PE INTEGER. The type of a variable statically encoded its home: either on the control unit or on the processing elements. Apart from restrictions on their home, the two INTEGER and REAL types behave like the corresponding types in ordinary FORTRAN. The CU LOGICAL type was more idiosyncratic—it had 64 independent bits that acted as flags controlling activity of the PEs. We will not discuss it further here.

Scalars and arrays of the five types could be declared as in FORTRAN. An ordinary variable or array of type CU REAL, for example, would be allocated in the (very small) control unit memory. An ordinary variable or array of type PE REAL would be allocated somewhere in the collective memory of the processing elements (accessible by the control unit over the data bus). Here are some ordinary variable declarations:

```
CU REAL A, B(100)
PE INTEGER I
PE REAL D(25), E(1000)
```

Apparently both the CU and PE variables here could be manipulated in ordinary scalar assignments, executed directly by the control unit. There were ad hoc restrictions on what kind of arithmetic operations could appear in such assignments, reflecting the limited processing power of the control unit.

The last data structure available in CFD was a new kind of array called a *vector-aligned array*. This was a very early language instantiation of the *distributed array* concept, which will be a focus of these lectures.

Compared with the distributed arrays of later languages, the CFD vector-aligned array was a primitive structure. Only the first dimension could be distributed, and the extent of that dimension had to be exactly 64. A vector-aligned array would be of PE INTEGER or PE REAL type, and the syntax for the distributed dimension involved an asterisk:

```
PE INTEGER J(*)
PE REAL X(*,4), Y(*,2,8)
```

These are parallel arrays. `J(1)` is stored on the first PE, `J(2)` is stored on the second PE, and so on. Similarly `X(1,1)`, `X(1,2)`, `X(1,3)`, `X(1,4)` are stored on PE 1, `X(2,1)`, `X(2,2)`, `X(2,3)`, `X(2,4)` are stored on PE 2, etc.

Parallel computation occurred only in *vector assignments*. The left-hand-side of the assignment had to be a vector-aligned array, with first subscript `*`. The right-hand-side could be a mix of scalar and vector expressions, with scalar expressions broadcast to all PEs.

A vector expression was a vector-aligned array with a `*` subscript in the first dimension. Communication between neighbouring PEs was captured by allowing the `*` to have some shift added, as in:

```
DIFP(*) = P(* + 1) - P(* - 1)
```

All shifts were cyclic (end-around) shifts, so this parallel statement is equivalent to the sequential statements

```
DIFP(1)  = P(2) - P(64)
DIFP(2)  = P(3) - P(1)
...
DIFP(64) = P(1) - P(63)
```

Essential flexibility was added by allowing vector assignments to be executed conditionally with a vector test, eg:

```
IF(A(*) .LT. 0)  A(*) = -A(*)
```

Less structured methods of masking operations by explicitly assigning PE activity flags in CU LOGICAL variables were also available; there were special primitives for restricting activity to simply-specified ranges of PEs. PEs could concurrently access different addresses in their local memory by using vector subscripts:

```
DIAG(*) = RHO(*, X(*))
```

Vector subscripts could only be used in "sequential" dimensions—*not* in the distributed, first dimension. The language had no built-in syntax for randomly reshuffling of data around the PE array.

In spite of (perhaps even because of) its simplicity and ad hoc machine-dependencies, CFD allowed researchers at Ames to develop a range of application programs that efficiently used the ILLIAC IV. It was thus quite successful.

Another important early language on the ILLIAC was Glypnir [5]. Glypnir was Algol-like rather than FORTRAN-like. It was also quite machine specific and had some similarities to CFD. The distributed array idea was not so explicit—instead there was a concept of a *sword*, which was more suggestive the *pvar* in the later SIMD language *LISP (see Section 3.4.2). Another parallel FORTRAN dialect for the ILLIAC called IVTRAN is discussed in [7]. According to [1] the compiler was never fully debugged.

2.2 ICL DAP Fortran

The ICL DAP (Distributed Array Processor), an early commercial parallel computer, was on the market by the start of the 80s. The basic idea was similar to the ILLIAC IV. The most obvious difference was that the DAP had many more, much simpler processing elements—4096 bit-serial processors arranged in a 64 by 64 grid. Each processing element performed single bit operations rather than floating point or integer arithmetic on complete words. This property was largely concealed by the FORTRAN compiler, which made the usual arithmetic types available through software. Peak floating point performance was considerably lower than the ILLIAC, but the hardware had excellent performance in some other application domains. The DAP was an optional module of the ICL 2900 mainframe computer, and shared main memory and other facilities with the host.

The DAP was programmed using DAP FORTRAN[3]. This language resembled CFD in adding new distributed array structures to the base language. Once again the shape of these structures was constrained to directly match the parallel hardware. But the DAP was more naturally regarded as a two dimensional processor grid, and this led to a richer family of distributed arrays. In general DAP FORTRAN was a higher level language than CFD; the parallel features were tied to a particular parallel architecture, but in other respects it was a fairly normal FORTRAN dialect.

Distributed arrays, called *constrained arrays* could have one or two distributed dimensions. In declarations, the first one or two extents were omitted:

```
DIMENSION A(), BB(,)
INTEGER II(,)
REAL AA(,3), BBB(,,5)
```

`A` is called a *vector*; `BB` and `II` are *matrices*. `AA` is an array of vectors and `BBB` is an array of matrices. The extent of the omitted dimensions was implicitly 64. A syntax reminiscent of the later Fortran 90 allowed sections of arrays to be formed by subscripting certain dimensions and leaving other subscripts empty. As in Fortran 90, array-valued expressions can computed and assigned, and this is the primary means of expressing parallelism. Nearest-neighbour communication is expressed by putting a `+` or `-` in a distributed dimension:

```
U(,) = 0.25 * (U(,-) + U(,+) + U(-,) + U(+,))
```

Shifting further than one place required a call to an array-valued intrinsic function.

The mechanism for restricting execution to a subset of processing elements was more unusual. A logical mask matrix could be used as a subscript in a matrix assignment target:

```
LOGICAL L(,)
...
L(,)  = BB(,) .LT. 0
BB(L) = -BB(,)
```

[3]The discussion here is based on chapter 13 of [8].

Again, despite its hardware-specific nature, DAP FORTRAN was popular with scientific programmers who used it, and was applied to a variety of serious applications.

3 Connection Machine Fortran

Connection Machine Fortran was a later SIMD language that strongly influenced High Performance Fortran. We will discuss it in some detail.

Thinking Machines Corporation was founded in 1983. The original intention was to supply connectionist machines to support artificial intelligence and symbolic computing. As it turned out their *Connection Machine* series of massively parallel computers was mainly exploited for more traditional scientific computation.

The CM-2 launched in 1986 was another SIMD architecture. Like the DAP, the physical PE was bit-serial. Physically it differed from the DAP in being constructed out of a set of *processing nodes* connected through a *hypercube network*. Each processing node contained 32 bit-serial PEs, and an optional floating point coprocessor. The largest CM-2 had 65536 PEs, and a peak performance of 28 GFLOPS[4].

Initially the Connection Machines were programmed in a parallel dialect of LISP called *LISP. This reflected the AI target market. Eventually a Fortran compiler became available [13]. By this time the Fortran[5] 90 standard was in preparation, so the syntax of CM Fortran was strongly influenced by Fortran 90.

Connection Machine Fortran included all of FORTRAN 77, together with the new array syntax of Fortran 90[6]. It added various machine specific features, but unlike CFD or DAP FORTRAN these appeared as *compiler directives* rather than special syntax in Fortran declarations or executable statements. A major improvement over the languages described in the previous section was that distributed array dimensions were no longer constrained to exactly fit in the size of the PE array; the compiler could transparently map dimensions of arbitrary extent across the available processor grid dimensions. Finally the language added an explicitly parallel looping construct called FORALL[7]. Before describing the novel features of CM Fortran in more detail, we will review some of the relevant features of Fortran 90.

3.1 Fortran 90 array processing

In Fortran, any variable or expression is either *scalar*, or has a non-zero *rank*. In the second case we are dealing with an array—the rank is the number of

[4]In the CM the term "processing element" became overloaded. To make best use of the floating point hardware, the software had the ability to optionally treat the 32 PE processing node as a single *logical* processing element.

[5]With this standard the approved spelling of FORTRAN changed to lower case.

[6]It did not include all the other new features of Fortran 90.

[7]A similar construct had appeared much earlier in IVTRAN.

dimensions to the array.

```
REAL A(5, 10), B(5, 10), C(5, 10)
```

declares `A`, `B` and `C` as rank-2 arrays (of real numbers). Their *extent* in the first dimension is 5, and in the second dimension is 10. The vector of extents associated with an array is called the *shape* of the array. `A`, `B` and `C` all have shape $(5, 10)$. Two arrays are *conformable* if they have the same shape. Any of the built-in operations of Fortran 90 can be applied directly to conforming arrays. If `A`, `B` and `C` are declared as above,

```
A + B
B * C
A + (B * C)
```

are legal expressions. Their results are arrays of the same shape as the operands. `B * C` stands for the table of values

$$\begin{array}{ccc} \texttt{B(1,1) * C(1,1)} & \ldots & \texttt{B(1,10) * C(1,10)} \\ \vdots & \ddots & \vdots \\ \texttt{B(5,1) * C(5,1)} & \ldots & \texttt{B(5,10) * C(5,10)} \end{array}$$

Values held in corresponding positions in the arrays are multiplied together. As a special case, scalars are "conformable" with any array. If `D` is scalar

```
REAL D
```

then `C + D` represents

$$\begin{array}{ccc} \texttt{C(1,1) + D} & \ldots & \texttt{C(1,10) + D} \\ \vdots & \ddots & \vdots \\ \texttt{C(5,1) + D} & \ldots & \texttt{C(5,10) + D} \end{array}$$

Array expressions (and variables) can also be formed by taking a *sections* of a named array object. An array section is built from some subset of the elements of an array object—those associated with a selected subset of the index range attached to the object. The simplest array section is formed by subscripting an array by colon-separated range of index values. Suppose an array `X` is declared by

```
INTEGER X(12)
```

Then `X(2:4)` is a section containing the second, third and fourth elements of `X`. It is an array-valued expression of shape (3), and can be used in the same context as any other such expression. Either the lower bound or the upper bound can be omitted in the index range of a section, in which case they default to the lowest or highest values taken by the array's index. So `X(:)` is a section containing the whole of `X`. This simple form of array section can be generalised with a "stride". `X(1:10:3)` is an array section of size 4 containing every third element of `X` with indices between 1 and 10 (ie, indices 1, 4, 7, 10). Collectively ranges like `1:10:3`, and the special cases `1:3` and simply `:`, are all referred to as *triplets*.

For multi-dimensional arrays, some dimensions could be subscripted with a normal scalar expression, and some could be "sectioned" with triplets. The rank of the resulting array variable is the number of dimensions that are triplet-subscripted.

Naturally one can use array-valued expressions in assignments. The simplest version has exactly the same syntax as the scalar Fortran assignment. The expression on the right hand side of the assignment must be conformable with the array variable on the left hand side. A special form of array assignment allows one to restrict the assignment to some subset of the array variable on the left-hand-side by specifying a mask. A mask is `LOGICAL` array expression, conformable with the variable. Assignment of the corresponding element of the array expression on the right-hand is only performed where the mask takes value `.TRUE.`. Where the mask is `.FALSE.`, the variable is left unchanged. For example, suppose `EVEN` is a logical array with indices from 1 to 10, whose even-indexed elements are `.TRUE.` and odd-indexed elements are `.FALSE.`. Then

```
WHERE (EVEN)
  X (1:10) = X (2:11)
ENDWHERE
```

replaces the values in the section `X(1:10)` having even indices with copies of the values of the next element in the array.

The set of Fortran intrinsic functions was extended to include many array operations. Some notable inclusions are the *array reduction* functions including `SUM` and `PRODUCT` which add or multiply together elements of an array; `MAXVAL` and `MINVAL` which return largest or smallest elements of an array; and `ANY` and `ALL`, which take the disjunction and conjunction of the elements of a `LOGICAL` array. Other functions include the `SPREAD` function which "adds an extra dimension" to an array, returning an array in which values from the original array are replicated over the range of the new index; `TRANSPOSE`, which returns an array containing in the elements of its two dimensional argument, transposed; and shift operations which return arrays with values shifted by a constant index offset in one of the dimensions of the argument array.

3.2 Array mapping

Although CM Fortran looked syntactically like standard Fortran, the programmer had to be aware of many nuances. Like the ILLIAC IV, the Connection Machine allowed Fortran arrays to either be distributed across the processing nodes (called *CM arrays*, or distributed arrays), or allocated in memory of the front-end computer (called *front-end arrays*, or sequential arrays). Unlike the control unit of the ILLIAC, the Connection Machine front-end was a conventional, general-purpose computer—typically a VAX or Sun. But there were still significant restrictions on how arrays could be manipulated, reflecting the two possible homes.

As noted above, the shape of a distributed array in CM Fortran was not constrained by the number of physical processing elements in the parallel machine. The new freedom was achieved as follows. First a *Virtual Processor* (VP) grid was defined. By default it would have some minimal shape such that

1. the VP grid was large enough to contain the desired array shape,
2. the extents of the grid were powers of two, and
3. the total number of VPs was an exact multiple of the number of physical PEs.

For example if the array was declared

```
REAL A (10, 64, 100)
```

the compiler might have chosen a VP set with shape (16,64,128). The total number of VPs is 128K—exactly twice the number of physical PEs, assuming one was targetting a 64K processor CM 2. Two virtual processors from the VP set would be assigned to each physical PE[8]. Now the array was mapped into the VP set. The array element at the lower bounds of the dimensions, for example array element (1,1,...), was placed in processor (0,0,...). The extra processors that pad the VP set out to the next legal size were deactivated or ignored during operations on the array. Typically many VP sets would have coexisted in a given program—one for each distinct array shape occuring in the program.

In the absence of special syntax to specify the home of an array in its declaration, CM Fortran adopted the pragmatic convention that any array used anywhere in a Fortran 90 whole-array operation was a CM array. An array that was only ever accessed through Fortran 77 scalar element references was a front-end array. This convention was applied on a per-procedure basis, so the compiler would only have to scan the body of a single procedure to determine if a locally-used array was to be treated as a CM array or a front-end array.

This convention had drawbacks. A CM array did not obey the usual Fortran rules on sequence and storage association. For example, it could not appear in an EQUIVALENCE statement, and the old Fortran 77 ways of passing parts of arrays to procedures or reshaping arrays across the procedure boundaries did not work for CM arrays. Moreover it was illegal to pass a CM array as an actual argument to a procedure that expected a front-end array as its dummy, or vice versa. So the legality of a procedure call might depend not only on the interface of the procedure and the form of the CALL statement, but also on whether caller or callee happened to use the argument in a Fortran-90 array assignment anywhere in their respective bodies—presumably an error-prone arrangement.

The LAYOUT compiler directive allowed the programmer to explicitly specify the home of an array. The LAYOUT directive also gave the programmer

[8] A useful property of the hypercube is that any grid whose extents are powers of two can be embedded in the hypercube graph in such a way that there is an edge between grid neighbours. So a VP grid could always be mapped to the hypercube network so that neighbouring points could communicate directly.

some more refined control over the layout of a distributed array, overriding the default scheme. In general a dimension of a distributed array could be distributed over a dimension of a VP set, or it could be serial—mapped into the memory of a single processing element. Of course similar options had been available in earlier data-parallel languages including CFD and DAP FORTRAN, though CM FORTRAN was a little more flexible in how serial dimensions could be interspersed with parallel dimensions. In

```
      REAL A (10, 64, 100)
CMF$ LAYOUT A(:SERIAL, :NEWS, :NEWS)
```

the directive specifies that `A` is to be laid out with one serial dimension and two parallel ("NEWS-ordered") dimensions. By convention if an array was specified to have all serial dimensions, it was allocated on the front-end. LAYOUT directives could appear in a Fortran 90 *interface specification* for a procedure, providing one way to avoid the pitfalls described in the last paragraph.

Two arrays with the same extents and layouts for all their parallel dimensions have corresponding elements *aligned*. For example in

```
      REAL A (10, 64, 100)
      REAL B (64, 100)
CMF$ LAYOUT A(:SERIAL, :NEWS, :NEWS)
CMF$ LAYOUT B(:NEWS, :NEWS)
```

both arrays are placed in the same VP set, and the elements `A(:,i,j)` are resident on the same processor as the elements `B(i,j)`. Alignment relations like this are extremely important in practise. In constructing practical parallel algorithms, programmers must be very aware of alignment relations between data structures, and should exploit them to minimize communication overheads. Accordingly CM Fortran provided a second directive to explicitly specify that elements of a pair of arrays be aligned.

```
      REAL V(100), B(64, 100)
CMF$ ALIGN V(I) WITH B(1, I)
```

This forces the elements of `V` to be aligned with the first row of B. Note this layout for `V` that cannot be obtained with the LAYOUT directive alone, because it distributes the elements of a one-dimensional array over a two-dimensional VP grid. Even more complex alignment patterns were allowed

```
      REAL C(32,50)
CMF$ ALIGN C(I,J) WITH B(I+5, J+2)
```

These situations are visualized in Figure 2. There were well-defined limits to the complexity of alignment relations. Transposed alignments, for example, were not allowed—alignment dummies like `I`, `J` must appear in the same order in the alignee and alignment target dimensions.

3.3 FORALL

Parallelism in CM Fortran had to be expressed explicitly. The archetypal method of expressing this parallelism was by using the array assignments and

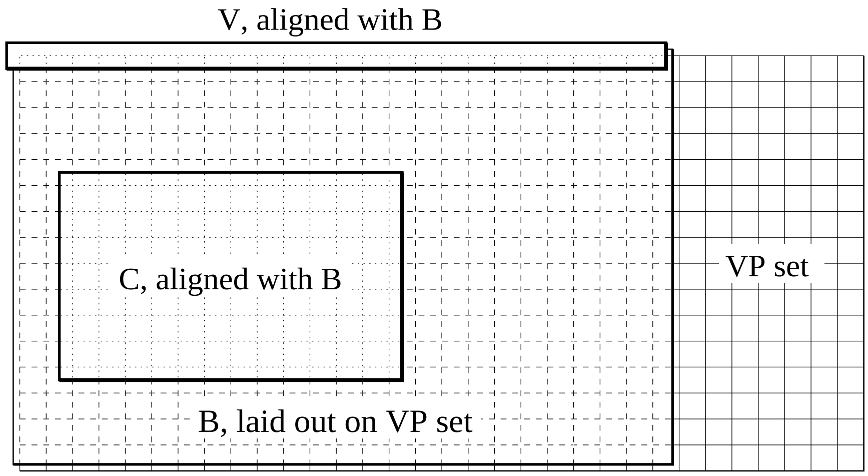

Figure 2: Array layouts and alignments in CM Fortran.

intrinsic functions of Fortran 90. Conceptually this approach was equivalent to the vector assignments of CFD or the matrix assignments of DAP FORTRAN. Nearest neighbour communication, say, could be expressed along the lines

```
U = 0.25 * (CSHIFT(U, 1, -1) + CSHIFT(U, 1, +1) +
            CSHIFT(U, 2, -1) + CSHIFT(U, 2, +1))
```

Used for complex algorithms, the Fortran 90 array syntax often becomes verbose and difficult to read. In fact there are simple parallel operations that are difficult to express efficiently in array syntax at all. CM Fortran provided the powerful FORALL construct for these situations.

A FORALL statement allows a collection of assignments to designated array elements to be executed in parallel. It is a restricted form of parallel loop. The range of each loop index is defined by a triplet. To assign each element of the array IDENTITY with its own index value we could execute

```
FORALL(I = 1:N)  IDENTITY(I) = I
```

A nearest neighbour update might look something like

```
 FORALL (I = 2:N-1, J = 2:N-1)
&    U(I,J) = 0.25 * (U(I,J-1) + U(I,J+1) + U(I-1,J) + U(I+1,J))
```

FORALL assignments had certain restrictions. In particular, no left-hand-side element could be assigned more than one value. A necessary but not sufficient condition is that subscript expressions in the left-hand-side must depend in some way on *all* loop indices. As in ordinary Fortran 90 array assignments, the semantics was supposed to be *as if* all expressions on the right-hand-side (for all values of loop indices) were completely evaluated before any assignments to the left-hand-side variables occurred[9].

[9] Technically, there are no loop-carried flow dependences.

In spite of these restrictions, a general FORALL statement was a rather complicated thing to translate efficiently, especially considering that the assignment in the body of the FORALL could itself be an array assignment, or the expression on the right-hand-side could involve array intrinsics, as in the "parallel prefix":

```
      FORALL (I = 1:N)  SCAN(I) = SUM(A(1:I))
```

or the matrix multiplication:

```
      FORALL (I = 1:N, J = 1:N)  C(I, J) = DOT_PRODUCT(A(I, :), B(:, J))
```

In complex cases the CM Fortran had to resort to translating the FORALL statement as a serial loop.

3.4 Related languages

3.4.1 MasPar Fortran

MasPar Computer Corporation was formed in 1988. The company was closely associated with Digital Equipment Corporation. The MasPar MP-1 was a 2D mesh (up to 128×128) of simple, 32-bit, processing elements, 16 per custom chip. Its front-end was a DEC computer.

MasPar Fortran [6] was rather similar to CM Fortran: it was a version of Fortran 77 extended with the Fortran 90 array processing features, a FORALL statement, and data-mapping directives. The default criterion for allocating arrays on the Data Parallel Unit (DPU) or front-end was the same as in CM Fortran. This default could be overridden by directives ONDPU or ONFE, eg:

```
      REAL A(100, 100), B(100, 100)
CMPF  ONDPU A, B

      CALL FOO(A, B)
```

A MAP directive was used in much the same way the LAYOUT directive of CM Fortran:

```
      REAL A(100, 100, 100)
CMPF  MAP A(XBITS, YBITS, MEMORY)
```

This would specify that the first two dimensions of the array were to be distributed over the x and y dimensions of the grid and the last is serial.

3.4.2 *LISP

The earliest language for programming the connection machine had been a dialect of Common LISP called **LISP* [12].

*LISP programs were executed by the front-end, and referred to memory in the Connection Machine processors through LISP objects called *pvars* (parallel variables). These objects contained information about a field in the Connection Machine memory where some *contents* were held, together with type information. Values were stored one per PE. So a pvar was *essentially* like a simple distributed array (as in CFD, for instance). However the point of view was slightly different.

The names of instructions that return pvars as their values ended with `!!` (supposed to be suggestive of two parallel lines). The names of other special parallel operations (which did not essentially return a pvar) started with `*`. *LISP operations could be carried out in a subset of the CM processors. The *currently selected set* was specified by using special forms such as `*all`, `*when`, `*cond` and `*if`.

Communication between processors had to specified explicitly with suitable pvar functions. A nearest neighbour update might look like

```
(*set a (+!! (pref-grid-relative!! a (!!  0) (!! -1))
             (pref-grid-relative!! a (!!  0) (!!  1))
             (pref-grid-relative!! a (!! -1) (!!  0))
             (pref-grid-relative!! a (!!  1) (!!  0))
        ))
```

where `pref-grid-relative!!` is a kind of shift operation.

3.4.3 C*

A slightly later language for the Connection Machine, C* was a data-parallel dialect of C. This language was first announced in 1987, and overhauled considerably for 1990. The discussion here is based on section 4.3 of [9].

C* introduced the idea of a *shape*, which was used to specify how parallel data was organized.

```
shape [128] [128] grid ;
real:grid a, b, c ;
```

This fragment would create a 128×128 shape called `grid`, and declare two parallel arrays `a` and `b` having this shape.

Parallel operations usually occur within the context of a *with* statement. This construct activates the positions of the shape, and sets the context for the parallel variables manipulated within its body. For example:

```
with(grid) {
    a = b + c ;
}
```

C* also had a builtin syntax for reduction operations, overloading the `+=`, etc, accumulating assignments of C. It had a *where* construct similar to Fortran 90:

```
with(grid) {
    where(a > 0.0) {
        ...
    }
}
```

A function `pcoord(d)` returned the logical coordinate in the `d`th dimension of the currently active shape.

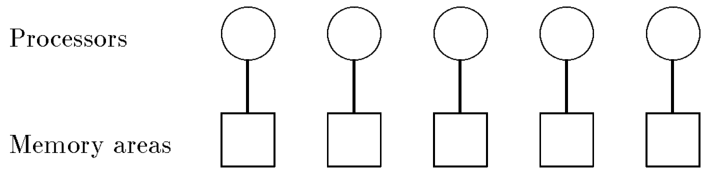

Figure 3: Idealised picture of a distributed memory parallel computer

4 High Performance Fortran

In 1993 the *High Performance Fortran Forum*, a coalition of many leading industrial and academic groups in the field of parallel processing, established an informal language standard called *High Performance Fortran* (HPF) [3, 4]. It was based on Fortran 90, but extended the set of parallel features, and provided extensive support for computation on *distributed memory* parallel computers. The standard was supported by a majority of vendors of parallel hardware, including Cray, DEC, Fujitsu, HP, IBM, Intel, Maspar, Meiko, nCube, Sun and Thinking Machines. Several companies announced their intention to develop HPF implementations, including hardware vendors DEC, Intel, Maspar, Meiko, Thinking Machines, and compiler vendors ACE, APR, KAI, Lahey, NA Software, Portland and PSR.

Since then several of those companies have gone out of business. With a few notable exceptions (such as Portland) many of those still in business have abandoned their HPF projects. With hindsight the goals of HPF were hugely ambitious—it tried to put too many new ideas into a single language. We consider that although the HPF language as originally defined might never be widely adopted, many of the ideas—for example its standardization of a distributed data model for SPMD computing—remain important. Therefore several ideas from HPF will be presented in detail here.

4.1 Multi-processing and distributed data

The most successful parallel computers presently available are built from a number of largely autonomous processors, each possessing its own local memory area (Figure 3). A processor can access values in its local memory very rapidly. It can also access values in another processor's memory, either by special machine instructions, or through message-passing software. But with current technology remote memory accesses are much slower (often by orders of magnitude) than local ones. This puts an onus on the programmer or compiler to minimise the number of accesses any processor makes to non-local memory. This is the *communication* problem.

A second problem is ensuring that each processor has a fair share of the total work load. Obviously, if most work ends up being done by one processor the benefits of parallelism are lost. This is the *load-balancing* problem.

High Performance Fortran allows the programmer to add various *directives*

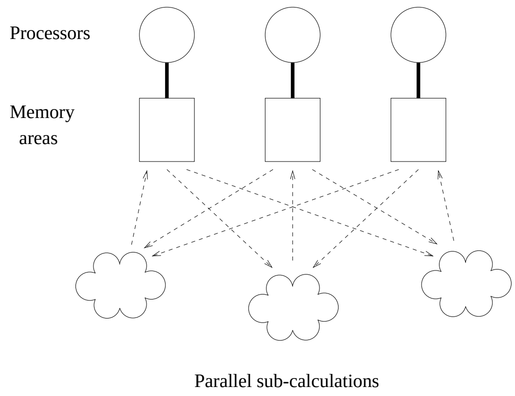

Figure 4: A data distribution leading to excessive communication

to a program. These tell the compiler how program data is to be distributed amongst the memory areas associated with a set of (real or virtual) processors. They *don't* allow the programmer to state directly which processor will perform a particular computation. But it is expected that if the operands of a particular sub-computation (for example, an assignment) are all found on the same processor, the compiler will allocate that part of the computation to the processor holding the operands, whereupon no remote memory accesses will be involved.

Hence the main burden on the HPF programmer is to distribute the program's data across the processors in such a way that

- as many sub-calculations as possible involve groups of operands allocated to the same processor, to minimize remote memory accesses, while
- the set of *independent* sub-calculations that can proceed in parallel at any time should involve data on as many different processors as possible, to maximise parallelism.

These may sound like conflicting goals. Sometimes they are, and programs which look highly parallel are nevertheless difficult to distribute effectively. Luckily, equally often it is possible to perform the juggling act successfully.

To make these points a bit more concrete, imagine an in idealised situation where the a particular step in a program contains p sub-calculations which can proceed in parallel. They might be the p elemental assignments making up an array assignment or `FORALL` construct. For the sake of argument, suppose each expression to be computed involves two operands. Together with the variable being assigned, this makes three operands for the whole sub-calculation. Also, suppose there are p processors available. (In general, the numbers of sub-calculations and processors are likely to be different—this case is just illustrative).

Figure 4 illustrates the situation where all operands of each sub-calculation reside in different memory areas. This means that each assignment involves at least two communications, wherever the calculation is performed.

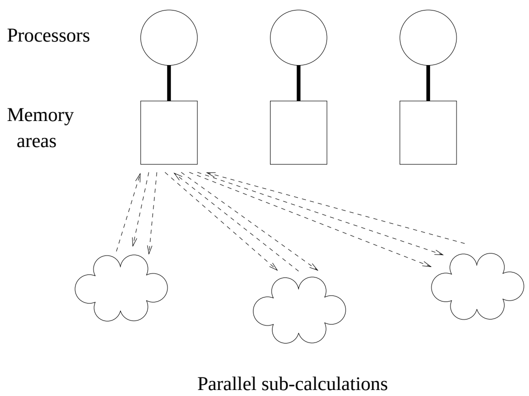

Figure 5: A data distribution leading to poor load balancing

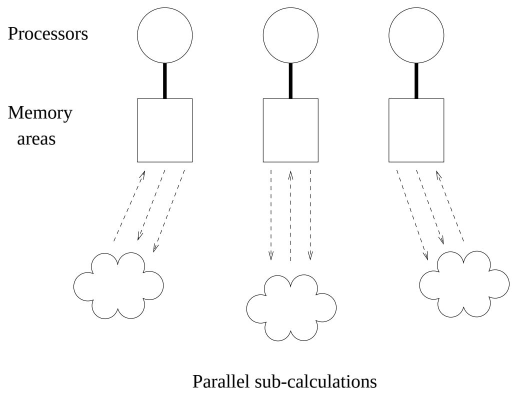

Figure 6: An ideal data distribution

Figure 5 illustrates a situation in which no communication is *necessary*, because all operands of *all* the assignments live on the same processor. But in this case the calculation will generally be performed where the data is, so no effective parallelism occurs. (Alternatively, the compiler might decide to share the tasks out anyway. But then *all* operands of tasks on processors other than the first would have to be communicated.)

Figure 6 illustrates the ideal situation, where all operands of an individual assignment occur on the same processor, but the operand groups of the sub-calculations are uniformly distributed over processors. In this case we can rely on the compiler to allocate each computation to the processor holding the operands, requiring no communication, but perfect distribution of workload.

Except in the most trivial programs, it will never be possible to choose a distribution of program variables over processors so that *all* steps in the calculation look like figure 6. The art of distributed memory programming is to arrange things so that the most critical regions of the program look as much like this as possible.

There was a certain awkwardness to the descriptions of these three situations. In the last case one could generally rely on the compiler to allocate sub-calculations to processors in the obvious way. In the first two cases one has to hedge one's bets, and say that compiler might allocate computations over processors in various ways, but in any case the outcome is likely to be unfavourable. Given that HPF places an onus on the programmer to think about these kinds of issues, one can argue that it would have been appropriate to allow the programmer fuller control over where particular sub-computations were performed. Some of these issues were addressed in HPF 2.0 by a new ON directive.

4.2 The Processor Arrangement

The programmer is expected to specify how program variables are distributed over the memory areas associated with a set of processors. So it is desirable for a program to be able define some representation of the set of processors it uses. This can be done through the `PROCESSORS` directive.

The syntax of a `PROCESSORS` definition is similar to the syntax of an array definition in Fortran:

```
!HPF$ PROCESSORS P (10)
```

This introduces a set of 10 abstract processors, assigning them the collective name `P`. Like arrays, abstract processor sets, or *processor arrangements*, can be multi-dimensional. For example

```
!HPF$ PROCESSORS Q (4, 4)
```

introduces 16 abstract processors in a 4 by 4 array. There are a couple of reasons why one might want to declare a processor arrangement to be multi-dimensional. Most importantly, the major data-structures in a program may be multi-dimensional arrays, and it may be that the best way to divide up this data over processors is by sub-dividing it into blocks of the same dimension as the whole arrays. In this situation it certainly simplifies matters if we can view the processor arrangement itself as an array of the relevant dimension.

An important point to appreciate is that a single program may contain many `PROCESSORS` declarations. There is no defined relation between two processor arrangements of different shapes[10]. Typically they will be independently mapped onto the same set of underlying physical processors.

4.3 The Template and distribution

Having seen how to define one or more target processor arrangements, we need to introduce mechanisms for distributing data arrays over those arrangements. HPF allows arrays to be distributed over processors directly (see Section 4.5), but it is often more satisfactory to go through the intermediary of an explicit *template*. HPF templates are used in ways reminiscent of the implicit *VP set* of CM Fortran or the *shape* of C*. In the MIMD world anticipated by HPF, a template is distinct from a processor arrangement. The set of abstract processors in an HPF processor arrangement might not exactly match the set of physical processors, but there is a tacit assumption that abstract processors will be used at a similar level of granularity to the physical processors. Usually it would be inappropriate for the shape parameters of the abstract processor arrangement to correspond to those of the data arrays of the algorithm. Instead the fine-grained grid of the data arrays is captured in the template concept.

Figure 7 represents the HPF data mapping scheme: the rest of this section is concerned with the bottom half of the diagram. Mapping of arrays to templates will be discussed in the next section.

A template can be declared in much the same way as a processor arrangement.

```
!HPF$ TEMPLATE T(50, 50, 50)
```

delares a 50 by 50 by 50 three-dimensional template called `T`. Having declared it, we usually want to establish a relation between a template and some processor arrangement. We want to say in more or less detail how the elements of the template are distributed amongst the elements of the processor arrangement. This is done by using DISTRIBUTE directive.

As a first example, suppose we have

```
!HPF$ PROCESSORS P1(4)
!HPF$ TEMPLATE T1(17)
```

There are various ways in which `T1` may be distributed over `P1`. The four basic distribution formats are illustrated in figure 8. In the figure, each template element inhabits a particular memory area. This should be taken to mean that any data element aligned with a particular template element will be stored in

[10] There is however, an implicit equivalence between processor arrangements of the *same* shape.

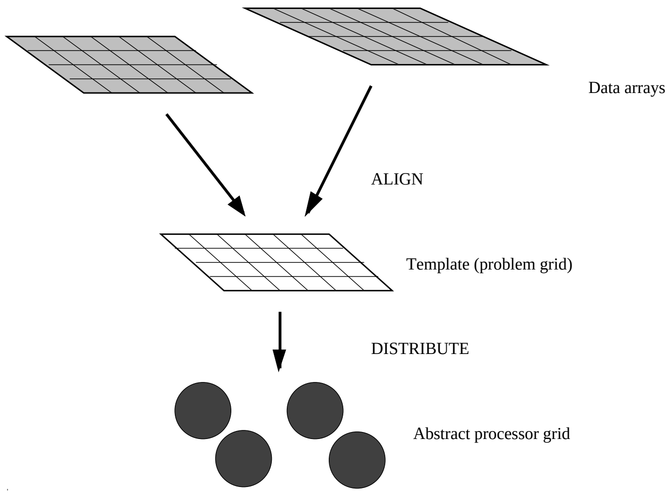

Figure 7: The stages of data mapping in HPF.

the associated memory area (the template element itself doesn't occupy any memory).

Simple block distribution is specified by

```
!HPF$ DISTRIBUTE T1(BLOCK) ONTO P1
```

In this case, each processor gets a contiguous block of template elements. All processors get the same sized block, unless the number of processors doesn't divide the number of template elements. In this case the template elements are divided evenly over most of the processors, with some trailing processor(s) having less (or zero).

Simple cyclic distribution is specified by

```
!HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1
```

The first processor gets the first template element, the second gets the second, and so on. When the set of processors is exhausted, go back to the first processor, and continue allocating the template elements from there.

In a variant of the block distribution, the number of template elements allocated to each processor can be explicitly specified, as in

```
!HPF$ DISTRIBUTE T1 (BLOCK (6)) ONTO P1
```

If this means that we allocate all template elements before exhausting processors, some processors are left empty. It is *illegal* with to choose a block size here which cause template elements to be left over after all processors have had their blocks allocated. But in an analogous variant of the cyclic distribution ("block-cyclic distribution")

```
!HPF$ DISTRIBUTE T1 (BLOCK (3)) ONTO P1
```

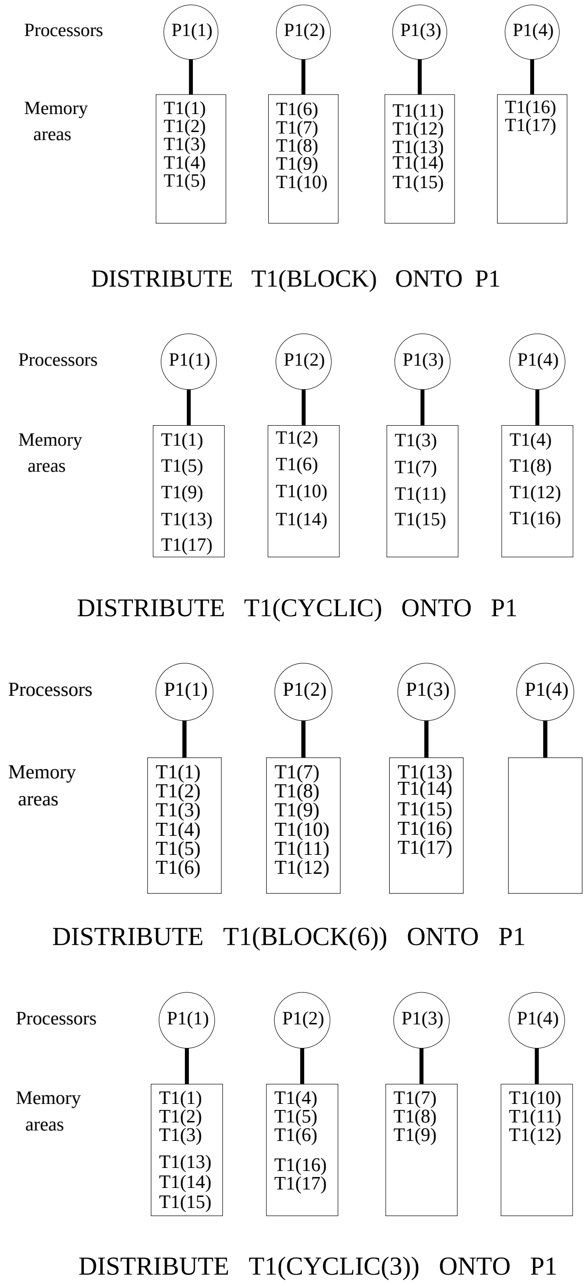

Figure 8: The four basic formats of template distribution

the product of the number of processors with the block size can be smaller than the template size, and allocation wraps round after the first assignment of blocks to all processors.

That covers the case where both template and processor are one dimensional. When the template both have (the same) higher dimension, each dimension can be distributed independently, mixing any of the four distribution formats. The correspondence between the template and the processor dimension is the obvious one. In

```
!HPF$ PROCESSORS P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (CYCLIC, BLOCK) ONTO P2
```

the first dimension of `T2` is distributed cyclically over the first dimension of `P2`; the second dimension is distributed blockwise over the second dimension of `P2`.

Finally, some dimensions of a template may have "collapsed distributions", allowing a template to be distributed onto a processor arrangement with fewer dimensions than the template. So

```
!HPF$ DISTRIBUTE T2 (BLOCK, *) ONTO P1
```

means that the first dimension of `T2` will be distributed over `P1` as was `T1` in the first example above. But for a fixed value of the first index of `T2`, all values of the second subscript are mapped to the same processor.

4.4 Data alignment

Arrays are aligned to templates through the ALIGN directive. Before describing this directive, we introduce a motivating example. The core of an LU decomposition subroutine might look like this:

```
      REAL A (N, N)
      INTEGER N, R, R1
      REAL, DIMENSION (N) :: L_COL, U_ROW

      DO R = 1, N - 1
        R1 = R + 1
        L_COL (R  : ) = A (R : , R)
        A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
        U_ROW (R1 : ) = A (R, R1 : )
        FORALL (I = R1 : N, J = R1 : N)
     &    A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
      ENDDO
```

A cursory inspection of this algorithm should suggest that a possible choice of template for the problem is

```
!HPF$ TEMPLATE T(N, N)
```

This template exactly matches the main data structure of the problem, namely, the array `A` which holds the matrix. To align `A` to `T` we can add an `ALIGN` attribute declaration for `A` as follows

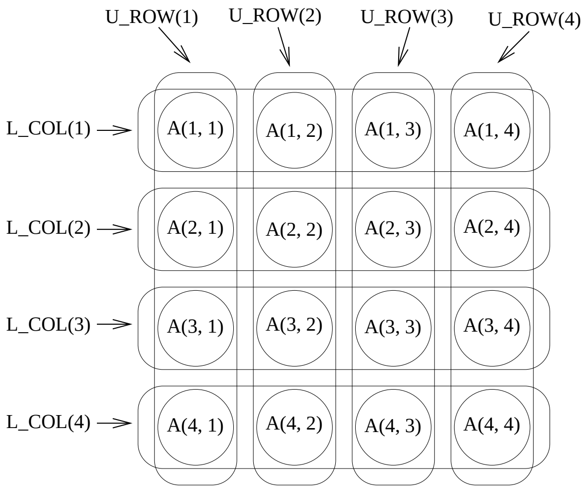

Figure 9: Alignment of the three arrays in the LU decomposition example

```
!HPF$ ALIGN A(I, J) WITH T(I, J)
```

This is almost identical to the ALIGN directive of CM Fortran. `I` and `J` act as *alignment dummies*. An alignment dummy is a named integer variable appearing as a subscript of the "alignee" (the array which is to be aligned) in an `ALIGN` directives. The idea is that the variable ranges over all possible index values associated with the array dimension involved. The "align target" (in this case the template `T`) can have expressions involving the alignment dummies amongst its subscripts. In this way, every element of the alignee is mapped to some element of the template.

How should the other arrays in the program, `L_COL` and `U_ROW` be mapped? Most of the work in a single iteration is in the statement

```
      A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
```

We can minimise (indeed, eliminate) the need for communication in this assignment if `L_COL (I)` is stored wherever `A (I, J)`, for any `J` is stored, and if `U_ROW (J)` is stored wherever `A (I, J)` for any `J` is stored. This can be specified by using a replicated alignment to the template `T`:

```
!HPF$ ALIGN L_COL (I) WITH T (I, *)
!HPF$ ALIGN U_ROW (J) WITH T (*, J)
```

where an asterisk as a template subscript means that data is replicated in the corresponding processor dimension. Figure 9 is an attempt to visualise the alignment of the three arrays and the template.

As claimed, the main assignment in the `FORALL` construct will require no communications, because all operands of each elemental assignment will be stored on the same processor. What about the other statements?

```
A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
```

requires no communication. It is equivalent to

```
FORALL (J = R1 : N)  A (R, J) = A (R, J) / L_COL (R)
```

and we know that `L_COL (R)` will be stored on any processor where `A (R, J)` is stored.

The other two array assignments *do* require communication. For example, the assignment to `L_COL` is equivalent to

```
FORALL (I = R : N)  L_COL (I) = A (I, R)
```

Because `L_COL (I)` is replicated in the `J` direction, whereas `A (I, R)` is held only on the processor holding the $J = R$ template element, updating the `L_COL` element involves broadcasting the `A` element to all interested parties. The compiler will duly insert these communications. Similarly, the `U_ROW` assignment involves broadcasts, this time in the `I` direction.

Having chosen a plausible alignment of our arrays to a template, we have to distribute that template. `BLOCK` distribution isn't particularly attractive for this computation, because successive iterations work on a shrinking area of the template. A block distribution could leave whole processors idle in later iterations. A `CYCLIC` distribution will achieve better load balancing (Figure 10). Our final HPF program, distributed onto an `M` by `M` processor arrangement, is displayed in Figure 11.

4.5 Other alignment options.

In the example given above the alignments illustated were "identity mapping" between array and template, and replicated alignments. More general alignment relations are possible.

For one thing, indices can be permuted:

```
      DIMENSION B(N, N)
!HPF$ ALIGN B(I, J) WITH T(J, I)
```

maps `B` to `T`, but in a "transposed" manner: `B (1, 2)` is mapped to `T (2, 1)`, etc. More generally, a subscript of an *align target* (not an alignee) can be a linear expression in *one* of the alignment dummies (with integer expression coefficients). For example

```
      DIMENSION C(N / 2, N / 2)
!HPF$ ALIGN C(I, J) WITH T(N / 2 + I, 2 * J)
```

Figure 12 illustrates how `C` would be aligned to `T`.

The rank of alignee and align-target do not have to match. A dimension of the alignee may be "collapsed"; or some dimension of the align-target might be a constant—for example a scalar might be aligned to the first element of a one-dimensional template; or the alignee may be "replicated" over some dimensions of the template. As an example of collapsing:

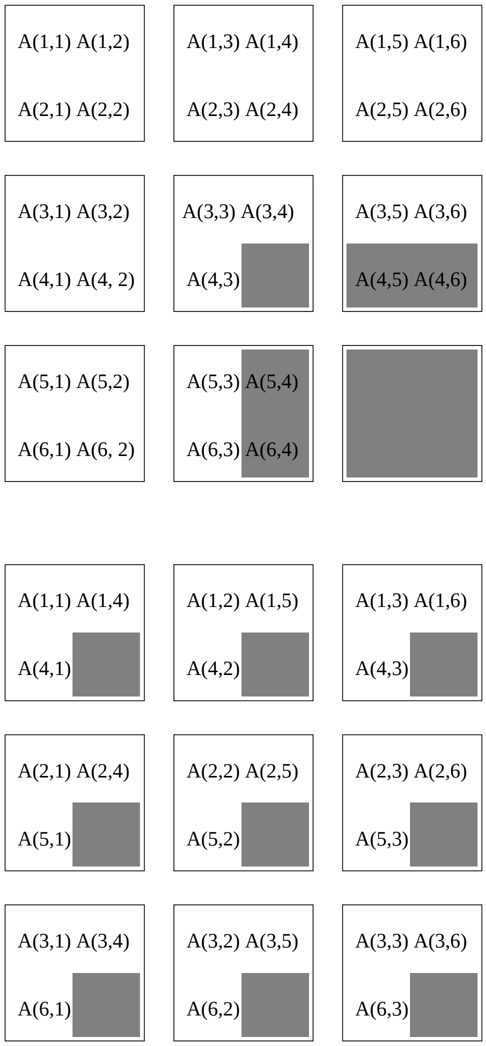

Figure 10: **A** elements involved (shaded) in iteration **R**=4, when **N**=6 and processor arrangement is 3 by 3. Block and cyclic distributions are illustrated.

```
      INTEGER M, N, R, R1

!HPF$ PROCESSORS P (M, M)

!HPF$ TEMPLATE T (N, N)
!HPF$ DISTRIBUTE T (CYCLIC, CYCLIC) ONTO P

      REAL A (N, N)
!HPF$ ALIGN A (I, J) WITH T (I, J)

      REAL, DIMENSION (SIZE (A, 1)) :: L_COL, U_ROW
!HPF$ ALIGN L_COL (I) WITH T (I, *)
!HPF$ ALIGN U_ROW (J) WITH T (*, J)

      DO R = 1, N - 1
        R1 = R + 1
        L_COL (R  : ) = A (R : , R)
        A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
        U_ROW (R1 : ) = A (R, R1 : )
        FORALL (I = R1 : N, J = R1 : N)
     &    A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
      ENDDO
```

Figure 11: Final HPF implementation of LU decomposition.

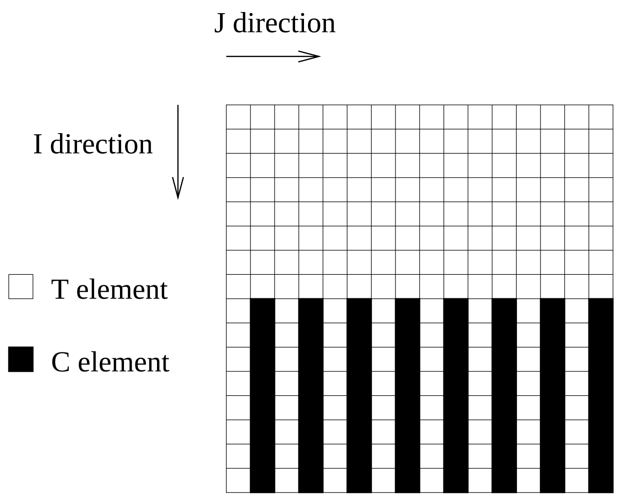

Figure 12: Example alignment of the array **C** to the template **T**

```
      DIMENSION D(N, N, N)
!HPF$ ALIGN D(I, J, K) WITH T(I, J)
```

The template element is independent of `K`. This means that for fixed `I`, `J`, all elements of `C` differing only in their third index mapped to the same template element.

Of course there are restrictions: as mentioned, the expressions used to subscript a template must be linear in the alignment dummies, with integer coefficients; each subscript can depend on at most *one* of the alignment dummies (local to the `ALIGN` directive); and each alignment dummy can appear in at most *one* of the subscript expressions for the template. Secondly, we should say that there are various shorthand forms of the alignment directive which were not described here—they don't add any new facilities.

The alignments described so far were all direct alignments to templates. It is also allowed to align an array to another array, using all the normal syntax of the `ALIGN` directive. So the alignments of `L_COL` and `U_ROW` in the example of the previous section could be written as

```
!HPF$ ALIGN L_COL(I) WITH A(I, *)
!HPF$ ALIGN U_ROW(J) WITH A(*, J)
```

In general, the meaning is that the alignee element is stored wherever the align-target array element is. Any such alignment *can* be written as an alignment to a template (provided the template is in scope). If there are several levels of indirection (array aligned to array aligned to array, etc) writing the "ultimate alignment" directly could be relatively complicated.

Finally, templates can be left completely implicit, with DISTRIBUTE directives can applied directly to arrays. So, in the example, the `T` directives could be omitted, and we could add the directive

```
!HPF$ DISTRIBUTE A (CYCLIC, CYCLIC) ONTO P
```

The implicit template has the same shape as the array distributed. It is *not* allowed for any array to be both distributed in this way *and* to appear as an alignee in an `ALIGN` directive. So this idiom can always be regarded as shorthand for introduction of an explicit template.

The last two conventions are particularly useful for procedure dummy arguments, because templates can't be explicitly passed through the subprogram interface.

References

[1] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 1989.

[2] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[3] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[4] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel, Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[5] D.H. Lawrie, T. Layman, D. Baer, and J.M. Randall. Glypnir—a programming language for Illiac IV. *Communications of the ACM*, 18(3):157–164, 1975.

[6] MasPar Computer Corporation, Sunnyvale, California. *MasPar Fortran User Guide. Software Version 1.1*, 1991.

[7] Robert E. Millstein. Control structures in Illiac IV Fortran. *Communications of the ACM*, 16(10):621–627, 1973.

[8] R.H. Perrot. *Parallel Programming*. Addison-Wesley, 1987.

[9] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[10] D. L. Slotnick. The conception and development of parallel processors—a personal memoir. *Annals of the History of Computing*, 4(1):20–30, 1982.

[11] K. G. Stevens, Jr. CFD—a FORTRAN-like language for the ILLIAC IV. *ACM SIGPLAN Notices*, 10(3):72–76, 1975.

[12] Thinking Machines Corporation, Cambridge, Massachusetts. *The Essention *LISP Manual, Release 1*, 1986.

[13] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Programming Guide, Version 1.0*, 1991.