# *Chapter 12*

# *Languages and Compilers*

*Ken Kennedy & Charles Koelbel*

*Target Length: 20 pages*

Because parallel computing is significantly more complicated than serial computing, it places significant burdens on the application developer. In addition to developing a correct problem solution, the user must also control and coordinate the uses of parallelism in the solution program. This is even more complicated because the methodology and mechanisms for exerting control may differ from target platform to target platform.

As a result, high performance parallel computing presents significant challenges and opportunities for programming language designers, compiler implementors, and run-time system developers. If program development support software can make it easier for programmers to design, implement, debug and tune parallel programs on a variety of target platforms, parallelism may become accessible to the larger community of application developers.

To understand the nature of the challenge for programming support software, we must consider the process for development of parallel applications. The implementor must be able to find opportunities for parallelism in his or her application, express the paralellism in a machine-independent way, and debug and tune the resulting application for a particular parallel platform. In designing language and compiler support for this process, we must keep three goals firmly in mind:

- Programming should be as easy as possible—that is, the end user should experience only slightly more complexity than for development of uniprocessor programs.
- The resulting programs should be portable across platforms with modest effort. It should not be necessary to maintain multiple source versions of the same program. Rather it should be possible to move the same source

to each platform, adjust some tuning parameters, and run with nearly the full performance available on the machine.

- The programmer should retain as much control over performance as possible. If performance problems develop, it should be possible for the programmer to address them within the high-level programming model—i.e., without having to resort to modification of low level code generated from the high-level representation.

These seemingly conflicting goals will be difficult to achieve because parallelism presents significant challenges to the application developer. To achieve the goals above, the programming system will need to solve three fundamental problems:

1. It must find extensive parallelism in the application presented by the user. It must then package and coordinate that parallelism during execution. Finding parallelism may involve a transfer of information from the user, but is must be possible to get this information without forcing a complete revision of the application.

2. It must overcome the performance penalties due to the complex memory hierarchies on modern parallel computers. This could involve extensive program transformations to increase locality. This is more challenging on parallel computers because there is often a tension between increasing parallelism and finding locality.

3. It must support migration of parallel programs to different architectures with only modest changes. This will entail devlopment of a programming interface that is not machine specific and strategies for optimizing and tuning applications for different architectures.

In designing strategies for support of parallel programming, we must keep in mind the principle that each component of the system should do what it does best:

- the *application developer* should be able to concentrate on problem analysis and decomposition at a fairly high level of abstraction;

- the *system*, including the programming language and compiler, should handle the details of mapping the abstract decomposition onto the computing configuration available at any given moment; and

- the *application developer and the system* should work together to produce a correct and efficient program through the use of execution monitoring, debugging, and tuning tools.

This chapter explores three technologies that have been reasonably successful in meeting the goals of parallel programming support for scientific computation: automatic parallelization, data parallel languages (High Performance

Fortran), and shared memory parallel programming interfaces (OpenMP). (Parallel object-oriented programming is considered in Chapter **??**.) The intent of our presentation is to give a somewhat tutorial introduction to these technologies, while providing background on the intellectual development that led to them and an assessment of their usefulness.

## 12.1 Automatic Parallelization

From the user's perspective, the most appealing approach to program decomposition is automatic parallelization. If a fully automatic system could efficiently parallelize applications, the user would be free to concentrate on *what* is being computed rather than *how* it is being computed. However, to be acceptable, a fully automatic scheme must generate code that achieves performance competitive with programs hand-coded by experts—the performance penalty should be no worse than a factor of two. This observation is based on strong evidence that object program performance has been a significant factor in the acceptance of new programming languages since the original Fortran I compiler, including this reflection by John Backus [12]:

> It was our belief 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... To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

Automatic parallelization research began in the 1970s as *automatic vectorization*, a technology to support portable programming on vector processors. The important technological tool used in automatic vectorization is *dependence analysis*, which seeks to determine whether pairs references to the same data structure (usually a subscripted variable) may access the same memory location [39, 40]. An example loop suggests how this can be done:

```
REAL A(1000,1000)
DO J = 2, N
  DO I = 2, N
    A(I,J) = (A(I,J+1)+ 2*A(I,J) + A(I,J-1))*0.25
  ENDDO
ENDDO
```

Any particular element in the interior of the array, say `A(`$m_1$`,`$m_2$`)` will be accessed on three iterations of the loop nest: (`I` $= m_1$, `J` $= m_2 - 1$), (`I` $= m_1$, `J` $= m_2$), and (`I` $= m_1$, `J` $= m_2 + 1$). Iteration (`I` $= m_1$, `J` $= m_2$) also assigns to that element. Therefore, the `J` loop iterations must execute in the correct order for a fixed value of `I` to avoid overwriting the element while the "old" value is still needed. However, the `I` loop iterations never interfere with each other and

therefore can execute in any order, including overlapped execution. In other words, the `I` loop is vectorizable while the `J` loop is not. Dependence analysis formalizes this test.

Returning to our example above, we see that there are two dependences from the assignment to itself—a dependence from the store to a use of `A(I,J-1)` on the next iteration and an *antidependence* (needed to ensure that loads do not move before stores) from the use of `A(I,J+1)` to the store into `A(I,J)` on the next iteration. These two dependences arise from the iteration of the loop on index `J`, so we say that they are *carried* by that loop. In terms of dependence, the test for vectorization can be stated as follows: A statement can be vectorized with respect to a given loop if that statement is not part of a dependence cycle carried by that loop. Hence, we see in the example above that the `J`-loop is not vectorizable, but that the `I`-loop is. The code can therefore be rewritten in Fortran 90 as:

```
REAL A(1000,1000)
DO J = 2, N
  A(2:N,J) = (A(2:N,J+1)+ 2*A(2:N,J) + A(2:N,J-1))*0.25
ENDDO
```

Vectorizers based on dependence analysis matured into extremely useful tools by the mid-1980s and came to be standard on all vector machines [6, 47]. Yet in spite of the sophistication of vectorizing compilers, it was still not possible to present a naively-coded Fortran program to any of them with the expectation of achieving high performance. Some subscripts simply cannot be fully checked by compile-time dependence analysis. In particular, references like `A(IND(I))` require runtime information not available to the compiler. Virtually every program had to be rewritten so that the computationally intensive loops were vectorizable. One can therefore characterize the contribution of vectorizing compilers as defining a subdialect of Fortran—the "vectorizable loop" subdialect—for which high performance would be achieved on virtually every vector machine.

Building on the success of vectorization, the research and development community turned its attention to automatic parallelization for asynchronous (MIMD) parallel processors with shared memory [4, 5, 39, 13, 46, 7, 47]. Beginning in the mid-1980s, such systems began to appear with 2 to 16 processors and, by the end of the decade, some systems with up to 32 processors became available. For configurations of modest size, the technology of automatic vectorization could be employed with good results. However, parallel computers soon moved to distributed memory architectures as described in Chapter **??**. This made the automatic parallelization problem far more complex, because the compiler now had the additional task of determining how to partition data to the memories of a processor in a way that maximized the number of local memory accesses and minimized communication, which was relatively expensive on such machines [8, 20, 21, 49]. Moreover, the compiler and runtime system had the task of arranging the communication operations themselves, which were far more complex than the simple loads and stores needed on shared memory machines.

Another important implication of the new architectures was that, even on shared memory parallel computers, the regions of parallel execution had to be large enough to compensate for the overhead of initiating and synchronization of the parallel computation. This led to research on how the compiler could find larger program regions to run in parallel. Dependence analysis, which worked so well for vectorization, now had to be applied over larger regions of the program, even across procedure boundaries. This led to research on *interprocedural analysis and optimization* by which a program and all its subroutines are analyzed as a whole [14, 15, 19, 23, 24, 31, 32, 44].

Through the use of increasingly complex analysis and optimization technologies, research compilers have been able to parallelize a number of interesting programs. However, due to the complexity of the techniques, the long compiler running times, and the small number of successful demonstrations, there exist few commercial compilers that attempt to parallelize whole applications on scalable parallel machines. Although this research has yielded many important new compilation techniques, it is now widely believed that automatic parallelization, by itself, is not enough to solve the parallel programming problem.

As a result of these observations, research has turned increasingly to language-based strategies that can get more information from the user while exploiting techniques from automatic parallelization to lessen the burden of programming.

## 12.2 Data-Parallel Languages, Exemplified by HPF

Early in the research efforts on parallel computing, Fox and others observed that the key to achieving high performance on distributed-memory machines was to allocate data to the various processor memories to maximize locality and minimize communication [28]. Once this is done, if each computation in a program is performed on the processor where most of the data involved in that computation resides, the program can be executed with high efficiency.

A second important observation was that if parallelism is to scale to hundreds or thousands of processors, *data parallelism* must be effectively exploited. Data parallelism is parallelism that derives from subdividing the (presumably large) data domain in some manner and assigning the subdomains to different processors. This strategy provides a natural fit with data layout, because the data layout falls naturally out of the division into subdomains.

These observations are the foundation for data-parallel languages, which provide mechanisms for supporting data parallelism, particularly through data layout. A number of such languages were developed in the late 1980s and early 1990s, including *Fortran D* [27, 36], *Vienna Fortran* [48, 22], *CM Fortran* [43], *C** [33], *data-parallel C*, and *PC++* [29]. These research efforts were the precursors of two informal standardization activities leading to *High Performance Fortran (HPF)* [35] and *High Performance C++ (HPC++)* [37].

The idea behind *High Performance Fortran*, an extended version of Fortran 90 generated by an informal standardization process in the early 1990s, is to automate most of the details of managing data. It accomplishes this goal by proving a set of directives that the user inserts to describe the data layout. The

compiler and run-time system translate these high-level directives into the complex low-level operations that actually communicate the data and synchronize processors when needed. An important quality of the layout directives is that they have no effect on the meaning of the program—they merely provide advice to the compiler on how to assign elements of the program arrays and other data structures to different processors for high performance. This layout specification is relatively machine-independent, so once it exists, the program can be tailored by the compiler to run on any of a variety of distributed-memory machines.

In HPF, the critical intellectual task for the programmer is to determine how data is to be laid out in the processor memories in the parallel machine configuration. We illustrate this by extending the example of the last section:

```
      REAL A(1000,1000), B(1000,1000)
      DO J = 2, N
        DO I = 2, N
          A(I,J) = (A(I,J+1)+ 2*A(I,J) + A(I,J-1))*0.25 &
&                  + (B(I+1,J)+ 2*B(I,J) + B(I-1,J))*0.25
        ENDDO
      ENDDO
```

HPF provides fairly fine-grained control over data layout of arrays through directives, encoded as structured comments. The `DISTRIBUTE` directive specifies how to partition a data array onto the memories of a real parallel machine. In this case, it is most natural to distribute the first dimension, since iterations over it can be performed in parallel. For example, the programmer can distribute data in contiguous chunks across the available processors by inserting the directive

```
!HPF$ DISTRIBUTE A(BLOCK,*)
```

after the declaration of A. HPF also provides other standard distribution patterns, including `CYCLIC` in which elements are assigned to processors in round-robin fashion, or `CYCLIC(K)` by which blocks of `K` elements are assigned round-robin to processors. Generally speaking, `BLOCK` is the preferred distribution for computations with nearest-neighbor elementwise communication while the `CYCLIC` variants allow finer load balancing of some computations. Also, in many computations (including the example above), different data arrays should use the same or related data layouts. The `ALIGN` directive specifies an element-wise matching between arrays in these cases. For example, to give array `B` the same distribution as `A`, the programmer would use the directive

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

Integer linear functions of the subscripts are also allowed in `ALIGN`, and are useful for matching arrays of different shapes.[1]

[1] In fact, other alignments would produce better performance for our example. However, we use the direct alignment above to illustrate points about communication later.

In addition to the distribution directives, HPF has special directives that can be used to assist in the identification of parallelism. Because HPF is based on Fortran 90, it also has array operations to express elementwise parallelism directly. These operations are particularly appropriate when applied to a distributed dimension, in which case the compiler can (relatively) easily manage the synchronization and data movement together. Using array notation in this example produces the following:

```
      REAL A(1000,1000)
      !HPF$ DISTRIBUTE A(BLOCK,*)
      !HPF$ ALIGN B(I,J) WITH A(I,J)
      DO J = 2, N
        A(2:N,J) = (A(2:N,J+1)+ 2*A(2:N,J) + A(2:N,J-1))*0.25 &
&                  + (B(3:N+1,J)+ 2*B(2:N,J) + B(1:N-1,J))*0.25
      ENDDO
```

Alternately, the programmer could retain the loop notation but explicitly identify the inner loop as parallel. The `INDEPENDENT` directive specifies that the loop that follows is safe to execute in parallel. In our example, this appears as

```
      REAL A(1000,1000)
      !HPF$ DISTRIBUTE A(BLOCK,*)
      !HPF$ ALIGN B(I,J) WITH A(I,J)
      DO J = 2, N
        !HPF$ INDEPENDENT
        DO I = 2, N
          A(I,J) = (A(I,J+1)+ 2*A(I,J) + A(I,J-1))*0.25 &
&                  + (B(I+1,J)+ 2*B(I,J) + B(I-1,J))*0.25
        ENDDO
      ENDDO
```

Many compilers can detect this fact for themselves using the dependence analysis discussed in Section 12.1. However, the directive ensures that all compilers to which the program is presented can do so. The `INDEPENDENT` directive is even more important for loops that are theoretically unanalyzable; often the programmer will have application-specific knowledge that allows the loop to be executed in parallel.

Using either of the above notations (or relying on the compiler dependence analysis) puts the burden of efficiently executing the loop on the HPF implementation. A typical implementation would distribute the computations in loop iterations according to the *owner-computes* rule, by which the processor owning the array element on the left hand side of the assignment statement would perform the computation for each iteration. In the above example, if there are 25 processors, the first processor would handle iterations 2 through 40, the second would handle 41 through 80, and so on. These calculations would be done completely in parallel. Note however, that the references to `B(I-1,J)` and `B(I+1,J)` give rise to communication when I is equal to $40k$ and $40k$+1 respectively. The compiler will generate this communication automatically, and would

package the communication to optimize performance. On distributed memory machines, this packaging would generally consist of sending all required values of B before the start of the loop body, thus avoiding repeated message start-ups.

The HPF compiler must often go to substantial lengths to preserve the meaning of the underlying Fortran 90 program. For example, we might code a sum reduction loop as:

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X = 0.0
DO I = 1, 10000
   X = X + A(I)
ENDDO
```

Although this is much simpler than the equivalent message-passing program written in MPI, it has a downside—the compiler must do a substantial amount of work to generate a program that displays reasonable efficiency. In particular, it must recognize that the main calculation is a sum reduction and replicate the values of X on each processor. Then it must generate the final parallel sum at the end. HPF provides directives that make it possible for the user to help the system recognize such opportunities.

In the example above, the usual INDEPENDENT directive would not be applicable because the repeated assignments to X create a data dependence. However, because reduction is a common operation with special properties that allow parallelization, HPF provides an additional clause for the directive to handle it:

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X = 0.0
!HPF$ INDEPENDENT, REDUCTION(X)
DO I = 1, 10000
   X = X + A(I)
ENDDO
```

This version is easier for the compiler to process into an efficient program. We note in passing that there is also a standard intrinsic function available for this reduction:

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X = SUM(A)
```

The compiler can implement this as a library function or by expanding the sum in-line. In either case, the generated code will operate as described above.

As a final example, we present a simple HPF code fragment that is intended model parts of a multigrid method. All arrays are aligned to a "master", which is the coarsest grid level; that grid is distributed in both dimensions to get maximal locality. We use INDEPENDENT directives to ensure portability across

compilers that might not recognize the parallelism in the computation loops; we could equally well have used array syntax.

```
REAL A(1023,1023), B(1023,1023), APRIME(511,511)
!HPF$ ALIGN B(I,J) WITH A(I,J)
!HPF$ ALIGN APRIME(I,J) WITH A(2*I-1,2*J-1)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)

!HPF$ INDEPENDENT, NEW(I)
DO J = 2, 1022  ! Multigrid Smoothing (Red-Black)
   !HPF$   INDEPENDENT
   DO I = MOD(J,2), 1022, 2
      A(I,J) = 0.25*(A(I+1,J) + A(I-1,J) + &
               A(I,J-1) + A(I,J+1)) + B(I,J)
   END DO
END DO

!HPF$ INDEPENDENT, NEW(I), REDUCTION(ERR)
DO J = 2, 510   ! Multigrid Restriction
   !HPF$   INDEPENDENT
   DO I = 2, 510
      APRIME(I,J) = 0.05*(A(2*I-2,2*J-2) + &
         4*A(2*I-2,2*J-1) + A(2*I-2,2*J) + &
         4*A(2*I-1,2*J-2) + 4*A(2*I-1,2*J) + &
         A(2*I,2*J-2) + 4*A(2*I,2*J-1) + &
         A(2*I,2*J))
   END DO
END DO
```

In the example, the qualifier `NEW(I)` is used in the `INDEPENDENT` directive for the outer loop to ensure that the inner loop induction variable `I` is replicated on each group of processors that execute different iterations of the outer loop. This is roughly equivalent to the `PRIVATE` directive in other parallel dialects.

HPF compilation has been the subject of a substantial research and development [36, 49, 9, 16, 17, 18, 11, 30, 34, 45, 3]. Eleven companies currently offer HPF products and over thirty applications have been or are being written in it, including some over 100,000 lines.

The principal drawback of HPF is its limited support for problems defined on irregular meshes, which represent a fairly large fraction of the important science and engineering applications. To address this and other problems, the HPF Forum completed a second round of HPF standardization to produce HPF 2.0 [26], which includes important irregular distributions such as distribution indirectly via a run-time array and the generalized block distribution, which allows blocks to be of different sizes.

## 12.3 Shared-Memory Parallel Programming in OpenMP

Although HPF provides excellent facilities for specifying data distribution, its mechanisms for specifying explict parallelism are fairly limited. Principal among these is the `INDEPENDENT` directive discussed in the previous section. Control of the parallelism in HPF is *implicit*, in that the system assigns work to processors rather than the programmer. Moreover, the control is linked to the partitioning of data among processor memories. On machines where the entire system memory is shared among all processors, such implicit methods seem obscure. Furthermore, types of parallelism other than data parallelism are often profitable on such machines, but not well-supported by HPF.

Although a number of machine vendors produced mechanisms for explicit specification of parallelism in the late 1980s, there was no widely-accepted parallel language standard for shared-memory parallel machines. To address this deficiency, the Parallel Computer Forum began an open standardization process that led to the definition of PCF Fortran [42] and eventually to the ANSI abstract interface standard X3H5 [10]. PCF Fortran combined the facilities of two programming models: loop parallelism and single-program multiple-data (SPMD) parallelism. In SPMD parallelism, as exemplified by early efforts such as The Force [38] and IBM's VM/EPEX [25], all the processors or threads available to the program execute the entire program. This execution is redundant until the threads encounter a work-distribution directive, such as a parallel loop, whereupon the work is divided among the available threads. At the end of the work-distribution construct the threads execute an implied barrier and then continue redundant execution. One advantage of this strategy is that each thread builds up its own replicated copy of the program state in local memory, enhancing the locality of references in the code. PCF Fortran included a feature called *parallel regions*, which were constructs within which SPMD execution could take place. Standalone parallel loops were also included in the specification as a shorthand for a parallel region that exactly brackets a work-distribution loop.

The PCF/X3H5 standard lay fallow until 1997, when an industry consortium led by Silicon Graphics refined and simplified these ideas to produce OpenMP,an informal standard parallel programming interface with bindings to Fortran 77 and C. The consortium is now considering additions to the OpenMP standard, including bindings for Fortran 95 and C++. OpenMP drew strongly on the ideas from PCF Fortran and it adopted the directive conventions pioneered by HPF to specify parallelism in the program. As in HPF, OpenMP directives in a standard-conforming program can be ignored as comments by a uniprocessor compiler with no difference in results. In this section, we focus on OpenMP because it is the most recent and widely-used of these systems; however, many of the technical ideas were common to the PCF Fortran and ANSI X3H5 dialects.

Perhaps the simplest way to specify parallelism in OpenMP is via an explicitly parallel loop, bracketed by the `PARALLEL DO` and the `END PARALLEL DO` directives. The `PARALLEL DO` directive can have a number of qualifying clauses that permit the specification of variables that are private to threads executing invidivual loop iterations, and variables that are used in a reduction. The

following simple example of a `PARALLEL DO` loop computes a simple relaxation step:

```
!$OMP PARALLEL DO
DO I = 2, N
   APRIME(I) = (A(I+1) +2*A(I) + A(I-1))*0.25
ENDDO
!$OMP END PARALLEL DO
```

Note here that the loop induction variable `I` is private by default. The `END PARALLEL DO` directive is optional. OpenMP provides mechanisms for specifying how the iterations of a parallel loop are to be assigned to threads within a team. The following variant will assign contiguous blocks of iterations to a single thread at compile time.

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
DO I = 2, N
   APRIME(I) = (A(I+1) +2*A(I) + A(I-1))*0.25
ENDDO
!$OMP END PARALLEL DO
```

Under this specification, each thread would get a single contiguous block of iterations. This is roughly equivalent to the effect that would be achieved in HPF by declaring APRIME to have a BLOCK distribution. The effect of a `BLOCK(K)` distribution can be acheived by explicitly specifying a chunk size:

```
!$OMP PARALLEL DO SCHEDULE(STATIC,10)
DO I = 2, N
   APRIME(I) = (A(I+1) +2*A(I) + A(I-1))*0.25
ENDDO
!$OMP END PARALLEL DO
```

This loop will hand out chunks of ten iterations to threads in round-robin fashion in the order of the thread number. If the keyword `STATIC` is replaced by `DYNAMIC` in the above loop, chunks would be distributed to threads at run-time as those threads became ready to execute. OpenMP also permits `GUIDED` scheduling in which chunk sizes decrease as the remaining number of iterations decreases, and `RUNTIME` scheduling in which the scheduling and chunk size can be selected at run time by setting environment variables. This permits the algorithm to make dynamic choices based on conditions discovered in the data.

To illustrate the reduction mechanism, we present the global sum example from HPF, rewritten to use the OpenMP directives:

```
PROGRAM SUM
REAL A(10000)
READ (9) A
SUM = 0.0
!$OMP PARALLEL DO REDUCTION(+: SUM)
```

```
DO I = 1, 10000
   SUM = SUM + A(I)
ENDDO
PRINT SUM
END
```

This example is strikingly similar to its HPF counterpart. The `REDUCTION` clause specifies that the final value of variable `SUM` is determined by summing the final values in all of the threads executing iterations of the loop.

Task parallelism can be achieved in OpenMP through the `PARALLEL SECTIONS` directive. The following example illustrates its usage on a two-processor version of a routine to find the maximum of a set of numbers:

```
!$OMP PARALLEL SECTIONS SHARED(N,A) PRIVATE(I), LASTPRIVATE(MAX1,MAX2)
!$OMP SECTION
IF (N>=1) THEN MAX1 = 1 ELSE MAX1 = 0
DO I = 2, N/2
   IF(A(I)>A(MAX1)) THEN MAX1 = I
ENDDO
!$OMP SECTION
IF (N>=N/2+1) THEN MAX2 = N/2+1 ELSE MAX2 = 0
DO I = N/2+2,N
   IF(A(I)>A(MAX2)) THEN MAX2 = I
ENDDO
!$OMP END PARALLEL SECTIONS
IF (MAX1>0) THEN
   IF (A(MAX2)>A(MAX1)) THEN IMAX = MAX2 ELSE IMAX = MAX1
ELSE
   IMAX = 0
ENDIF
```

The `LASTPRIVATE` clause on the sections directive indicate that MAX1 and MAX2 are private to threads that execute the sections, but they retain their last value on exit from the parallel sections clause. Indeed these two are tested outside the region to determine which is the index of the larger value.

The core parallel contstruct in OpenMP is the *parallel region*, bracketed by the `PARALLEL` and `END PARALLEL` directives, which delimit a region for SPMD execution by multiple threads. Within a a parallel region, different work-sharing directives may appear. The `DO` directive specifies the extent of a DO-loop work sharing directive. The previously-discussed `PARALLEL DO` is simply shorthand for a parallel region with a `DO` directive nested within it. Similarly, the `SECTIONS` directive within a parallel region permits the specification of parallel tasks within the region. Note that each unit of work (loop iteration or section) within a work-sharing directive is executed by a single thread. Parallel regions may be nested inside one another but work-sharing directive may not, unless they are enclosed in an intervening parallel region directive. To illustrate these ideas, we will return to the multigrid code fragment discussed in the previous section.

```
REAL A(1023,1023), B(1023,1023), APRIME(511,511)

CALL OMP_SET_NESTED(.TRUE.)
!$OMP PARALLEL
!$OMP DO PRIVATE(J)
DO J = 2, 1022  ! Multigrid Smoothing (Red-Black)
   !$OMP PARALLEL PRIVATE(I)
   !$OMP DO
   DO I = MOD(J,2), 1022, 2
      A(I,J) = 0.25*(A(I+1,J) + A(I-1,J) + &
               A(I,J-1) + A(I,J+1)) + B(I,J)
   END DO
   !$OMP END PARALLEL
END DO

!$OMP DO PRIVATE(J)
DO J = 2, 510   ! Multigrid Restriction
   !$OMP PARALLEL PRIVATE(I)
   !$OMP DO
   DO I = 2, 510
      APRIME(I,J) = 0.05*(A(2*I-2,2*J-2) + &
         4*A(2*I-2,2*J-1) + A(2*I-2,2*J) + &
         4*A(2*I-1,2*J-2) + 4*A(2*I-1,2*J) + &
         A(2*I,2*J-2) + 4*A(2*I,2*J-1) + &
         A(2*I,2*J))
   END DO
   !$OMP END PARALLEL
END DO
!$OMP END PARALLEL

! Multigrid convergence test
ERR = MAXVAL( ABS(A(:,:)-B(:,:)) )
```

Note that there is an implied barrier between the first and second do loop nests in the outer parallel region. If if this barrier is not desired, the user may attach a `NOWAIT` clause to the `END DO` OpenMP directive, in which case this directive is required. In the example above, the inner loops are nested within a nested parallel region, permitting them to be executed in parallel. The call to `OMP_SET_NESTED` before the initial parallel region permits parallelism to be used in both dimensions of the computation.

OpenMP also provides lock variables to allow fine-grain synchronization between threads. We illustrate this by an example of parallelizing a simple relaxation code using wavefront parallelism. The scalar computation looks like this:

```
PARAMETER (N  = 2048) ! Total number of elements
REAL A(N,N)
```

```
DO J = 2, N-1
   DO I = 2, N-1
      A(I,J) = 0.25*(A(I+1,J) + A(I-1,J) + A(I,J-1) + A(I,J+1))
   ENDDO
ENDDO
```

A simple `PARALLEL DO` cannot be used in this case, because both the `J` and `I` loops carry data dependences. However, the computation can be partially parallelized in pipeline fashion as follows. Partition each column (by blocks) among the threads. At the beginning of each processor's section of the colum, force the calculation of `A(I,J)` to wait until the thread computing `A(I-1,J)` finishes that calculation so that it can get the correct input value. Lock variables do exactly this type of waiting; only one thread may hold a lock at any given time, forcing others to delay until it is finished. A two-dimensional array of OpenMP locks can therefore handle the synchronization as follows:

```
PARAMETER (NP = 8)      ! Number of processors
PARAMETER (NEP = 256)   ! Number of elements per processor
PARAMETER (N  = NP*NEP) ! Total number of elements
REAL A(N,N)
INTEGER LCK(NP,N)

!$OMP PARALLEL SHARED(A,LCK) PRIVATE(ME,JLO,JHI,I,J)
ME = OMP_GET_THREAD_NUM()+1   ! This thread's id
ILO = MAX( 2, (ME-1)*NEP+1 )  ! This thread's starting point
IHI = MIN( N-1, ME*NEP )      ! This thread's ending point

! Initialize the locks
DO J = 2, N-1
   CALL OMP_INIT_LOCK( LCK(ME,J) ) ! Leaves lock unset
   IF (ME>1) CALL OMP_SET_LOCK( LCK(ME,J) )
ENDDO
! Make sure other threads have done their initialization
!$OMP BARRIER

! Execute this thread's portion of the loop nest
DO J = 2, N-1
   IF (ME>1) THEN
      ! Wait to acquire lock, then go forward
      CALL OMP_SET_LOCK(LCK(ME-1,J)) ! Waits for lock unset
      CALL OMP_UNSET_LOCK(LCK(ME-1,J))
   ENDIF
   DO I = ILO, IHI
      A(I,J) = 0.25*(A(I+1,J) + A(I-1,J) + A(I,J-1) + A(I,J+1))
   ENDDO
   CALL OMP_UNSET_LOCK( LCK(ME,J) )
```

```
ENDDO

!$OMP END PARALLEL
```

We should note that this code is likely to be impractical on most implementations because of the overhead in time and space of managing so many locks. We can reduce the number of locks by synchronizing groups of columns, rather than one at a time. This sacrifices some parallelism (by delaying the start of the pipeline on some processors) in exchange for reducing the overall overhead. The optimal number of columns to group in this way will depend on the parameters of the machine, but the outline of the blocked code would always be similar to the following:

```
PARAMETER (NP = 8)         ! Number of processors
PARAMETER (NB = 16)        ! Number of blocks
PARAMETER (NEB = 16)       ! Number of elements per block
PARAMETER (NEP = NB*NEB)   ! Number of elements per processor
PARAMETER (N  = NP*NEP)    ! Total number of elements
REAL A(N,N)
INTEGER LCK(NP,NB)

!$OMP PARALLEL SHARED(A,LCK) PRIVATE(ME,JLO,JHI,I,J,JJ,JLO,JHI)
ME = OMP_GET_THREAD_NUM()+1   ! This thread's id
ILO = MAX( 2, (ME-1)*NEP+1 )
IHI = MIN( N-1, ME*NEP )

! Initialize the locks
DO JJ = 1, NB
   CALL OMP_INIT_LOCK( LCK(ME,JJ) )
   IF (ME>1) CALL OMP_SET_LOCK( LCK(ME,JJ) )
ENDDO
!$OMP BARRIER

! Execute this thread's portion of the loop nest
DO JJ = 1, NB
  JLO = MAX( 2, (JJ-1)*NEB+1 )
  JHI = MIN( N-1, JJ*NEB )
  IF (ME>1) THEN
      ! Wait to acquire lock, then go forward
      CALL OMP_SET_LOCK(LCK(ME-1,JJ))
      CALL OMP_UNSET_LOCK(LCK(ME-1,JJ))
  ENDIF
  DO J = JLO, JHI
    DO I = ILO, IHI
       A(I,J) = 0.25*(A(I+1,J) + A(I-1,J) + A(I,J-1) + A(I,J+1))
    ENDDO
  END DO
```

```
      CALL OMP_UNSET_LOCK( LCK(ME,JJ) )
    ENDDO

    !$OMP END PARALLEL
```

OpenMP is an excellent programming interface for uniform-access shared-memory machines. However, it provides the user with no way to specify locality in machines with non-uniform shared memory or distributed memory. On clusters of multiprocessor workstations, it is often used in conjunction with MPI, with OpenMP used for nodes and MPI used for message-passing between nodes. In the long term, a mixture of OpenMP and HPF directives seems a promising way to provide programming support for modern machines with a mixture of shared-memory and distributed-memory parallelism.

## 12.4 Supporting Technologies

To support the goal of making it possible to provide the user with a high level of abstraction without denying the opportunity to have fine-grained control over performance, the programming system must provide mechanisms for understanding the performance of applications and for overcoming any bottlenecks that are discovered in the tuning process. In addition, certain functions that are used over and over again in parallel programs need to be pre-tuned for execution on each parallel platform. This requires certain component technologies be developed along with the language compilers. Two of these—tools and tuned libraries—are of critical importance to the succes of a new languages.

*Programming Support Tools* All of the strategies envisioned for application development establish a complex relationship between the source version of the program and the version that runs on the computational grid. Science and engineering users need to have ways to understand performance of a given program and to tune it when it is unacceptable. Furthermore, the explanation of program behavior must be presented in terms of the source rather than the object version. Otherwise, the advantages provided by language abstraction will be lost. This becomes particularly challenging when some of the compilation process is done at run time.

The HPF experience has established that the compiler must generate two things to support performance analysis and tuning [2]: (1) calls to the performance monitoring system at critical points, where what is "critical" must be decided by some combination of user and system; and (2) information on how to map performance information back to the source of the program when it becomes available after execution.

In addition, the compiler and language must provide mechanisms that permit the program performance to be improved once the bottlenecks have been identified. These performance-improving changes must typically be made in the program source, so they will be preserved for the next run. Thus the tools must understand the relationship between the structure of the program and typical performance problems and they must be able to make transformations based on

that understanding.

*Libraries* There are many functions that are common in parallel programming yet difficult to implement efficiently for different platforms without hand tuning. It has been common to encapsulate these functions in programming support libraries. HPF was one of the first languages to specify an extensive library as a part of the langauge and Java has followed suit with a large collection of special-purpose library interfaces. The advantage of a library is that it can be hand tuned to achieve optimal performance on each target platform. The disadvantage is that, without such tuning, performance is likely to suffer. One of the most important impediments to widespread acceptance of HPF has been problems with the library implementations. This has been compounded by the absence of a well-developed, portable math library such as the CMSSL, which was developed for the Connection Machine. To be truly useful, all of the standard libraries must be capable of accepting the data types provided in the language—scientific programmers expect no less.

One area of importance in the future will be methodologies for development of libraries that can be easily and efficiently integrated into applications via transformations in the compiler. This will be discussed further in the next section.

## 12.5 Future Trends

As of this publication, the reseach community and commercial vendors have been actively working on programming support for parallel computer systems for over fifteen years, yet the improvements in ease of programming have been reasonably modest. In our opinion, this has been because designers have been exclusively focused on making it possible to write parallel programs that can be ported to a variety of parallel computing platforms. With technologies such as MPI, PVM, Pthreads, HPF, OpenMP, and Java, these problems have been well addressed. However, as new parallel computing platforms emerge they will bring new challenges for compiler developers.

Over the first decade of the next century, we see two major challenges for research on programming systems:

- *Programming Support for the Computational Grid.* There is great emerging interest in using the global information infrastructure as a computing platform. By drawing on the power of high performance computing resources across the world, it may be possible to solve problems that cannot currenltly be attacked by any single computer system, parallel or otherwise. However, the so-called Computational Power Grid, or *Grid* for short, presents nightmarish problems for the application developer because of the dynamic nature of the underlying computing and communications resources. The critical issues are how to build applications that are tolerant of the changes in resource base and how to construct execution environments that can deliver reliable progress on a given application. One strategy being pursued by a number of CRPC researchers in the *GrADS*

(Grid Application Development Software) Project is to implement an execution environment that constantly monitors progress of an application and automatically reconfigures it whenever performance falls below certain specifications. To implement this strategy, compilers and libraries will need to developed with the notion of reconfigurability built in from the outset.

- *Problem-Solving Environments and High-Level Programming Systems.* Programming for parallel execution environments is still clearly an expert's game. If parallel computing is to ever become more widely used, we need ways to make it easier for end users to develop programs. One strategy that promises to become more prominent over the next few years is the use of sophisticated problem-solving environments (PSEs). In such environments, domain-specific macro operations could be encapsulated as language primitives and programs written in high-level easy-to-use scripts. Visual Basic, Matlab, and data base query languages are three examples of such systems. If sript-based PSEs are to be used for applications where efficiency and performance are critical factors, the compilation systems will need to be able to automatically integrate the macro operations with scripts and translate the resulting global program to make effective use of scalable parallelism.

If substantive progress is made on these two issues over the next decade, it should be possible to come much closer to the dream of making the collection of networked computers into a problem solving system for ordinary users, much as the internet has become the common man's information system. Such a goal is worthy of a major national effort.

## 12.6 Summary

Parallel computation is a challenging activity because, at the lowest level, the application developer must discover parallel work and coordinate the activities of multiple processors carrying it out. The goal of high-level language and compiler strategies is to make this job easier by doing as much as possible for the user. In this chapter we have described three key technologies developed over the past 15 years for support of high-level parallel programming:

1. *Automatic Parallelization*, in which the compiler translates a sequential program to a parallel one. Although this is the ideal strategy from the point of view of the end user, it has not been successful in achieving acceptable degrees of scalability. Nevertheless, the techniques of automatic parallelization are fundamental to the support of most other high-level strategies.

2. *Data Parallel Languages* as exemplified by High Performance Fortran. Data parallel languages support a style of parallelism derived from decomposing array data structures across the processors of distributed memory

machines. High Performance Fortran, provides a set of data decomposition directives that serve as hints to the compiler on how to achieve high locality and implicit parallelism on such systems.

3. *Shared Memory Parallel Programming Interfaces* as exemplified by OpenMP. Shared-memory parallelism is primarily concerned with work decomposition, since the ideal target systems for such interfaces have a uniform-access shared global memory. OpenMP is the most prominent example of such systems. It uses a system of directives that specify where multiple threads should be applied and how to assign work to those threads.

These three strategies represent the most promising results of the research conducted by the community on support for parallel computing over the lifetime of CRPC. Although they represent fairly modest advances, we believe they have set the stage for much more dramatic improvements that will come in the near future.

### Further Reading

For more information on the topics covered in this chapter, the following works are recommended:

- The book *Parallel Computing Works!* [28] by Fox, Williams, and Messina, which compiles an enormous amount of information about parallel computation, particularly in the early days of distributed memory machines.
- The book *High Performance Compilers for Parallel Computing* [47] by Wolfe, which covers most of the vectorization and parallelization subjects.
- The Springer-Verlag book *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems* [41], edited by Santosh Pande and Dharma P. Agrawal, which contains a collection of articles on compiling for modern parallel machines.
- The Morgan-Kaufmann book *Advanced Compiling for High Performance* by Kennedy, which provides an in-depth coverage of automatic methods of vectorization, parallelization, and management of memory hierarchies.
- The survey article *Interprocedural Analysis and Optimization* [23] by Cooper, Hall, Kennedy, and Torczon, which provides a fairly comprehensive overview of whole-program compilation technologies.
- The article *Requirements for Data-Parallel Programming Environments* [1], by Adve et. al., which gives an overview of considerations in designing programming tools that are integrated with the language compiler system.

## 12.7 Acknowledgments

# *Bibliography*


[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C. Tseng. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[2] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[3] Vikram Adve and John Mellor-Crummey. Advanced code generation for High Performance Fortran. In Santosh Pande and Dharma P. Agrawal, editors, *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

[4] F. E. Allen, M. Burke, P. Charles, and R. Cytron. An overview of the PTRAN analysis system for multiprocessing. *Parallel and Distributed Computing*, 5:617–640, 1988.

[5] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[6] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[7] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[8] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[9] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[10] ANSI Working Committee X3H5. *Parallel Processing Model for High Level Programming Languages*, April 1994.

[11] Applied Parallel Research, Sacramento, CA. *Forge High Performance Fortran xhpf User's Guide*, version 2.1 edition, 1995.

[12] J. Backus. The history of FORTRAN i, II and III. *ACM SIGPLAN Notices*, 13(8):165–180, August 1978.

[13] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[14] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1979.

[15] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[16] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[17] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[18] T. Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Working Conference on Massively Parallel Programming Models*, Berlin, 1993.

[19] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[20] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[21] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic data layout for distributed-memory machines in the D programming environment. In *Proceedings of AP'93 International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Saarbrücken, Germany, March 1993.

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

[23] K. D. Cooper, M. W. Hall, K. Kennedy, and L. Torczon. Interprocedural analysis and optimization. *Communications in Pure and Applied Mathematics*, 48:947–1003, 1995.

[24] K. D. Cooper and K. Kennedy. Interprocedural side effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[25] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. VM/EPEX – a VM environment for parallel execution. Technical Report RC11225(#49161), IBM T. J. Watson Research Center, Yorktown Heights, NY, January 1985.

[26] High Performance Fortran Forum. High Performance Fortran language specification. Technical Report CRPC-TR92225, Center for Resarch on Parallel Computation, Rice University, January 1997.

[27] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. The fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[28] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kauffman, San Francisco, CA, 1994.

[29] D. Gannon et al. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing '93*, November 1993.

[30] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[31] M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland,OR, August 1993.

[32] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural parallelization analysis: A case study. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[33] L. Hamel, P. Hatcher, and M. Quinn. An optimizing C* compiler for a hypercube multicomputer. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[34] J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal of Digital Equipment Corp.*, 7(3):5–23, Fall 1995.

[35] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[36] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[37] E. Johnson, D. Gannon, and P. Beckman. HPC++: experiments with the Parallel Standard Template Library. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[38] Harry F. Jordan. The Force. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[39] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.

[40] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[41] Santosh Pande and Dharma P. Agrawal. *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*. Springer-Verlag, 1997.

[42] Parallel Computing Forum. PCF: Parallel Fortran extensions. *Fortran Forum*, 10(3), September 1991.

[43] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 1.0 edition, February 1991.

[44] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[45] Kees van Reeuwijk, Will Denissen, Henk Sips, and Edwin Paalvast. An implementation framework for hpf distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):897–914, September 1996.

[46] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[47] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

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

[49] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.