Migrating CM Fortran Applications to HPF

Larry Meadows lfm@pgroup.com
Douglas Miles miles@pgroup.com

The Portland Group, Inc.

Abstract

*In the course of developing **pghpf**, the PGI High Performance Fortran compiler, several CM Fortran applications have been converted to HPF and run on multiple platforms as part of customer benchmarking exercises. The existing base of CM Fortran codes, along with MP Fortran codes developed for the MasPar machines, represent perhaps the largest body of existing data parallel applications. With production applications available in such areas as oil reservoir simulation, post-stack migration, fluid flow, and particle simulation, this base represents a key starting point for the development of corresponding production HPF applications. This paper surveys aspects of these code porting exercises from the perspective of both the compiler implementor and the application programmer.*

1.0 Introduction

High Performance Fortran (HPF)[1] has its roots in existing data parallel languages such as CM Fortran and Vienna Fortran. While not sweeping in nature, there have been several successful application implementations on both the CM-2 and CM-5 using CM Fortran (CMF) [2-6]. Early successes such as these helped prompt the formation of the HPFF and the subsequent definition and publication of the HPF draft standard.

This paper describes porting issues encountered in the migration of CMF codes to HPF. Section 2 discusses differences in the way data distribution directives are specified in the two languages. Section 3 notes some differences between the HPF Library and the corresponding CMF utility routines on which they are based. Section 4 outlines some issues relating to optimization, and section 5 gives some data on the effort required in various ports and lists some conclusions.

2.0 Data distribution directives

The CM Fortran data layout directives are generally more simple than the data distribution directives available in HPF. There is no concept of "template" in CM Fortran. The 3 available directives are `LAYOUT`, `ALIGN`, and `COMMON`.

CM Fortran layout directives are of the form:

```
cmf$ layout array-name(axis-desc(s))
```

where one or more axis descriptors specify the distribution along each axis of the given array. An axis descriptor can be `:serial`, `:news`, or `:send`. A serial axis is collapsed on processor, and can be thought of identically to the "*" specification for a collapsed axis in an HPF distribution directive. `:news` and `:send` distributed axes are designed to allow the programmer to think of data as being distributed among processors in a grid or nearest neighbor fashion. While not equivalent, the ports done thus far have been effective through simply replacing appearances of `:news` or `:send` with an HPF block distribution.

Thus, layout directives of the form:

```
cmf$ layout x(:serial,:news)
cmf$ layout y(:news,:news)
```

would be replaced by HPF directives of the form:

```
!hpf$ distribute x(*,block)
!hpf$ distribute y(block,block)
```

One key difference in the directive specifications from a mechanical standpoint is that an arbitrary number of arrays of arbitrary dimension can be specified in a single CM Fortran layout directive. For example, the following is a legal layout directive:

```
cmf$ layout x(:serial,:news), y(:news)
```

The strict attributed form of the HPF distribution directives prohibits such a specification. The common use of layout directives of the above form puts the directive conversion slightly out of reach of a simple *sed* script. The simplest equivalent HPF directive sequence would be of the form:

```
!hpf$ distribute x(*,block)
!hpf$ distribute y(block)
```

All of the applications ported thus far rely entirely on canonical alignments rather than alignments through an explicit `ALIGN` directive (is there a message there?). However, the specification for the CM Fortran alignment

To appear, Frontiers '95, Washington, D.C. February 6-9, 1995, © IEEE

directive appears to be a strict subset of that available in HPF, so conversions should be straightforward.

The CM Fortran COMMON directive is largely a holdover from the CM-2 where it was used to specify whether a given common block should reside on the front-end processor or on the CM array. Simply ignoring these directives and letting the HPF compiler handle the common blocks in a natural fashion is effective.

PGI is considering the possibility of directly supporting CM Fortran directives. While not difficult, the questions arises as to whether it's better to support most or all of CM Fortran or to provide the user with tools that allow easy migration to HPF. A distinct possibility is the ability to accept CM Fortran as input and produce equivalent HPF as output. Most of the difficulty lies in support for the CMF utility routines, as the next section elaborates.

3.0 CM Fortran utility routines

CM Fortran includes an extensive library of utility routines which support dynamic array allocation, parallel prefix and suffix (scan) operations, array scatter operations, parallel I/O, random number generation, and sorting. These routines were designed to provide users with acceptable performance on these important operations in the early releases of CM Fortran, before the compiler was fully mature. Many applications rely on them to a significant degree. One of the applications ported included over 300 calls to such routines. With the exception of parallel I/O, most of these operations have equivalents in High Performance Fortran in the form of HPF Library routines. This section describes some of the differences between the implementations that have a bearing on optimization.

Scan operations. Conversion of CM Fortran scan operations to their HPF Library equivalents is straightforward. The only significant difference is in the form of the segment specifier. In CM Fortran, segment boundaries for scan operations are delimited by TRUE values in the logical segment vector. In HPF, a symmetric representation was adopted. Under this convention, any shift from TRUE to FALSE or vice versa represents a segment boundary. Thus, a segment vector such as the following:

```
(/T,F,F,T,F,F,F,T,F/)
```

in CMF would represent 3 segments of size 3, 4 and 2 elements respectively. The equivalent segment vector in HPF is represented as follows:

```
(/T,T,T,F,F,F,F,T,T/)
```

thus a call such as the following in CM Fortran:

```
call cmf_scan_add(dst,src,seg,1,
&         cmf_upward,cmf_inclusive,
&         cmf_start_bit,.true.)
```

must be replaced by two calls to the HPF Library. The first converts the mask to the HPF convention, and the second performs the scan operation:

```
seg = parity_prefix(seg,1,.true.,.false.)
dst = sum_prefix(src,1,.true.,seg,.false.)
```

Conversion of the segment vector format is potentially a significant source of overhead, so the above solution should only be used until an application can be converted to use native HPF-style segment vectors. One seemingly significant omission from the HPF library is the failure to include an optional mask argument on the copy_prefix and copy_suffix routines. As a result, one of the applications ported required the addition of a MERGE after completion of the copy scans.

Array scatters. Array scatter operations with combining are more straightforward to translate, but can potentially be far less efficient in HPF unless the compiler is able to interpret the HPF Library call adequately. Array scatters in CM Fortran are achieved with a sequence of calls which define the schedule for the scatter operation and separately perform the scatter itself. This allows the programmer to explicitly re-use schedules, thereby eliminating a significant source of overhead. A typical sequence might look as follows:

```
geo = cmf_get_geometry_id(x)
call cmf_make_send_address(saddr)
call cmf_deposit_grid_coordinate(geo,saddr,
&        1,index,mask)
call cmf_send_add(x,saddr,y,mask)
```

The first three calls combine to create the schedule for scattering of the array elements under control of the index vector. This schedule can then be re-used an arbitrary number of times. The equivalent HPF Library formulation is as follows:

```
x = sum_scatter(y,x,index,mask)
```

It's much simpler, but potentially far less efficient. It's important in this case for the HPF compiler to interpret the HPF Library call (in particular a sequence of such calls) and generate code which re-uses communication schedules where possible. Section 5 discusses this optimization in more detail.

Parallel I/O. Parallel I/O is an area that is simply not addressed by version 1 of the HPF draft standard. Fortunately, CM Fortran implements a relatively simple yet effective form of parallel I/O that can be easily emulated on a Unix file system. PGI has implemented such an emulation library, though it is not part of the production version of the compiler. These simple emulation routines allow the programmer to open, close,write, read, and unlink parallel files on the Unix

file system using an interface that is identical to that available in CM Fortran. Only the fixed-machine-size version of these routines has been implemented. Interfaces for the more general routines are provided, but are not fully functional.

Random numbers and sorting. CM Fortran provides a simple random number generator, `CMF_RANDOM`, similar to the Fortran 90 `RANDOM_NUMBER` intrinsic. In addition, faster random number generators are provided which factor out table initialization and if necessary guarantee that an identical sequence is generated regardless of the number of processors on which the program is run. HPF provides no analog to these fast random number generators, an issue which should be addressed in version 2 of the HPF draft standard. In the applications ported thus far, PGI has taken the approach of allowing the user to optionally link in a fast parallel random number generator in place of the standard `RANDOM_NUMBER` intrinsic. While inconvenient, it is the simplest way of providing the necessary functionality. Implementation of the CM Fortran versions is under consideration.

Sorting, another important operation that is difficult to perform efficiently on a parallel computer, is achieved through use of the `CMF_ORDER` utility routine in CM Fortran. The HPF Library routine `GRADE_UP` provides identical functionality. However, `GRADE_UP` returns a permutation of the input array indices which describes the ordering whereas `CMF_ORDER` actually returns a rank vector. Conversion from one format to the other can be performed with a simple `FORALL` statement of the form:

```
forall (i=1:n) rank(perm(i)) = i
```

where `perm` is the permutation vector returned from `GRADE_UP`. Again, this is a case where the application should eventually be modified to use the permutation vector returned by `GRADE_UP` in order to avoid translation overhead.

4.0 Compiler optimizations

An HPF compiler must perform many optimizations in order to generate efficient code. Much previous work has focused on efficient compilation of FORTRAN 77 codes into data parallel SPMD codes. In contrast, our work on porting CM Fortran codes has caused us to focus on efficient compilation of Fortran 90 data parallel codes. Several areas have proved to be important.

Shift Operations. CM Fortran provides an extremely efficient implementation of circular shift. In fact, many programs have been constructed to use `cshift`, even when the problem did not naturally map onto this formulation.

Initially, circular shifts were implemented in *pghpf* by creating a temporary and calling a runtime routine. However, this resulted in excessive data copying and memory use. Two `cshift` optimizations have since been added (note that similar remarks apply to `eoshift`):

(1) When the dimension to be shifted is known at compile time, and the shift amount is a constant, an overlap circular shift is used. For example, consider a four-point stencil computation such as:

```
B = A + cshift(A,-1,1) + cshift(A,1,1) +
        cshift(A,-1,2) + cshift(A,1,2)
```

This is implemented by allocating overlap areas in each dimension of A, then performing an overlap circular shift call, and performing the computation in-line:

```
call overlap_cshift(A,1,1,1,1)
B = A + A(2:n+1,:) + A(0:n-1,:) +
        A(:,2:n+1) + A(:,0:n-1)
```

(2) Even if the shift amount is not known at compile time, the circular shift is done in-line as two array assignments. This avoids run-time overhead and exposes the data motion to further optimizations.

Schedule Reuse. In most data parallel programs, the same pattern of data motion is performed many times on different data. A pattern of communication is called a *schedule*. The *pghpf* runtime system allows any schedule to be reused.

Two kinds of schedules exist. *Regular* schedules depend only on a regular index set, and not on the value of runtime data. The creation of these schedules requires no communication. *Irregular* schedules depend on the runtime values of array variables, and require communication for their creation. Reuse of both types of schedules is helpful, but is of most benefit for irregular schedules.

The CM Fortran utility library allows explicit creation of schedules and their reuse, as indicated in the previous section. The challenge for an HPF compiler is to recognize and reuse schedules without programmer intervention. There are two cases where this is valuable:

(1) As noted above, a sequence of HPF Library scatter calls results in a reusable schedule when the index and mask arrays are the same, and the source and destination arrays have the same mapping.

(2) Similar reuse can be obtained when an index array is used multiple times as a vector subscript. For example, one of the benchmark codes contains statements like:

```
temp(rankno) = x
x = temp
temp(rankno) = y
y = temp
```

where `rankno` is an array of integers. A schedule for the array scatter operation can be created once and used multiple times.

Default distributions and subroutine interface. One of the most persistent problems in conversion of CM Fortran programs to HPF is the default distribution of CM Fortran arrays. By default, every CM

Fortran array of the same size is distributed the same way. Thus, it is often not necessary to specify layout directives for CM Fortran arrays.

In contrast, *pghpf* does not distribute arrays by default; rather, they are replicated. Thus, when converting a CM Fortran program, it is often necessary to add HPF mapping directives that had no counterpart in the original CM Fortran program.

An additional problem is the dynamic redistribution allowed by HPF at subroutine boundaries. Full support of this requires substantial overhead that is not present in CM Fortran programs.

pghpf provides solutions to these problems using interprocedural analysis. The compiler warns of redistributions at subroutine boundaries; it is rare that these redistributions are intended by the user. Future work will propagate distributions around boundaries, and reduce overhead by reusing array descriptors when distributions do not change.

5.0 Conclusions

Three substantial applications have been ported from CMF to HPF by The Portland Group. These are:

EULER, a multi-material simulation benchmark written by Olaf Lubek's group at LANL and derived from their production code PAGOSA (17,000 lines).

ELWA, a demonstration elastic wave simulator from KTH in Sweden (3000 lines).

PSICM, a production particle simulator written by Leo Dagum at NASA AMES (8500 lines).

Each of these applications presented different problems. Euler, despite being the largest code, was the easiest port, as most of the work had already been done by LANL. Euler demonstrates good scalability. Almost all communication is done using `cshift`; unfortunately, these mostly involve shifting in unknown dimensions, so the overlap optimizations do not apply.

Elwa was written using the poly-shift operation in the CM Fortran library. Further, all the computations were done inside subroutine calls. Converting the poly-shift operations to circular shifts and inlining the subroutines resulted in a straightforward stencil code.

PSICM was the most challenging application ported thus far. It requires efficient support for irregular communications, integer sorting, scan operations, and array scatters with combining. Early performance results on this application have been promising, but performance scaling using the original CM Fortran formulation is heavily dependent on inter-processor latency. As of this writing, the application is being re-worked to be more latency tolerant.

It has proven to be straightforward to port CM Fortran applications to HPF. Most of the effort has been in replacing or implementing CM Fortran utility routines, and in transliterating layout directives. The definition of CM Fortran appears to have been driven more from a performance perspective while HPF stresses broad functionality as well. This provides a substantial challenge to HPF compiler implementors, in that the performance of a CM Fortran application on a CM-5 will be an early standard by which HPF compilers on all platforms are judged.

Appendix - Fortran 90 / CM Fortran differences

Both HPF and CM Fortran are based on Fortran 90. However, CM Fortran was defined and implemented before the final draft of the Fortran 90 standard was published. As a result there are several minor "spelling" differences between the two languages. A brief summary of those differences is included in this appendix for easy reference. Where there is no chance for ambiguity, *pghpf* supports both the CM Fortran and the HPF spellings of common Fortran 90 features. A compile-time option to *pghpf* directs the compiler to use the CM Fortran interpretation in those cases where a conflict occurs.

Intrinsics. There are 4 intrinsics that have identical calling sequences but whose names differ between the two dialects:

CM Fortran		*F90/HPF*
`dotproduct`	`==>`	`dot_product`
`dlbound`	`==>`	`lbound`
`dubound`	`==>`	`ubound`
`dshape`	`==>`	`shape`

The *pghpf* compiler recognizes either the CM Fortran or HPF spellings of these intrinsics. Aside from potential namespace pollution, these additional spellings are benign.

There are 3 intrinsics that have identical names, but whose calling sequences differ only in the order of the arguments:

CM Fortran		*F90/HPF*
`cshift(x,d,s)`	`==>`	`cshift(x,s,d)`
`eoshift(x,d,s,b)`	`==>`	`eoshift(x,s,b,d)`
`reshape(shp,x,pad)`	`==>`	`reshape(x,shp,pad)`

where "d" is the `DIM` argument, "s" is the `SHIFT` argument, and "b" is the `BOUNDARY` argument. The reshape intrinsic has an additional optional `ORDER` argument which is identical between the two dialects. A compile-time option causes *pghpf* to interpret the above intrinsics using the CM Fortran calling convention rather than the F90/HPF calling convention.

Lastly, there are a few intrinsics that are supported in CM Fortran that never made it into Fortran 90. They are the `PROJECT`, `LASTLOC`, `FIRSTLOC`, `RANK`, `DIAGONAL`, and `REPLICATE` intrinsics. These are not currently supported by *pghpf*, but may well be added if they are encountered

commonly. None of the applications ported thus far use these intrinsics.

Array constructors. CM Fortran allows the specification of array constructors using either the Fortran 90 standard method:

```
(/ array-constructor-value(s) /)
```

or the alternative method:

```
[ array-constructor-value(s) ]
```

Again, it was straightforward to add support for the alternative array constructor syntax, so it is recognized by default by *pghpf*.

Array dimensions. A final trivial difference is that CM Fortran allows the declaration of array dimensions using an `ARRAY` keyword that has functionality identical to that of `DIMENSION` attribute in Fortran 90. This alternative syntax is accepted by *pghpf* as well.

References

[1] High Performance Fortran Forum. *High Performance Fortran Language Specification* (Version 1.0). Rice University, May 3, 1993.

[2] L. Dagum. "Three-Dimensional Direct Particle Simulation on the Connection Machine", *Journal of Thermophysics and Heat Transfer* Vol. 6, No. 4, Oct.-Dec. 1992.

[3] C. Levit and D. Jespersen. "A Computational Fluid Dynamics Algorithm on a Massively Parallel Computer", *Int. J. Supercomputer Appl.* 3(4):9-27, 1989.

[4] N.K. Madavan. "Unsteady Turbomachinery Flow Simulations on Massively Parallel Architectures", *Computing Systems in Engineering*, 3(1-4):241-249, 1992.

[5] P. Lomdahl et al. "50 GFlops Molecular Dynamics on the Connection Machine 5", *Supercomputing '93 Proceedings,* 520-527, 1993.

[6] L. Long and J. Myczkowski. "Solving the Boltzmann Equation at 61 gigaflops on a 1024-node CM-5", *Supercomputing '93 Proceedings,* 528-534, 1993.