The Fortran-P Translator: Towards Automatic Translation of Fortran 77 Programs for Massively Parallel Processors

MATTHEW O'KEEFE, TERENCE PARR, B. KEVIN EDGAR, STEVE ANDERSON, PAUL WOODWARD[1], AND HANK DIETZ[2]

[1]*Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN 55415*
[2]*School of Electrical Engineering, Purdue University, West Lafayette, IN 47907*

ABSTRACT

Massively parallel processors (MPPs) hold the promise of extremely high performance that, if realized, could be used to study problems of unprecedented size and complexity. One of the primary stumbling blocks to this promise has been the lack of tools to translate application codes to MPP form. In this article we show how applications codes written in a subset of Fortran 77, called Fortran-P, can be translated to achieve good performance on several massively parallel machines. This subset can express codes that are self-similar, where the algorithm applied to the global data domain is also applied to each subdomain. We have found many codes that match the Fortran-P programming style and have converted them using our tools. We believe a self-similar coding style will accomplish what a vectorizable style has accomplished for vector machines by allowing the construction of robust, user-friendly, automatic translation systems that increase programmer productivity and generate fast, efficient code for MPPs. © 1995 John Wiley & Sons, Inc.

1 INTRODUCTION

Distributed memory massively parallel processors (MPPs) consisting of many hundreds or even thousands of processors have become available and offer peak performance in the tens to hundreds of Gigaflops range [1, 2]. Achieving a significant fraction of this performance would allow the study of many important physical problems (compressible, turbulent flows [3], meso-scale weather prediction [4], ocean circulation, etc.) that would otherwise be intractable. Unfortunately, recent published results [5] have suggested that these performance levels are not yet sustainable on complete applications codes. The goal of the work described here is to achieve very high performance on MPPs while retaining portability and ease of programming for real applications codes.

The approach we propose is to exploit certain characteristics of application codes that map well to MPPs. These massively parallel applications are written in a subset of Fortran 77 we call Fortran-P and translated to a language suitable for MPP execution. There are many codes, including piecewise parabolic method [PPM; 6] and advanced regional prediction system [ARPS; 4], that naturally fit the definition of Fortran-P and can be

Received February 1994
Revised August 1994
e-mail: okeefe@ee.umn.edu

Scientific Programming, Vol. 4, pp. 1–21 (1995)
CCC 1058-9244/95/010001-21

directly translated. The Fortran-P language was derived from the style used by Woodward [6] in coding his hydrodynamics algorithms for a variety of high performance computers. (Woodward first developed several of these ideas when implementing PPM on a Cray using solid-state disks.) We have refined this language and implemented tools that automatically translate Fortran-P programs into massively parallel codes running on the CM-200, CM-5, and other parallel machines. These are complete codes used daily at our site to do large, state-of-the-art calculations.

In the Fortran-P approach, the programmer is responsible for writing correct code that matches the Fortran-P model. In our current implementation, this means that the programmer must use certain keywords and directives to guide the translation process, and hence our approach is not purely automatic. However, as our translation tools evolve and mature our ultimate goal is to reduce or remove altogether the need for these additional lexical constructs to direct the translation.

This article is organized as follows: We first describe why it can be difficult to sustain near-peak performance for MPPs even on highly parallel codes. We propose as a solution the Fortran-P model, where restricted forms of Fortran 77 code are translated to MPP form. The Fortran-P programming model and translation process are then presented in an informal way. In the final section the current Fortran-P translator implementations are described followed by performance results to date and conclusions.

2 PERFORMANCE OF MPPs

Currently, the high performance often advertised for MPPs is achieved primarily for small kernels and benchmark programs [7]. Recent performance studies comparing current MPPs to conventional vector supercomputers have shown that MPPs sustain a smaller fraction of their peak performance than vector machines [5]. This seems especially true for applications requiring frequent global or irregular communication [8], although this result is unsurprising given that the cost of fully connecting a large number of processing elements is prohibitive. However, even for highly parallel applications with local, regular communications current sustained MPP speeds, although often impressive, often do not approach achievable peak speeds. In this article, we focus on the problems attaining speedups on the highly parallel, regular applications that can be expressed in Fortran-P.

The computing environment at Minnesota includes two large MPP machines, the CM-200, CM-5,* and a variety of large Cray vector machines.† Our experience with the CM-200 and CM-5 running compact applications codes (5K to 10K lines) such as PPM [6, 9, 10] is that these machines have often been unable to sustain a significant fraction of the peak speed except for very large problem sizes. For smaller problem sizes various overheads, including communication, tend to reduce performance. Sustained performance is directly related to how long it takes a calculation to run. Just as important is the time necessary for porting the code from a vector supercomputer or workstation to an MPP. The current lack of tools for this arduous process has slowed the acceptance of MPP technology.

The poor sustained performance (relative to peak) we have observed on highly parallel, regular applications is due primarily to three factors:

1. Particular code idioms (or styles) are necessary to obtain best performance and deviations from these styles can degrade performance significantly (machine efficiency).
2. In some cases, the programming models and compilers offered by vendors do not match the model used in large applications codes (compiler efficiency).
3. It is currently difficult for programmers to modify whole codes to match the restricted idioms that provide the best performance (programmer efficiency).

The first two factors determine sustained machine performance; the third factor determines programmer performance (i.e., how many lines of efficient code can be written and debugged each day). In the following paragraphs we expand on each factor.

* The machine configurations at the University of Minnesota were, for the CM-200: 1024 processors, 4 Gbytes main memory, 7 Gflop/s peak speed; and for the CM-5: 544 nodes, 2176 processors, 17 Gbytes main memory, 65 Gflop/s peak speed with vector units.

† The University has a four-processor Cray X-MP, a four-processor, full memory (4 Gbytes main) Cray-2, and an eight-processor Cray Y-MP C90 with 4 Gbytes main memory.

2.1 Machine Efficiency

MPP performance sensitivity is due to the large number of efficiency-critical features [ECFs; 11] found in these machines: an ECF is any machine architecture or implementation feature that must be used efficiently to achieve good performance. Registers are a classic uniprocessor ECF: efficient use of registers can reduce program execution time significantly. (Good register allocation is a primary difference between "toy" and production quality compilers.) Other uniprocessor ECFs include cache management and pipeline scheduling, especially for deeply pipelined machines such as the DEC Alpha and MIPS/SGI R4000.

MPPs have the same ECFs as uniprocessors (which are, after all, the MPP building blocks) plus another class of ECFs, referred to as parallel ECFs, related to the interconnection network and distributed memory. Two key parallel ECFs are the network latency and bandwidth, which determine message start-up time and transfer rate. For programs that require frequent short bursts of communication between processors, network latency is most important; for infrequent communication with large messages network bandwidth is key. In most cases, both factors are important and they dictate whether a particular problem mapping to the machine will be efficient. In addition, network performance and processor performance should be balanced: faster processors require higher bandwidth, lower-latency networks. Speeding up one without the other is pointless (although in some cases slow networks can be compensated by larger processor memories).

2.2 Compiler Efficiency

Because there are more ECFs in MPPs than in uniprocessors, and because these ECFs interact in myriad and often unexpected ways, writing a good optimizing and parallelizing MPP compiler for a full language (such as Fortran 90 or a parallel C dialect) is a daunting task. MPP performance is often quite sensitive to program coding style. On the CM-200 at Minnesota, we have seen order-of-magnitude differences in performance between loops coded in two different (but equivalent) styles. The coding styles that achieve good performance are often not obvious, resulting from idiosyncrasies of the compiler, run-time libraries, and the machine itself. For example, the CM Fortran compiler for the CM-200 often has difficulty determining when local (and therefore cheap) communication is feasible; if it cannot, the compiler generates slow global router communications [12, 13]. (During our PPM performance studies on the CM-200, we quickly learned that global router communication often reduced performance by at least one and in some cases two orders of magnitude on certain loops.) The use of certain coding idioms can ensure that local (not global) communications are generated [14].

The interface between the microsequencer and front end for the CM-200 can also affect performance significantly: Best performance is achieved when long (but not too long) sequences of conformant array operations are executed on the microsequencer. Currently the compiler does not try to optimize the CM-200 code to improve microsequencer performance, so programmers who want this additional performance must hand-code large conformant blocks into their program.

Current compiler limitations resulting from basic assumptions about how the machine should be programmed can also reduce performance. For example, version 2.0 of the CM Fortran compiler would not parallelize across dimensions declared as local to a processor (:SERIAL).‡ We discuss the implications of this limitation in Section 4.

2.3 Programmer Efficiency

Programmer efficiency is greatly reduced when many manual program transformations are necessary both to port the code and to obtain performance. Often the porting process and performance-related transformations are so intricate and involved that it would be reckless for a programmer to attempt to implement them manually for even a small section of code, much less the whole program. Even if the whole program could be transformed, it would be difficult to read and maintain. In addition, these transformations are usually compiler or machine specific: Each machine would require a different coding style and hence a different version of the code. Maintaining consistency between these versions would be a very difficult task indeed.

If the code must be translated from a serial language such as Fortran 77 to a data parallel lan-

‡ This limitation was only recently removed in version 2.1 of the compiler [15].

guage such as CM Fortran then significant effort is involved in getting the code to run on the parallel machine. Additional parallel indices and data layout directives must be added, and this can be a tedious and highly error-prone process. Code must often be added to implement explicit communication (e.g., when updating fake zones); our experience has shown that this is often the most difficult code to write and debug. (The number of fake zones depends on the balance between network and processor speeds and hence should vary between machines to achieve best performance.) The resulting parallel program can often be less readable and more difficult to maintain and port.

2.4 Summary

From the previous discussion, it is clear that currently there is a "catch-22" in MPP performance: Excessive time is spent either optimizing and parallelizing codes (but codes run fast) or running the unoptimized slow code itself (but programming time is short). Once codes do run fast, I/O can become a serious bottleneck: I/O requirements for large fluid calculations are discussed by Woodward [16]; an approach to meeting these requirements is proposed by Avneson et al. [42].

Some of these performance problems will be eased as the systems software matures and the architectures become more stable, but difficulties in porting codes from workstations and vector supercomputers to distributed memory MPPs will remain. The fundamental issue is: What codes can exploit massive parallelism so that the MPP architecture actually does run fast? And for these codes, how can the programming be made simpler and less time-consuming?

In the following sections we describe how innovative translation tools can be applied to help solve the problem of programming MPPs for a class of important applications. This problem clearly requires innovations in compilers, architectures, and translation to achieve a complete solution.

3 FORTRAN-P: TRANSLATION TOOLS FOR SELF-SIMILAR APPLICATION CODES

Many large applications codes developed in "serial" languages such as Fortran 77 have been resistant to good vectorizing and parallelizing compiler technology simply because the underlying algorithms are inherently serial. There is a widespread and to some extent justified view that programmers will have to reconsider their basic algorithms for the new massively parallel environment. The result of this viewpoint has been a focus on language design rather than the translation of codes expressed in serial languages to MPP form [18]. The latter problem, especially the selection of an efficient data layout for a distributed memory machine, has in general been considered too difficult to automate and has been left to the programmer (see Gupta and Banerjee [19] for an exception).

We do not claim that all codes written in traditional languages like Fortran 77 can be automatically translated for efficient execution on an MPP. Rather, we argue that many programmers have, over the years, developed an understanding of how to express their parallel algorithms in serial programming languages. In fact, this is precisely what is done when writing in a "vectorizable" style for a vector machine. Over time compiler technology, in the form of program analysis and recognition of common programming idioms, has matured to the point that vectorization is highly automated. We believe that a similar approach using a self-similar programming style can allow efficient automatic parallelization of serial code for MPPs. The language target for the translator could be data-parallel with data layout primitives (such as CM Fortran [20] or HPF [18]), a message-passing model, or message-passing combined with a native code compiler for a node.

We have developed such a tool for a subset of Fortran 77 we call Fortran-P. (In fact, Fortran-P also includes several Fortran 90 array intrinsics.) Fortran-P codes are self-similar, allowing them to fully exploit the parallelism in an MPP, even as these machines scale up to ever larger sizes. Compilers can also exploit this property to generate efficient data layouts and communication code.

The Fortran-P translator allows the MPP environment to mirror the typical vector supercomputer work environment in which codes are written and maintained in Fortran 77 and then vectorized (and often parallelized) as an integral part of the compilation process. In the Fortran-P environment, the programmer develops and maintains codes on a workstation or a vector supercomputer and translates the program to MPP form only as a final step before running a large calculation where the enormous memory and computing capacity of an MPP are required. Programmers can take advantage of the mature and stable development environments available on these machines, debugging codes on smaller problems until the program is ready for a large production run on the MPP. This approach takes advantage of the portability of Fortran 77 across a variety of machines, from PCs to vector supercomputers.

Unlike other approaches [21], we automatically generate the data distribution given the self-similar property of Fortran-P programs. These earlier approaches have required that explicit data layout directives be added by the programmer; this is unnecessary in Fortran-P. In addition, we handle the overlap regions (what we call fake zones and what others often refer to as ghost zones) in a fundamentally different way than previous efforts that are based on the "owner computes" rule. This rule requires that the processor that "owns" a particular data item must be the only processor to compute and write to it.

We explicitly break the owner computes rule to increase performance; processors redundantly compute data "owned" by their neighbors to avoid communication where possible and useful. An additional benefit of this approach is that message-passing calls transferring boundary data need not appear after most loops in the MPP code generated by the translator, as in the work of Hiranandani et al. [21]; in fact, these calls can be collected and grouped into one routine called once per iteration (or time step for hyperbolic problems).

In the next two sections, we describe the Fortran-P programming model and the translation process.

3.1 Fortran-P Programming Model

Self-similar programs are written so that the computations applied to the whole domain are precisely the same as those applied to each subdomain. (This is analogous to the self-similarity of many fluid flows: Intuitively, self-similarity means that properties that are apparent globally are also found in small subregions of the flow; hence, if an observer could simultaneously examine both the complete domain and a small subdomain of the flow [without clues to the actual domain sizes] then he or she would be unable to tell them apart.) Programs written in Fortran-P perform computations over a logically homogeneous grid of zones where (in two dimensions) the data domain appears as shown in Figure 1. Here NX is the problem size in the x dimension, NY is the problem size in the y dimension, and nbdy is the number of

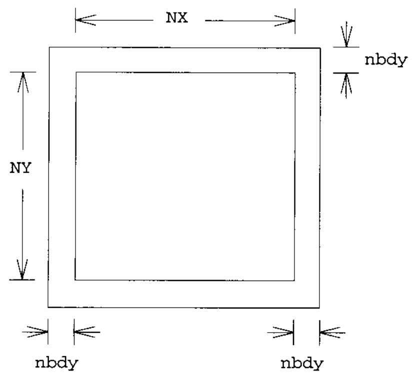

FIGURE 1 A two-dimensional data domain with surrounding boundary zone.

boundary (fake) zones. Boundary zones are employed to implement several different kinds of boundary conditions [6]. Note that the current translator also supports one- and three-dimensional grids (higher dimension grids are possible).

In general, a self-similar program has the following properties:

1. A logically regular computational grid. An irregular grid would not be self-similar as subdomains could have different grid topologies. In Fortran-P, each subdomain grid must have a topology identical to that of the global grid. This means that the loops range over the full extent of real zones (and partially over the fake zones) and that each zone is computed in a loop iteration just like every other zone.
2. Dependence distances [22] are small constants independent of the problem size, which implies local communication operations in the translated code. If dependence distances were a function of the grid size then when partitioning is performed nonlocal communication would be required. (There is an exception. For periodic boundary conditions the dependence distance is the equal to the extent of the particular dimension. Boundary zone handling is discussed in a following section.) In Fortran-P, updates of boundary zones imply that interior fake zones (to be described shortly) must be updated as well.
3. Each subdomain requires nearly the same amount of computation. The same algorithm is applied to equal-sized subdomains so, in general, the amount of computation will be nearly the same. Some variance will

arise from data-dependent operations but this should be balanced out when subdomains are relatively large.

4. Reduction operations are permitted, recurrences are not. Reductions can be implemented in two steps: First, locally within each processor, followed by a global reduction step using the result from the local computation. Recurrences are not self-similar in the sense that each result depends on a different number of operations and the calculation is inherently sequential.

These properties permit the data domain of self-similar Fortran-P programs to be automatically partitioned and then distributed equally across all processors. Each processor is given the same amount of work to perform. Fortran-P programmers write code so that operations that apply locally also apply globally. The translator can then exploit parallelism at multiple levels, both across the machine and within a node.

Assuming a 3 × 3 processor grid the two-dimensional domain is partitioned as shown in Figure 2. Note how each interior subdomain (or subpatch) is now surrounded by a picture frame of fake zones. These fake zones actually contain redundant data from neighboring processors as shown in Figure 3.

Fake zones (also referred to as overlap regions [23]) allow computations to be performed using only local data; as the algorithm proceeds, the fake zones become corrupted and eventually must be updated using real zones from neighboring processors. In current Fortran-P codes fake zone updates are done simultaneously with boundary zone computation. No interprocessor communication is generated for loops that range over real (interior) as well as fake zones. This allows the zones just inside the boundaries to be updated properly without any special boundary treatment, so long as the values in the fake zones are properly set.

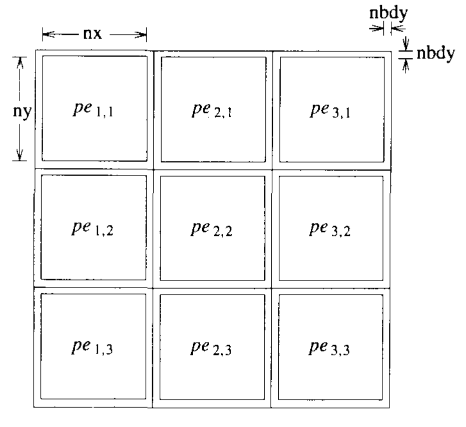

FIGURE 2 Partitioning a two-dimensional domain into 3 × 3 patches.

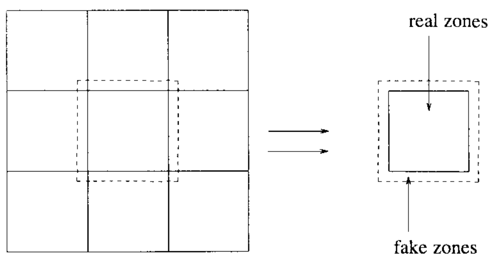

FIGURE 3 Fake zones for an interior subdomain.

Fake zones are indicated by the programmer when declaring arrays: Real zones extend from *l* to *n*; *nbdy* fake zones on either side are added, hence the full extent of the array becomes $-nbdy + l$ to $n + nbdy$. The symbol *n* will vary for each parallel dimension. (A template program for Fortran-P—*muscl16*—is available via anonymous ftp from `ftp.arc.umn.edu` in the /pub directory. This program shows how boundary handling can be written properly in Fortran-P.)

Another advantage of fake zones is that they allow the compiler to trade off memory with communication; larger overlap regions reduce the frequency of communication but require more memory. Hence, the compiler could vary the overlap region size to obtain best performance on a given machine. (For example, the CM-200 [24], which has four megabytes of memory per node and relatively slow network speed, would perform better with more fake zones than the Maspar MP-1 [25], which has only 64 Kbytes of memory per node but a relatively fast network.)

In the current Fortran-P model boundary code is written so that fake zone updates are generated when a loop is found that writes only the boundary array elements. Fortran-P currently recognizes four kinds of boundary conditions: reflecting, periodic, continuation, and prescribed value [6]. More general boundary conditions could easily be supported within this Fortran-P framework.

Global operations such as finding a global minimum or maximum are sometimes required in Fortran-P programs; e.g., in PPM, the Courant number must be calculated during each time step, which requires a global maximum. The current translator calls fast, vendor-supplied intrinsics (when available) to perform global operations efficiently. These kind of global reductions can most efficiently be performed in self-similar style: The reduction is performed locally on each processor and the same operation is then applied globally across processors using the local results.

Certain kinds of elliptic equations requiring global information can be addressed with multigrid relaxation schemes, which are made up entirely of local, self-similar operations applied in an iterative sequence. At each multigrid level the problem remains self-similar; refinements to the grid resolution can be incorporated naturally into the model. Hence Fortran-P can support these implicit difference schemes as well as the explicit schemes for which it is now used.

Many algorithms can take advantage of compress and decompress operations (available on certain vector machines directly in hardware) on vectors. For example, certain multifluid calculations requiring tracking of fluid interfaces [26] can be computed most efficiently this way. In the Fortran-P model, compress and decompress operations on subdomain data can be effectively performed on each local node. Although operations on the compressed data might in principle proceed faster if this work load were rebalanced, no such dynamic load balancing is currently implemented in our model.

We note that some popular algorithms violate the exclusive use of local data that characterize self-similar algorithms. Such algorithms include standard spectral methods and implicit methods for partial differential equations that employ standard linear algebraic techniques. These algorithms cannot be written in Fortran-P. However, as noted before implicit methods such as multigrid are self-similar. In addition, new implicit methods such as spectral elements, which use only local data within subdomains, are being developed to replace more traditional spectral techniques on MPPs.

3.2 The Translation Process

The Fortran-P translation process can be divided into two stages: (1) parallelization and data layout followed by (2) performance-enhancing optimizations. In the following sections we show only the current translation for the CM-200; the CM-5 translation is described briefly in Section 4 and in Appendix 1.

Parallelization and Data Layout

In this stage the code is parallelized for MPP execution. Data layout directives [20] are generated that partition the data domain equally among the processors. Integer arrays, floating point scalars, and floating point arrays are promoted in dimension and laid out across the processors; integer scalars are placed on the front end§ (CM-200 [1], Maspar MP-1 [25]). Current Fortran-P codes use integer scalars almost exclusively for index calculations and these scalars are stored on the front end. The arrays are promoted in dimension to map onto either a two or three-dimensional processor grid, as specified by the programmer (generally, two and three-dimensional data grids are mapped to two and three-dimensional processor grids). (The implementation can support other, more general, mappings; this will provide us with a mechanism to rapidly generate and benchmark several versions of the code with different data-to-processor grid mappings.)

As an example, the following subroutine *monslp* was taken from the code *muscl*15, a two-dimensional hydrodynamics code developed by Woodward [38] (Fig. 4). The original Fortran 77 version for this subroutine is converted by the translator to the code shown in Figure 5.

Note that the `CMF$ LAYOUT` directive can be used to specify the layout of each array dimension as either parallel (`:NEWS`, across processor nodes) or serial (`:SERIAL`, within a single node). Other MPP languages have similar directives [18, 25, 27]. This subroutine operates on one-dimensional arrays extracted from a two or three-dimensional subdomain. Note that the array extent `n` must be adjusted to reflect the size of the two-dimensional subdomain allocated to each node. The parameters `NODE_X` and `NODE_Y` indicate the extent of the processor grid in the x and y dimensions. As can be seen in the translated code additional parallel dimensions are added to each array reference to distribute subdomains among the processing elements.

Code to implement and update fake zones is added during this phase in connection with the handling of boundary conditions. During the development of the CM Fortran version of Fortran-P very close attention had to be paid to this portion of the code because it involved communication among nodes. For the CM-200, we were forced to use particular idioms and then check that the generated code used the mesh interconnect (referred to as news) rather than the global router [24], which was slow relative to the mesh interconnect.

Fake zones introduced by parallelization are set automatically by the Fortran-P translator

§ The front end is called the partition manager on the CM-5 [1].

```
      subroutine monslp (a, da, dal, dalfac, darfac, n)
      dimension   a(n), da(n), dal(n), dalfac(n), darfac(n)
      do 1000    i = 2,n
1000  dal(i) = a(i) - a(i-1)
      do 2000    i = 2,n-1
      dda = dalfac(i) * dal(i)  +  darfac(i) * dal(i+1)
      s = sign (1., dda)
      thyng = 2.  *  aminl (s * dal(i),  s * dal(i+1))
      da(i) = s  *  amaxl (0.,  aminl (s * dda, thyng))
2000  continue
      return
      end
```

FIGURE 4 *monslp* subroutine taken from *muscl15*.

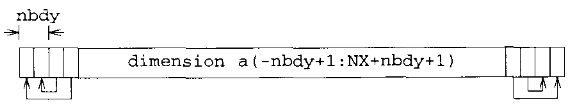

FIGURE 6 Fake zone updates for reflecting boundary conditions.

when the programmer specifies boundary setting in the original program. Only boundary fake zones on the edges of the processor array must be explicitly set by the programmer. For a reflecting boundary condition for a one-dimensional data array the Fortran-P program would be written as follows:

```
      dimension a(-nbdy+1 : NX+nbdy+1)
      do 10 i = -nbdy+1, 0
        a(i) = a(1-i)
10    continue
      do 20 i = 1, nbdy
        a(nx+i) = a(nx+1-i)
20    continue
```

The two boundary code segments set the left and right fake zones of the array a as shown in Figure 6. On a two-dimensional processor grid the Fortran-P code segments are transformed as follows:

```
      do 10 i = -nbdy+1, 0
       a(i, 1, :)= a(1-i, 1, :)
10    continue
      do 15 i = -nbdy+1, 0
       a(i, 2:NODE_x, :)
                = a(nx+i, 1:NODE_x-1, :)
15    continue
      do 20 i = 1, nbdy
       a(nx+i, NODE_X, :)
                  = a(nx+1-i, NODE_X, :)
20    continue
      do 25 i = 1, nbdy
       a(nx+i, 1:NODE_X-1, :)
                     = a(i, 2:NODE_X, :)
25    continue
```

Note that for version 2.1 of CM Fortran the translator generates Fortran 90 array sections instead of serial loops on the first dimension. The a(i, 1, :) = . . . reference sets the i^th element of

```
      subroutine monslp(a,da,dal,dalfac,darfac,n)
      parameter ( NODE_X = 16, NODE_Y = 16)
      dimension darfac( n, NODE_X, NODE_Y), dalfac( n, NODE_X, NODE_Y)
     &, dal( n, NODE_X, NODE_Y), da( n, NODE_X, NODE_Y), a( n, NODE_X,
     & NODE_Y)
CMF$ LAYOUT darfac(:SERIAL,:NEWS,:NEWS)
CMF$ LAYOUT dalfac(:SERIAL,:NEWS,:NEWS)
CMF$ LAYOUT dal(:SERIAL,:NEWS,:NEWS)
CMF$ LAYOUT da(:SERIAL,:NEWS,:NEWS)
CMF$ LAYOUT a(:SERIAL,:NEWS,:NEWS)
      dimension thyng(NODE_X, NODE_Y)
CMF$ LAYOUT thyng(:NEWS,:NEWS)
      dimension s(NODE_X, NODE_Y)
CMF$ LAYOUT s(:NEWS,:NEWS)
      dimension dda(NODE_X, NODE_Y)
CMF$ LAYOUT dda(:NEWS,:NEWS)
      do 1000 i =  2, n
1000  dal( i, :, :) = a( i, :, :) - a( i - 1, :, :)

      do 2000 i =  2, n - 1
      dda = dalfac( i, :, :) * dal( i, :, :) + darfac( i, :, :) * dal(
     & i + 1, :, :)
      s = sign( 1., dda)
      thyng = 2. * aminl( s * dal( i, :, :), s * dal( i + 1, :, :))
      da( i, :, :) = s * amaxl( 0., aminl( s * dda, thyng))
2000  continue

      return
      end
```

FIGURE 5 *monslp* subroutine after Fortran-P translation.

array a on the first column of processors. Because of the range of i, we are explicitly setting the fake zones on the left edge of the processor grid. Similarly, a(nx+i, NODE_X, :) = . . . sets the fake zones on the right edge of the processor grid. Loops labeled 15 and 25 are generated by the Fortran-P precompiler and dictate how the interior fake zones are to be updated with data from neighboring processors. These interior fake zone operations are merely data transfers from neighboring processors. For example, statement 20 ensures that references to fake zones on the left edge of a processor obtain data that are consistent with those on the left neighbor processor.

Assuming NODE_X=3, the loading of fake zones for the left edge, in one row of the grid, appears as shown in Figure 7, where the hatched faked zones are loaded by the code (appearing after the loop) inserted by the Fortran-P compiler. Similarly, the updating of fake zones on the right edge, in one row of the grid, appears as shown in Figure 8.

For a two-dimensional data array, columns of data are moved into fake zones rather than single array elements. For example, the following code would set the boundary zones on the left edge of the processor array b (assuming a periodic reflective boundary):

```
    do 30 i = -nbdy+1, 0
      do 30 j = -nbdy+1, ny+nbdy
      b(i,j) = b(i+nx,j)
30  continue
```

where the j loop iterates over an entire column of array b including the boundary zones on the top and bottom. In this case, the Fortran-P precompiler generates the following code:

```
    do 30 i = -nbdy+1, 0
      do 30 j = -nbdy+1, ny+nbdy
        b(i,j,1,:)
                      = b(i+nx,j,NODE_X,:)
30  continue
```

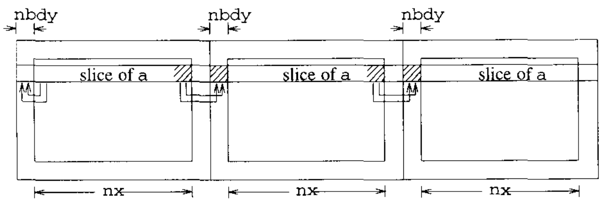

FIGURE 7 Fake and boundary zone updates for left side of grid.

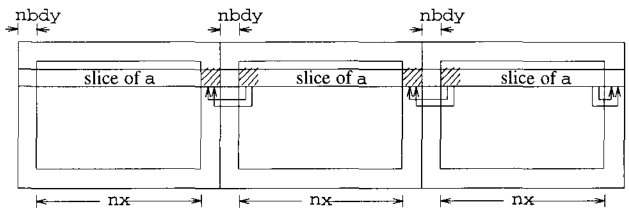

FIGURE 8 Fake and boundary zone updates for right side of grid.

```
    do 35 i = 1, nbdy
      b(nx+i, :, 1:NODE_X-1, :)
                    = b(i, :, 2:NODE_X, :)
35  continue
```

The Fortran-P programmer writes code to implement the periodic boundary condition and the data movement for the Fortran-P loop appears as shown in Figure 9. Notice that the boundary zones in the periodic case are being loaded from data residing on the opposite side of array b. After data layout and partitioning, these data reside on the opposite side of the processor grid in column NODE_X (columns are numbered 1 to NODE_X). As before, the statement following the loop is generated by the precompiler to update the interior fake zones as shown in Figure 10.

As mentioned previously, global reductions are done in a self-similar style. (Note that Fortran-P could easily support most of the Fortran 90 array intrinsic functions; currently we support all Fortran 77 intrinsics [excluding character-related functions]). The reduction operation is first performed on a subdomain, then the same reduction is applied globally across subdomains to obtain the scalar result. On the CM-200, code is gener-

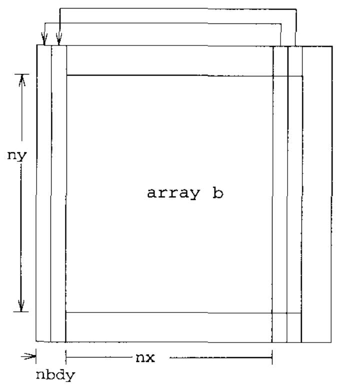

FIGURE 9 Periodic boundary condition, left edge of two-dimensional array b.

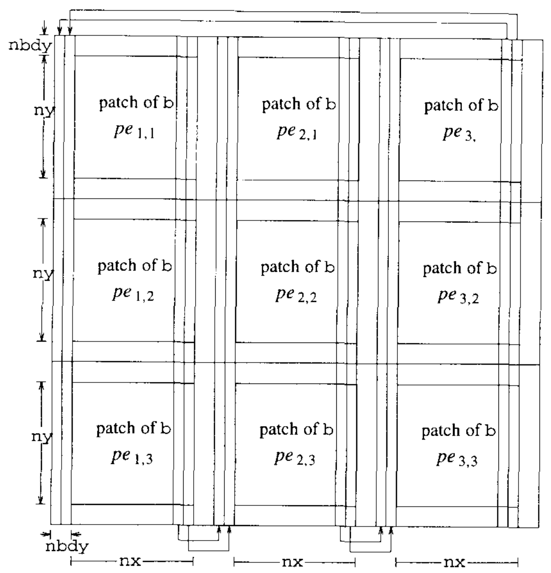

FIGURE 10 Fortran-P partitioning of two-dimensional data array.

ated to broadcast this result to all processors; for the CM-5, the scalar result is left on the control processor. As an example, PPM codes perform a Courant number calculation [6, 28] every time step which requires a global maximum operation. (In order for the numerical method to be stable, the Courant number, courno, must be less than one in every zone. Effectively, this means that the numerical time step is limited by the zone with the largest Courant number in the entire grid.) The Fortran-P code from *muscl15* uses the Fortran 90 intrinsic maxval and the statement appears as:

```
    courmx = maxval(courno(1:n))
```

The precompiler adds two processor dimensions to courno and courmx and scales the value n to spread them across the machine. The global maximum operation as generated by the Fortran-P precompiler (for the CM-200) is shown below. (Note that for the Fortran 90 intrinsic maxval (array, dim) an (optional) second argument indicates that the result array contains the maximum value along dimension dim of array; the rank of the result array is one less than array.)

```
c   find max value for each 2-D
                             subdomain
c
10  courmx = maxval(courno(1:n, :, :)
c   now find global max, broadcast
                           this result
c
20  courmx_global
                = maxval(courmx(:, :))
    courmx = courmx_global
```

This two-step process is unnecessary if the compiler performs it locally, then globally; it appears current and future versions of CM Fortran will support this. Other reduction operations expressed as Fortran 90 intrinsic array functions (ANY, ALL, COUNT, MAXVAL, MINVAL, PRODUCT) could be handled similarly.

The translation of elemental intrinsic functions in Fortran-P is transparent [20]. Recall that "for an elemental intrinsic function with one argument, calling the function with an array argument causes the function to be applied to each element of that array, with each application yielding a corresponding scalar result. This collection of result values is returned in the form of an array of the same shape as the argument" [29]. In other words, the elemental intrinsics are inherently self-similar. During Fortran-P translation, each elemental intrinsic is passed through unchanged, operating instead on all the elements in the promoted argument variable.

The current Fortran-P implementation recognizes the cvmg family of Cray Fortran intrinsics [30] for implementing merge operations (the semantics are similar to the C condition expression with the ternary operator "? :"). These cvmg intrinsics are converted to the Fortran 90 MERGE intrinsic, which is recognized by the CM Fortran compiler. Compress and decompress operations on vectors can be performed using the Fortran 90 PACK and UNPACK intrinsics, which could be translated to execute independently on each node.

Once all these transformations are completed, the code can be compiled and executed on an MPP. However, additional transformations, described next, can improve performance considerably.

PERFORMANCE-ENHANCING OPTIMIZATIONS

With immature, early-release compilers, restricted programming styles must often be used to achieve efficient execution, and this seems especially true on MPPs. In certain cases we have seen a two-order-of-magnitude difference in performance between two different but equivalent loop coding styles. For example, we found that in the boundary section code using Fortran 90 array notation, as in the following example

```
 rho( -nbdy+1:0, -nbdy+1:ny+nbdy,
                           1, :) =
&        rho( nx-nbdy+1:nx,
              -nbdy+1:ny+nbdy, NODE_X, :)
 p( -nbdy+1:0, -nbdy+1:ny+nbdy,
                           1, :) =
&        p( nx-nbdy+1:nx,
              -nbdy+1:ny+nbdy, NODE_X, :)
 ux( -nbdy+1:0, -nbdy+1:ny+nbdy,
                           1, :) =
&        ux( nx-nbdy+1:nx,
              -nbdy+1:ny+nbdy, NODE_X, :)
 uy( -nbdy+1:0, -nbdy+1:ny+nbdy,
                           1, :) =
&        uy( nx-nbdy+1:nx,
              -nbdy+1:ny+nbdy, NODE_X, :)
```

instead of tightly-nested DO loops

```
      do 200 i = - nbdy + 1, 0
      do 200 k = - nbdy + 1, ny + nbdy
      rho( i, k, 1, :) = rho( i + nx,
                         k, NODE_X, :)
      p( i, k, 1, :) = p( i + nx, k,
                         NODE_X, :)
      ux( i, k, 1, :) = ux( i + nx, k,
                         NODE_X, :)
      uy( i, k, 1, :) = uy( i + nx, k,
                         NODE_X, :)
200   continue
```

caused the compiler to generate general routing communication [14], slowing down the *muscl15* code by a factor of 100. The Fortran-P translator was modified to generate the faster loop form.

In developing the Fortran-P translator, many similar timing experiments were performed on different loops to determine source forms with the best performance. These forms have been incorporated into the CM Fortran code generated by the translator. This is an important advantage of automating the translation process; these kinds of source transformations can be extremely tedious and time-consuming to perform by hand.

More traditional compiler source transformations such as loop unrolling and forward substitution are also applied to improve performance. Forward substitution is applied to vectorizable

```
      do 9000   i = -j+6,n+j-4
      difusl = amaxl (difuse(i-1), difuse(i))
      dwoll = dt * difusl
      courno(i) = amaxl (courno(i), ((dwoll + dwoll)/
     &             aminl (dxnu(i-1), dxnu(i))))
      ddmll = dwoll * rhonu(i-1)
      ddmrl = dwoll * rhonu(i)
      dmassl(i) = ddmll - ddmrl
      dmomtl(i) = ddmll * utnu(i-1)  -  ddmrl * utnu(i)
      dmoml(i) = ddmll * unu(i-1)  -  ddmrl * unu(i)
      denl(i) = ddmll * enu(i-1)  -  ddmrl * enu(i)
9000  continue
```

FIGURE 11 Loop 9000 before forward substitution.

loops: Floating-point scalar temporaries are substituted directly into expressions, leaving only assignments into array references. For example, forward substitution on the loop taken from *muscl15* (Fig. 11) would result in the loop shown in Figure 12. In both the CM-5 and CM-200 [1, 24] the front end processors broadcast instructions, addresses, and other information as execution proceeds; generally, a packet of such information must be sent whenever the size and/or shape of the arrays being operated on changes, or when a transition from a scalar to array or array to scalar operation occurs [13]. This feature will likely be repeated on future MPPs with data-parallel compilers. Forward substitution results in larger sequences of conformant array operations and performance improvements on the order of 10–50% due to the significant reduction in front end transfer overheads. Loop unrolling can give a similar effect for smaller loop bodies.

On distributed memory MPPs it is important to reduce communication among processors to obtain good performance. Unfortunately, current compilers often generate unnecessary communication if they cannot determine that input or output data are local to a node [12, 13]. Compound-

```
      do 9000 i =  - j + 6, n + j - 4
      courno( i ) = amax1( courno( i ),(( dt *( amax1( difus
     &e( i - 1 ), difuse( i )))) +( dt *( amax1( difuse( i -
     & 1 ), difuse( i ))))) / amin1( dxnu( i - 1 ), dxn
     &u( i )))
      dmassl( i ) =(( dt *( amax1( difuse( i - 1 ), difuse(
     &i )))) * rhonu( i - 1 )) -(( dt *( amax1( difuse( i -
     &1 ), difuse( i )))) * rhonu( i ))
      dmomtl( i ) =(( dt *( amax1( difuse( i - 1 ), difuse(
     &i )))) * rhonu( i - 1 )) * utnu( i - 1 ) -(( dt *
     &( amax1( difuse( i - 1 ), difuse( i )))) * rhonu( i,
     &)) * utnu( i )
      dmoml( i ) =(( dt *( amax1( difuse( i - 1 ), difuse( i
     & )))) * rhonu( i - 1 )) * unu( i - 1 ) -(( dt *(
     &amax1( difuse( i - 1 ), difuse( i )))) * rhonu( i )
     &)) * unu( i )
      denl( i ) =(( dt *( amax1( difuse( i - 1 ), difuse( i,
     & )))) * rhonu( i - 1 )) * enu( i - 1 ) -(( dt *( a
     &max1( difuse( i - 1 ), difuse( i )))) * rhonu( i
     &)) * enu( i )
9000  continue
```

FIGURE 12 Loop 9000 after forward substitution.

ing this problem is that networks in current MPPs are often characterized by high latency and low bandwidths [5, 31] resulting in performance degradation if employed frequently.

We have worked closely with the Fortran compiler group at Thinking Machines to identify where this happens in typical Fortran-P codes; Thinking Machines has supplied us with several library routines that allow our Fortran-P translator to remove extraneous communication in the transformed CM Fortran codes. These routines include copy functions that copy data from one array into another without invoking unnecessary node communication (*vector_move_always*), as well as routines to equivalence arrays offset by small constants to avoid communication during the differencing operations common in Fortran-P codes. The *equiv_ld* routine also effectively removes the subgrid ratio problem described in the next section.

4 CURRENT FORTRAN-P IMPLEMENTATION AND RESULTS

The current Fortran-P translator (known as the alpha version) parses Fortran-P programs and generates intermediate representation (IR) trees; the back-end generates parallel versions of the code in CM Fortran [20]; different translations are generated for the CM-200 and the CM-5 because the underlying machine architectures and their interactions with the compiler are different. The CM-5 and CM-200 share some phases of the CM Fortran compiler but the final code generation phases are distinct, as are the run-time libraries [13].

The alpha version of the translator was implemented using the Purdue compiler construction tool set [PCCTS; 32]; future versions will be implemented with PCCTS and SORCERER [33, 34], a source-to-source translator generator. We can retarget our translators for Cray MPP Fortran [35] and Maspar Fortran [25]; a straightforward translation to these dialects involves changing the data layout directives to match those used in each dialect. In addition, we intend to support a message-passing implementation.

The Fortran-P translator has been employed to translate two PPM codes, *muscl15* and *hppmfair 14*, and the shallow water version of ARPS. The computational approach used by PPM and ARPS to exploit massively parallel processing is similar and these codes fit very naturally within the Fortran-P model, forming the core of our current Fortran-P applications suite. Both PPM and ARPS are inherently self-similar in design.

PPM codes have been used to study a variety of hydrodynamic phenomena. These codes use a logically regular grid and treat every grid zone alike wherever possible. The cost of this approach is some small smearing of important flow structures in such problems as multifluid calculations and flow around obstacles. Frequently, irregular, unstructured computational grids are used for these problems. However, given the large numerical grids possible on modern computers combined with the sophisticated shock and contact discontinuity capturing of PPM, these disadvantages are kept to a minimum. This shifts the difficulties of massively parallel computation from the dynamic data layout and load balancing to the design of the algorithm on a logically regular mesh. Because of the very high efficiency which such algorithms can obtain on machines like the CM-5 and CM-200, such fine grid simulations may cost less than methods requiring elaborate, unstructured grids.

The ARPS code for meso-scale weather prediction employs a regular grid and explicit finite differencing to implement a hydrodynamic model that can capture and predict localized, nonlinear weather phenomena [36].

In the following two sections we describe performance results obtained for these translated versions of PPM and ARPS on the CM-200 and CM-5. In general, applications employing explicit, finite difference and finite volume numerical techniques [28] are good candidates for self-similar implementation.

4.1 CM-200

For the CM-200 results described in this section, we used a single quad that contains 256 Weitek floating point units (all arithmetic was 64-bit) and 1 Gbyte of memory; only NEWS communication was required in the final, optimized, Fortran-P-generated versions. The slicewise Fortran compiler (version 1.2) was employed; runs were performed in dedicated mode. All timing data were obtained from *cmtimer* library routines [19]. These results do not include any time for I/O.

A Case Study in PPM Translation: muscl15

The Fortran-P tool has successfully translated *muscl15*, a two-dimensional hydrodynamics program consisting of nearly 5000 lines of code, from

Fortran-P to CM Fortran. The *muscl15* program uses a MUSCL algorithm developed by van Leer and Woodward [37, 38]. The original MUSCL scheme was one-dimensional Lagrangian; *muscl15* represents extensions by Woodward to perform two-dimensional Eulerian calculations by adding a remap step and by applying operator splitting to treat gradients in the x- and y-directions independently. The *muscl15* code is almost completely vectorizable. The Fortran-P translation of *muscl15* was fully automatic once the code was modified to fit the Fortran-P model; the translated code has been tested and executed on the CM-200. We performed both performance and correctness debugging while testing the Fortran-P translator with *muscl15*.

As in most PPM codes, *muscl15* proceeds according to the directional-splitting algorithm of Strang [39] by performing an *x-pass*, where gradients are applied to each row of the data domain, followed by a *y-pass* where gradients are applied to each column; these passes are then applied again in reverse order. The sequence *x-y y-x* makes up a pair of time steps. The operations are performed on temporary one-dimensional arrays loaded with data copied from the original two-dimensional arrays (representing pressure, density, and x- and y-velocities); the overhead for these data copies is trivial because many floating point operations are performed on each row or column.

Each pass is preceded by an update of the boundary zones; the Fortran-P compiler recognizes that in this code section only boundary regions are accessed and generates the necessary fake zone updates. The width of the boundary zones in *muscl15* is fixed at five zones, just enough to obviate the need for any communication during a single pass. During each pass a series of differencing operations, implemented as vector loops, is applied to each data strip; as this differencing proceeds the values within fake zones become invalid and no longer represent the true value of the corresponding zones in neighboring processors; loop bounds become progressively "narrower" as the pass over a single strip is performed and the valid strip of fake zones moves in towards the real zones. We consider computations over boundary zones as overhead but include them in the total flop/s count; real flop/s include only operations on real zones. This distinction will become important when we examine the subgrid ratio issue.

We counted the total flop/s per two-dimensional subdomain (*nxn* real zones) per time step in *muscl15* as $2308n^2 + 4n*nbdy - 10168n$ where *nbdy* is the width of the fake zones on each side. The real flop/s per two-dimensional subdomain equaled $2308n^2$. The resulting Mflop/s (measured per time step) for varying subdomain size on the CM-200 are given in Table 1; the problem size was fixed at 2^{20} zones. Note that we show the total and real Mflop/s and their ratio (which we refer to as the real zone ratio) for increasing subgrid sizes.

In Table 1 we see that performance is tied to the subgrid ratio, which is the number of subdomains per physical processor. The CM Fortran compiler version 2.0 parallelizes across subgrids assigned to a single physical processor. The subgrid ratio, in effect, becomes the vector length executed by each processor and to get the best performance this ratio must be high [14]. This, in turn, suggests that each piece of an array assigned to a physical processor be small so that there are many small pieces that can then be overlapped upon the physical processors to create many subgrids per node. A drawback of such an approach is that communication is required each time an expression is calculated using arrays offset from each other (as is typical in finite difference and finite volume numerical techniques); if communication is slow relative to computation, this approach can degrade performance. (Conversations with other CM-200 users who employ this strictly data parallel approach suggests that even on highly parallel codes this effect does limit the speed to less than 300 Mflop/s per quadrant of the CM-200.) In

Table 1. *muscl15* Speed and Time on CM-200 for Fixed Problem Size (2^{20} Zones, 256 Floating Point Nodes, 8K Processors, 1 Gbyte Memory)

Processor Grid	Subgrid Ratio	Real Zone Ratio	Total Mflop/s	Total Time(s)	Real Mflop/s
16 × 16	1	0.90	152	34.5	138
32 × 32	4	0.85	550	10.3	468
64 × 64	16	0.74	866	7.9	612
128 × 128	64	0.59	814	10.1	480

contrast, the Fortran-P approach is to partition the data domain directly, assigning each processor a contiguous subdomain to compute. The Fortran-P approach exploits the fact that each node in both the CM-200 and CM-5 implements either pipeline or vector parallelism. These nodes can exploit such parallelism available in each subdomain to run at maximum speed. Communication with neighboring processors is required occasionally, when fake zones become stale, and to implement boundary conditions. Exploiting this node parallelism is important because the interconnection networks are slow relative to processing elements: Fast processors need not overwhelm a slow network if they compute primarily with local data.

In *muscl15* the real zone ratio drops as the subgrid ratio increases because the fake zone width remains constant as the subgrids become smaller. For *muscl15* a subgrid ratio of 16 works best for the version 2.0 compiler, which does not support parallel execution across array elements located within a single, physical processor (i.e., declared as :SERIAL). Without this restriction, the results in Table 1 suggest that *muscl15* could run at nearly 800 real Mflop/s per quad.

Subdomain partitioning to increase subgrid ratios would be unnecessary if the CM Fortran compiler would allow parallel execution along serial dimensions; in fact, partly in response to our requests this support has become available in the CM Fortran compiler version 2.1.

Table 2 shows the performance of *muscl15* for increasing problem sizes. For these runs the two-dimensional subdomain size was fixed at 64 × 64 zones and the processor grid was increased incrementally from 16 × 16 to 64 × 32. It can be seen that the real zone ratio remains nearly constant and the speed increases to about 760 Mflop/s for the largest grid (over 8 million zones). This confirms the result from Table 1 concerning the performance attainable for *muscl15* on the CM-200.

ARPS Weather Code

Prior to working with the actual ARPS code [4], which is fully compressible and three dimensional, we translated a simpler, two-dimensional shallow water code which, although incompressible, embodies the nonlinear dynamics of the full ARPS code. The shallow code neglects the vertical structure of the atmosphere as well as moist and turbulent processes. Because the structures used to map the atmosphere to the processing elements work in horizontal patches, there is a close correspondence between the shallow water model and the full ARPS code. The Fortran-P translator converted the shallow water model after appropriate programmer modifications to meet the Fortran-P model. Working with code developers we added additional code to indicate boundary updates were necessary (by writing only into fake zones) and extended array dimensions and loop bounds by *nbdy* on each side. The converted code consists of approximately 900 lines of CM Fortran. The two-dimensional shallow code data domain was decomposed into equal-sized patches that were then mapped to the physical processing elements.

Table 3 presents performance results for the shallow code runs and is similar to Table 1 for *muscl15*: The subgrid ratio is varied between 1, 4, 16, and 64 but the problem size remains fixed at 2^{20} zones. The subdomain size for subgrid ratio of one was 64 × 64; the computation included 100 time steps.

We do not show the real zone ratio for the shallow code. The shallow code uses a simpler boundary zone treatment than *muscl15*, requiring only two fake zones on a side; within the time step almost no fake zone updates are calculated (except for the actual boundary-handling code itself). Hence, almost all zone updates are to real zones. In addition, the shallow model uses simpler dif-

Table 2. ***muscl15*** **Speed and Time on CM-200 for Increasing Problem Size (64 × 64 Zones/Subdomain, 256 Floating Point Nodes, 8K Processors, 1 Gbyte Memory)**

Processor Grid	Subgrid Ratio	Real Zone Ratio	Total Mflop/s	Total Time(s)	Real Mflop/s
16 × 16	1	0.90	152	34.5	138
32 × 16	2	0.91	308	34.3	282
32 × 32	4	0.92	602	34.9	554
64 × 32	8	0.92	828	50.8	762

Table 3. ARPS Shallow Code Performance with Fixed Problem Size (256 Floating Point Nodes, 8K Processors, 1 Gbyte Memory)

Processor Grid	Subgrid Size	Subgrid Ratio	Shallow Mflop/s	Shallow Time(s)
16 × 16	64 × 64	1	122	198
32 × 32	32 × 32	4	380	63
64 × 64	16 × 16	16	950	25
128 × 128	8 × 8	64	1275	19

ferencing than the full ARPS because it contains less "physical" detail; more fake zones would likely be required on the full code. There are approximately 229 flop/s per zone update per time step (vs. 2308 flop/s per zone update for *muscl15*).

It is evident again from Table 3 that the subgrid ratio is the key to performance; 1275 Mflop/s is achieved on a single quad with a subgrid ratio of 64 on a million zone calculation. A subgrid ratio of 4 yielded 380 Mflop/s.

In Table 4 the problem size was not fixed but varied with the subgrid ratio, which varied from 1 to 8. Two subdomain sizes were employed: 64 × 64 (2048 zones) and 64 × 32 (4096 zones) while the number of subgrids varied from 256 (subgrid ratio of one) to 2048 (subgrid ratio of 8). Hence the problem size varied from 1/2 million zones to over 8 million zones.

As in previous results, it is clear in Table 3 that performance is linked directly to the subgrid ratio and increases to a maximum of 780 Mflop/s with a subgrid ratio of 8. This represents a large problem size: over 8 million zones on a grid with dimensions 4096 × 2048. Because the speed increases nearly linearly with the problem size the execution time stayed nearly constant for these runs as the problem size increased. Performance results with the full ARPS code using the Fortran-P translator can be found in O'Keefe and Sawdey [40].

4.2 CM-5

As mentioned earlier, the CM-5 at Minnesota has 544 nodes (2176 vector units) and 17 Gigabytes of main memory (32 Mbytes per node). Each node has four vector units that were installed on the machine in late 1992. For the timings described here, a 128-node partition was employed. We used version 2.0 of the CM Fortran compiler with 64-bit arithmetic; timings were performed with exclusive access to the partition. As with the CM-200 all timing data were obtained using *cmtimer* library routines [20]. Timings on a 512-node partition show a factor of 4 speedup over 128 nodes if the problem size is also increased by factor of 4. This is to be expected as node communication is minimal due to the characteristics of Fortran-P algorithms (and in particular PPM) and special run-time routines called by the Fortran-P precompiler, that we describe in the following section

Another Case Study in PPM Translation: hppmfair14

We translated *hppmfair14*, a recently developed PPM code with an improved boundary treatment for irregular shapes. The *hppmfair14* code is over 5000 lines of Fortran-P. We briefly describe the boundary treatment in this new code.

As an example, consider the flow around an object in a wind tunnel. A simplistic way of representing the boundary interface between the object

Table 4. ARPS Shallow Code Performance with Increasing Problem Size (256 Floating Point Nodes, 8K Processors, 1 Gbyte Memory)

Processor Grid	Subgrid Ratio	Patch Size 64 × 64 Mflop/s (sec)	Patch Size 64 × 32 Mflop/s (sec)
16 × 16	1	120 (186)	110 (99)
32 × 16	2	250 (180)	225 (95)
32 × 32	4	440 (191)	440 (98)
64 × 32	8	780 (217)	765 (114)

and the moving gas on a regular grid would be as an impenetrable series of "stair steps." This approach is workable because of the large computational grids (typically a million zones) possible on modern supercomputers. Because PPM uses solutions to the Riemann shock tube problem to determine fluxes at zone interfaces, the simple solution to a Riemann problem at a reflecting wall (either moving or stationary) may be used to provide well-defined pressures at the edges of those zones bordering on an object.

This boundary treatment has been improved and implemented in the *hppm14fair* code. The improvement is based on the simple line interface calculation (SLIC) method [26], allowing "fractional" steps that produce a smoother boundary for a given computational grid [41]. This retains the computational ease inherent with uniform grids while permitting the freedom necessary for the description of complex shaped objects. The flux at a zone interface is constructed using the fractional volume of the object in that zone and its neighboring zone by blending the solution to a Riemann problem at a reflecting wall with the solution to the Riemann problem due to the discontinuities in the fluid states on either side of the zone interface.

The Fortran-P translation for the CM-5 was different from that for the CM-200. The Fortran-P translator performed two major tasks to produce CMF from the source code. First, it translated serial DO-loops into Fortran-90 style array expressions. The necessary array declarations and layout directives were also generated, both for explicit array variables and scalar temporary variables within loops. These scalar temporaries were promoted to temporary arrays. Second, it created aliases where needed to remove unnecessary communication between processors.

The beta (version 2.0) CMF compiler does not vectorize across explicitly local (:SERIAL) axes so we declared local axes as special (:NEWS) axes. Operations on the extracted one-dimensional strips in *hppmfair14* used arrays offset in the processor local axes by small constants; the compiler generates unnecessary communication in this case. We needed a way to "convince" the compiler that no communication was required. This was accomplished by a library routine furnished by Thinking Machines that created aliases for arrays indexed by small constant offsets. The alias was then used in place of the actual offset array reference. The Fortran-P translator proved invaluable for creating these aliases because there were many. For example, everywhere there was a reference such as `dx(i-1)` in a DO-loop indexed by `i`, an alias such as `dx_m1` and `dx` itself was supplied in the calling argument list. For the worst case routine in *hppmfair14* this required 220 parameters. The use of aliases saves memory and reduces unnecessary floating point operations as the subgrid ratio can be one. Note that no new temporary arrays are created, merely temporary array names that are aliases for array sections from the one-dimensional strips.

Several other performance enhancements were performed by hand as they were in very localized areas of the code and were considered likely to change radically very soon. These included the use of another run-time library routine, *vector_move_always*, to perform data movement local to a processor; without this special routine we found that code that extracts one-dimensional strips from the two-dimensional data arrays generates unnecessary communication, slowing the calculation significantly, *vector_move_always* also proved useful in reducing communicating in the boundary section and fake zone code. In addition, special techniques related to obtaining high performance communication between processors were employed. Note that the need for *vector_move_always* and aliases was removed in version 2.1 of CM Fortran. Performance was approximately equivalent between version 2.1 and version 2.0 with these special calls.

Table 5 summarizes current CM-5 performance results for the translated version of *hppmfair14*. The processor grid was set to $32x$

Table 5. *hppmfair14* Performance with Increasing Problem Size (128 Nodes, 512 Vector Units, 32 × 16 Processor Grid)

Problem Size	Subgrid Size	Real Zone Size	Total Mflop/s	Real Mflop/s
512 × 256	32 × 32	16 × 16	1160	580
1536 × 768	64 × 64	48 × 48	1773	1330
3584 × 1792	128 × 128	112 × 112	2011	1760

16, yielding one subdomain for each vector unit. From Table 5 it is clear that performance depends on problem size, varying from 580 Mflop/s for 131,000 zones to 1760 Mflop/s for 6.4 million zones. Scaling the larger problem size by a factor of 4 would yield almost 6.5 Gflop/s on the full 512-node CM-5.

These results are preliminary and will improve as the CM-5 compiler improves and in particular as more optimization passes are implemented and tuned. Our initial studies have shown that overheads related to the run-time system seem to be significant.

4.3 Summary

We have described performance results for three Fortran-P codes, all translated by the Fortran-P translator for the CM-200 and CM-5. Performance on the CM-200 is directly related to the subgrid ratio; for Fortran-P codes a wider fake zone boundary reduces the frequency of communication required. Unfortunately, this conflicts with the need for partitioning subdomains further to increase the subgrid ratio and results in wasted memory and floating point operations. For the *muscl15* code this ultimately limited performance to 600 Mflop/s. Given the simple differencing and narrow fake zone width in the shallow code, increasing the subgrid ratio is not as much a problem, and over 1200 Mflop/s is achieved on this code. This problem has been removed in version 2.1 of CM Fortran. We used special run-time routines from Thinking Machines to reduce communication and avoid large subgrid ratios on the CM-5; on large problems we have been able to show good performance.

The results in this section have shown that Fortran-P codes can be translated to execute at very high speed on both the CM-200 and CM-5. We plan to use the translator as a powerful tool for translating codes and executing large calculations on MPPs and in understanding the performance of these machines on real Fortran-P application codes.

5 CONCLUSIONS

In this article we have described the Fortran-P approach to programming MPPs. We have shown that it is possible to translate self-similar codes automatically and achieve good performance. Future work includes further refinement of the programming model and improvements to the Fortran-P translator to achieve even better performance on the CM-5. The Fortran-P translator will serve a dual role as a translation engine for converting codes and performing real calculations as well as a tool to experiment with different source translations to isolate compiler, machine, and application code bottlenecks in the MPP environment.

ACKNOWLEDGMENTS

We would like to thank several people at Thinking Machines for their crucial assistance and expert guidance during the translator development, in particular Woody Lichtenstein, Tony Kimball, and Paul Fink. We would like to thank the Fortran-P group at Minnesota, including Steve Soltis, Aaron Sawdey, Olivier Meirhaeghe, and Duc Nguyen for their help with translator development and performance measurements. Kurt Fickie and Tim Rohaly at the Army Research Lab (ARL) provided early support and input on the first versions of the translator. Bob Numrich at Cray Research helped us understand the Cray MPP Fortran model and how to target it for Fortran-P. Kelvin Droegemeier helped us understand and translate the ARPS shallow code. The staff at the Minnesota Supercomputer Center kept the machines running and were there to assist us with many problems: we thank Steve Saroff, Barbara Bryan, Alan Klietz, Jeff Kays, Barb Kraft, and Gary Hansen.

We would like to thank the funding agencies that have generously supported this effort. Paul Woodward was supported by the U.S. Department of Energy grant DOE DE-FG02-87ER25035 as was Steve Anderson. Matthew O'Keefe was supported by a National Science Foundation Research Initiation Award, by the U.S. Army Ballistics Research Lab (now the ARL), and by travel support from the Center for the Analysis and Prediction of Storms, an NSF Science and Technology Center at the University of Oklahoma. Hank Dietz was supported by the Office of Naval Research and the National Science Foundation. Paul Woodward, B. Kevin Edgar, and T. Parr were supported through the Army High Performance Computing Research Center, which also generously provided machine time on the CM-200 and CM-5 and the infrastructure and environment to make it all happen.

REFERENCES

[1] Thinking Machines Corporation, "Connection Machine Model CM-5," Technical Summary, October 1991.

[2] M. Walker, "The Cray MPP," talk given at ECMWF Conference on Parallel Processors in Meterology, Reading, England, November 1992.

[3] D. Porter, A. Pouquet, and P. Woodward, "Kolmogorov-like spectra in decaying three-dimensional supersonic flows," submitted to *Physics of Fluids*.

[4] K. Droegemeier, K. Johnson, K. Mills, and M. O'Keefe, *Proceedings of the 5th Workshop on the Use of Parallel Processors in Meteorology*. Reading, England: ECMWF, 1992.

[5] O. Lubeck, M. Simmons, and H. Wasserman, *Supercomputing '92*. Minneapolis, MN: IEEE Press, 1992, pp. 403–412.

[6] P. R. Woodward, *Astrophysical Radiation Hydrodynamics*. D. Reidel Publishing Co., 1986, pp. 245–326.

[7] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr., *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation*. Toronto, Canada: Association for Computing Machinery, 1991, pp. 145–156.

[8] D. Bailey, E. Barsczc, L. Dagum, and H. Simon, *Supercomputing '92*. Minneapolis, MN: 1992, pp. 386–393.

[9] P. R. Woodward and P. Colella, "The numerical solution of two-dimensional fluid flow with strong shocks," *J. Comp. Phys.*, vol. 54, pp. 115–173.

[10] P. Colella and P. R. Woodward, "The piecewise parabolic method (PPM) for gas-dynamical simulations," *J. Comp. Phys.*, vol. 54, pp. 174–201, 1984.

[11] H. Dietz, "The refined-language approach to compiling for parallel supercomputers," Ph.D. thesis, Polytechnic University, August 1986.

[12] W. Lichtenstein, *How CMF Works*. Cambridge, MA: Thinking Machines Corp., 1992.

[13] G. Sabot, *1992 Conference on Frontiers of Massively Parallel Proceedings*. McLean, VA: IEEE Press, 1992, pp. 12–20.

[14] G. Sabot, "CM Fortran optimization notes: slicewise model," Cambridge, MA: Thinking Machines Corp., Technical Report TMC-184, March 1991.

[15] Thinking Machine Corporation, *CM Fortran Release Notes, Preliminary Documentation for version 2.1 Beta 1*. Cambridge, MA: Thinking Machine Corporation, April 1993.

[16] P. Woodward, "Interactive scientific visualization of fluid flow," *IEEE Computer* vol. 26, pp. 13–26, 1993.

[17] D. Arneson, S. Beth, T. Ruwart, and R. Tavakley, "A testbed for a high performance file server," *Proc. 1993 IEEE Symposium on Mass Storage Systems*." Monterey, CA, 1993.

[18] *High Performance Fortran Language Specification, version 1.0, High Performance Fortran Forum*. May 1993.

[19] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distributed Systems*, vol. 3, pp. 179–193, 1992.

[20] Thinking Machine Corporation, *CM Fortran Reference Manual*. Thinking Machine Corporation, 1991.

[21] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD distributed memory machines," *Communications ACM*, vol. 35, pp. 66–78, 1992.

[22] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press, 1989, p. 6.

[23] M. Gerndt, "Updating distributed variables in local computations," *Concurrency Practice Exp.*, vol. 2, pp. 171–193, 1990.

[24] Thinking Machines Corporation, "Connection Machine Model CM-2 Technical Summary," TMC Technical Report TR89-1.

[25] T. Blank, *35th IEEE Computer Society International Conference (COMPCON)*. 1990, pp. 20–24.

[26] W. Noh and P. Woodward, "SLIC (simple line interface calculation)," Lecture Notes Phys., vol. 59, 1976.

[27] G. Fox, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Technical Report TR 90-141, Department of Computer Science, Rice University, December 1990.

[28] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing*. Cambridge, MA: Cambridge University Press, 1989.

[29] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener, *Fortran 90 Handbook*. New York, NY: McGraw-Hill, 1992.

[30] Cray Research, Inc., *CF77 Compiling System, Volume 1: Fortran Reference Manual*. Eagan, MN: Cray Research, Inc., version 4.0, 1990.

[31] Z. Bozkus, S. Ranka, and G. Fox, *1992 Conference on Frontiers of Massively Parallel Proceedings*. McLean, VA: 1992.

[32] T. Parr and H. Dietz, "Purdue compiler construction tool set (PCCTS) reference manual," *SIGPLAN Notices*, vol. 27, 1992.

[33] T. J. Parr, "Sorcerer—a source-to-source translator generator," AHPCRC Preprint 93-094, University of Minnesota, September 1993. *International Conference on Compiler Construction*. Edinburgh, Scotland: ACM, April 1994.

[34] A. Sawdey, M. O'Keefe, and T. Parr, "Implementing a Fortran 77 to CM Fortran Translator Using the Sorcerer Source-to-Source Translator Generator," AHPCRC Preprint 93-102, University of Minnesota, October 1993.

[35] D. Pase, T. MacDonald, and A. Meltzer, *MPP Fortran Programming Model.* Cray Research, Inc., November 24, 1992.

[36] *ARPS Version 3.0 User's Guide*, Center for the Analysis and Prediction of Storms, University of Oklahoma, Norman, OK, October 1992.

[37] B. van Leer, "Towards the Ultimate Conservative Difference Scheme. V. A Second-order sequel to Godunov's Method," *J. Comp. Phys.*, vol. 32, pp. 101–136, 1979.

[38] B. van Leer and P. Woodward, *Proceedings of the International Conference on Computing Methods in Nonlinear Mechanics.* Austin, TX: 1979.

[39] W. G. Strang, "On the construction and comparison of difference schemes," *SIAM J. Numerical Analysis*, vol. 5, pp. 506, 1968.

[40] M. T. O'Keefe and A. C. Sawdey, *Proceedings of the Les Houches Workshop on HPC in the Geosciences.* Boston: Kluwer. Les Houches, France, 1993.

[41] B. K. Edgar and P. R. Woodward, "Diffraction of a shock wave by a wedge: comparison of PPM simulations with experiment." *Int. Video J. Eng. Res.*, 1994 (in press).

[42] D. Arneson, S. Beth, T. Ruwart, and R. Tavakley, *Proceedings of the 1993 IEEE Symposium on Mass Storage Systems.* Anaheim, CA: IEEE Press, pp. 169–176, 1993.

[43] C. Fischer and R. LeBlanc, *Crafting a Compiler.* Menlo Park, CA: Benjamin/Cummings, 1988.

[44] S. McCormick, ed., *Multigrid Methods: Theory, Applications, and Supercomputing.* New York, NY: Dekker, 1988.

APPENDIX 1

In this appendix we show the corresponding CM-5 translation for the *monslp* subroutine. Note that the CM-200 translation for *monslp* was given in Section 3 of this article. The translated code for the CM-5 is:

```
      subroutine monslp(a,da,dal,dalfac,darfac,n)

      include 'defs.h'
      include '/usr/include/cm/CMF_defs.h'
      parameter (NODE_X =64,NODE_Y =16)
      real, array(NODE_X,NODE_Y,SUB_G,SUB_L:SUB_R) :: darfac,dalfac,dal,da,a
CMF$LAYOUT darfac(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dalfac(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dal(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT da(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT a(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
      integer,save, array(CMF_SIZEOF_DESCRIPTOR) :: a_m1
      integer,save, array(CMF_SIZEOF_DESCRIPTOR) :: dal_p1
      call equiv1d(dal_p1,dal,1)
      call equiv1d(a_m1,a,-1)

      call x_monslp(da,dal_p1,darfac,dalfac,a_m1,a,dal)

      return
      end

      subroutine x_monslp(da,dal_p1,darfac,dalfac,a_m1,a,dal)
      real, array(NODE_X,NODE_Y,SUB_G,SUB_L:SUB_R) ::
      da,dal_p1,darfac,dalfac,a_m1,a,dal
CMF$LAYOUT da(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dal_p1(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
```

```
CMF$LAYOUT darfac(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dalfac(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT a_m1(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT a(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dal(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
      real,save, array(NODE_X,NODE_Y,SUB_G,SUB_L:SUB_R):: thyng,s,dda
CMF$LAYOUT thyng(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT s(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
CMF$LAYOUT dda(:block=1:procs=NODE_X, :block=1:procs=NODE_Y,
:block=SUB_G:procs=1, :block=SUB_XY:procs=1)
      dal(:,:,:,2:n) = a(:,:,:,2:n)n - a_m1(:,:,:,2:n)
1000  continue

      dda(:,:,:,2:n-1) = dalfac(:,:,:,2:n-1)* dal(:,:,:,2:n-1)+
      darfac(:,:,:,2:n-1)* dal_p1(:,:,:,2:n-1)
      s(:,:,:,2:n-1) = sign(1., dda(:,:,:,2:n-1))
      thyng(:,:,:,2:n-1) =2.* amin1( s(:,:,:,2:n-1)* dal(:,:,:,2:n-1),
      s(:,:,:,2:n-1)* dal_p1(:,:,:,2:n-1))
      da(:,:,:,2:n-1) = s(:,:,:,2:n-1)* amax1(0., amin1( s(:,:,:,2:n-1)*
      dda(:,:,:,2:n-1),
     &  thyng(:,:,:,2:n-1)))
2000  continue

      return
      end
```

which was translated from the following Fortran-P version:

```
      subroutine monslp (a, da, dal, dalfac, darfac, n)
      parameter( NODE_X=64, NODE_Y=16 )
      dimension  a(n), da(n), dal(n), dalfac(n), darfac(n)
      do 1000  i = 2,n
      dal(i) = a(i) - a(i-1) 1000  continue
      do 2000  i = 2,n-1
      dda = dalfac(i) * dal(i)  +  darfac(i) * dal(i+1)
      s = sign (1., dda)
      thyng = 2.  *  amin1 (s * dal(i),  s * dal(i+1))
      da(i) = s  *  amax1 (0.,  amin1 (s * dda, thyng)) 2000  continue
      return
      end
```

We see in the translated code the variables dal_p1 and a_m1 to dal(i+1) and a(i-1), respectively. The alias is established by a function call to equiv1d which itself is called within the "wrapper" function *monslp*; the original routine is called by the wrapper function and is renamed as *x_monslp*. All equivalencing occurs within the wrapper function.

The CMF$ LAYOUT directives employ the detailed array layout capability now available in version 2.0 of CM Fortran. These directives allow more precise control of data layout. Note that unlike the CM-200 Fortran-P translation the parallel dimensions now come first, before the local array dimensions.

APPENDIX 2

The following code is a program fragment consisting of a series of loops that perform differencing operations. A possible program context is inside the time step of a finite difference algorithm, where the arrays are computed successively, each a function of a previously calculated, lower-order quantity. For this program fragment, the boundary zones move in by two zones on each side. The correct value of *nbdy* for the whole program will depend on the differencing used in the rest of the program.

For example, when solving hyperbolic PDEs these differencing loops occur in the context of an outer time step loop. The high-order array variables ultimately are used to calculate a state variable array (perhaps pressure or density) for the next discrete time unit, which is computed with these higher order variables.

```
c
c  Incomplete program fragment representing boundary handling.
c  In these arrays, real zones extend from 1 to n; fake zones
c  extend  nbdy  zones on a side.

c
c  Move in one zone from the right.
c
      do 100   i = -nbdy+1,n+nbdy-1
      dx(i) = xl(i+1) - xl(i)
1000  continue

c
c  Move in one zone from the left.
c
      do 200   i = -nbdy+2,n+nbdy-1
      u(i) = dx(i) - dx(i-1)
200   continue

c
c  Move in another zone from the left
c

      do 300   i = -nbdy+3,n+nbdy-1
      dul(i) = u(i) - u(i-1)
300   continue

c
c  Move in another zone from the right.
c

      do 400   i = -nbdy+3,n+nbdy-2
      ddul(i) = dul(i+1) - dul(i)
400   continue
```