OpenMP and HPF: Integrating Two Paradigms*

Barbara Chapman[1] and Piyush Mehrotra[2]

[1] VCPC, University of Vienna, Vienna, Austria.
[2] ICASE, MS 403, NASA Langley Research Center, Hampton VA 23681.
barbara@vcpc.univie.ac.at, pm@icase.edu

Abstract. High Performance Fortran is a portable, high-level extension of Fortran for creating data parallel applications on non-uniform memory access machines. Recently, a set of language extensions to Fortran and C based upon a fork-join model of parallel execution was proposed; called OpenMP, it aims to provide a portable shared memory programming interface for shared memory and low latency systems. Both paradigms offer useful features for programming high performance computing systems configured with a mixture of shared and distributed memory. In this paper, we consider how these programming models may be combined to write programs which exploit the full capabilities of such systems.

1 Introduction

Recently, high performance systems with physically distributed memory, and multiple processors with shared memory at each node, have come onto the market. Some have relatively low latency and may support at least moderate levels of shared memory parallelism. This has motivated a group of hardware and compiler vendors to define OpenMP, an interface for shared memory programming which they hope to see adopted by the community. OpenMP [2] supports a model of shared memory parallel programming which is based to some extent on PCF [3], an earlier effort to define a standard interface, but increases the power of its parallel constructs by permitting parallel regions to include subprograms.

HPF was designed to exploit data locality and does not provide features for utilizing shared memory in a system; OpenMP provides the latter, however it does not support data locality. In this paper we consider how these paradigms might be combined to program distributed memory multiprocessing systems (DMMPs) which may require both of these. We assume that such a system consists of a number of interconnected **processing nodes**, or simply **nodes**, each of which is associated with a specific physical memory module and one or more general-purpose **processors** which share this memory and execute a program's instructions at run time: we emphasize this two-level structure by calling such systems SM-DMMPs.

* This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while both authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

We first consider interfacing the paradigms, enabling each of them to be used where appropriate. However, there is also potential for a tighter integration of at least some features of HPF with those offered by OpenMP. We therefore examine an approach which combines HPF data mappings with OpenMP constructs, permitting exploitation of shared memory whilst coordinating the mapping of data and work to the target machine in a manner which preserves data locality.

This paper is organized as follows. Section 2 briefly describes the features of each paradigm, and in Section 3 we consider an interface between them based upon the HPF extrinsic mechanism. We speculate on the potential for a closer integration of the two, and illustrate each of them with a multiblock application example.

2 A Brief Comparison of HPF and OpenMP

High Performance Fortran (HPF) is a set of extensions to Fortran, designed to facilitate data parallel programming on a wide range of architectures [1]. It presents the user with a conceptual single thread of control. HPF directives allow the programmer to specify the distribution of data across processors, thus providing a high level description of the program's data locality, while the compiler generates the actual low-level parallel code for communication and scheduling of instructions. HPF also defines directives for expressing loop parallelism and simple task parallelism, where there is no interaction between tasks. Mechanisms for interfacing with other languages and programming models are provided.

OpenMP is a set of directives, with bindings for Fortran 77 and C/C++, for explicit shared memory parallel programming ([2]). It is based upon the *fork-join* execution model, in which a single master thread begins execution and spawns worker threads to perform computations in parallel as required. The user must specify the parallel regions, and may explicitly coordinate threads. The language thus provides directives for declaring such regions, for sharing work among threads, and for synchronizing them. An order may be imposed on variable updates and critical regions defined. Threads may have private copies of some of the program's variables. Parallel regions may differ in the number of threads which execute them, and the assignation of work to threads may also be dynamically determined. Parallel sections easily express other forms of task parallelism; interaction is possible via shared data. The user must take care of any potential race conditions in the code, and must consider in detail the data dependences which need to be respected.

Both paradigms provide a parallel loop with private data and reduction operations, yet these differ significantly in their semantics. Whereas the HPF INDEPENDENT loop requires data independence between iterations (except for reductions), OpenMP requires only that the construct be specified within a parallel region in order to be executed by multiple threads. This implies that the user has to handle any inter-iteration data dependencies explicitly. HPF permits private variables only in this context; otherwise, variables may not have different values in different processes. OpenMP permits private variables – with potentially differ-

ent values on each thread – in all parallel regions and work-sharing constructs. This extends to common blocks, private copies of which may be local to threads. Even subobjects of shared common blocks may be private. Sequence association is permitted for shared common block data under OpenMP; for HPF, this is the case only if the common blocks are declared to be sequential.

The HPF programming model allows the code to remain essentially sequential. It is suitable for data-driven computations, and on machines where data locality dominates performance. The number of executing processes is static. In contrast, OpenMP creates tasks dynamically and is more easily adapted to a fluctuating workload; tasks may interact in non-trivial ways. Yet it does not enable the binding of a loop iteration to the node storing a specific datum. Thus each programming model has specific merits with respect to programming SM-DMMPs, and provides functionality which cannot be expressed within the other.

3 Combining HPF with OpenMP

In this section we examine two ways of combining HPF and OpenMP such that we can exploit the features of both approaches. Note that the logical processors of HPF can be associated with either the *nodes* or with the individual *processors* of an SM-DMMP. In the former case, the user must rely on automatic shared memory parallelization to exploit all processors on a node. In the latter case, a typical implementation would create separate address spaces for each individual processor, and even communication between two processors on the same node will require an explicit data transfer.

3.1 OpenMP as an HPF Extrinsic Kind

It is a stated goal of HPF to interoperate with other programming paradigms. The language specification defines the *extrinsic* mechanism for interfacing to program units which are written in other languages or do not assume the HPF model of computation. The programming language and computation model must be defined. An interface specification must be provided for each *extrinsic* procedure. This approach keeps the two models completely separate, which implies that the compilers can also be kept distinct; in particular, the OpenMP compiler does not need to know anything about HPF data distributions. For the HPF calling program, execution should proceed just as if an HPF subprogram was invoked. HPF provides language interfaces for Fortran (90/95), Fortran 77 and C. We consider only Fortran programs and OpenMP constructs in this paper.[1]

HPF provides the pre-defined programming models *global*, *serial* and *local* for use with extrinsics. It also permits additional models so long as conceptually one copy of the called procedure executes initially. That is the case for an OpenMP program. We define an *extrinsic* model below which allows us to invoke OpenMP subroutines and functions from within an HPF program. We also indicate how an alternative model might be defined.

[1] The two Fortran language versions differ in the assumptions on sequence and storage association, particularly when arguments are passed to subroutines.

The OpenMP extrinsic model We base the *OpenMP* extrinsic kind upon the predefined local model. It assumes that processors defined in an HPF program are associated with nodes of the target system (the number of nodes, not the total number of processors, is returned by HPF's NUMBER_OF_PROCESSORS function).

An OpenMP extrinsic procedure called under this model is realized by creating a single local procedure on each node (or HPF processor). These execute concurrently and independently, exclusively within their respective node, until they return to the HPF calling program upon completion. The calling program blocks until all local procedures have terminated. With the extrinsic kind *OpenMP*, therefore, each local procedure is an independent OpenMP routine, initially executed by a master thread on a processor of the node it is invoked on. It will create worker threads on the same node when a parallel region is encountered; no communication is possible with a local OpenMP procedure on another node. Only the processors and memory on that node are available to it.

An explicit interface will describe the HPF mapping of data required by the routine; any remapping required is performed before the *OpenMP* extrinsic procedure is called. Each invocation will access the local part of distributed data only. Scalar data arguments are available on each node. Only data visible to the master thread will be returned: if they are scalar, the value must be the same on each node. The OpenMP routine may not return to the caller from a parallel region or from within a synchronization construct, and may not have alternate entries. An OpenMP library routine will return information related to the procedure executing on the node where it is invoked only.

This programming model might be very useful for systems with high latency between nodes, such as clusters of workstations, where it is important to map data to the node which needs them. It permits use of a finer grain of parallelism to exploit the processors within each such node once the data is in place. However, it is also restrictive: an *OpenMP* extrinsic routine can only exploit parallelism within a single node, since other nodes, and their data, are not visible to it.

An alternative extrinsic model Other extrinsic kinds may be defined which conform to the requirements of HPF. In particular, an alternative can be defined which permits an OpenMP routine to utilize an entire platform, i.e. it begins with a single master thread which is cognizant of all processors executing the program. The team of threads created to execute a parallel region in the OpenMP code may thus span the entire machine. Since the OpenMP routine will not understand HPF data mappings, all distributed arguments must be mapped to the processor which will initiate the routine during its setup.

This model enables the user to switch between programming models, depending upon which is most suitable for some region of the program. However, it does not enable OpenMP to take advantage of HPF's data locality, and the initial overhead might be large. Its practical value is therefore not clear.

3.2 Merging HPF and OpenMP

An interface between two paradigms obviously does not permit full exploitation of their features. The *OpenMP* extrinsic may be satisfactory for SM-DMMP architectures with high latency. However, the alternative model is less appealing as an approach to programming large low-latency SM-DMMPs, which may benefit from both data locality and the flexibility of a shared memory model. We thus consider augmenting the `OpenMP` directives with `HPF` directives directly, so that both shared-memory parallelism and data locality can be expressed within the same routines. `OpenMP` directives can then describe the parallel computation, while `HPF` directives may map data to the nodes where it is used and bind the execution of loop iterations to nodes storing distributed data.

Data Mappings The `HPF` data mapping directives were carefully defined to affect a program's performance, but not its semantics. Thus they can be easily integrated into `OpenMP` without changing the semantics of `OpenMP` programs.

It is natural to associate logical `HPF` processors with processing nodes under this model, since data will be shared within a node unless it is thread-private. Techniques from `HPF` compilers may be applied to prefetch data in bulk transfers where permitted by the semantics of the code. The resulting language might allow privatization of distributed variables just as `HPF` does inside `INDEPENDENT` loops. In general, however, we expect that `HPF` directives will be used to explicitly map large data objects, which are likely to be shared rather than being private to each thread.

`HPF` does not support sequence or storage association for explicitly mapped variables. This restriction must carry over to the integrated language so that the compiler may exploit the mapping information. Common blocks with distributed constituents must follow rules of both `HPF` and `OpenMP`. That is, if they are named in a `THREADPRIVATE` directive, or in a `PRIVATE` clause, each private copy inherits the original distribution. Storage for the explicitly mapped constituents is dissociated from the storage for the common block itself.

In `HPF` programs, mapped data objects may be implicitly remapped across procedure boundaries to match the prescriptive mapping directives for the formal arguments. On the other hand, if they have been declared `DYNAMIC`, objects can be explicitly remapped using a `REDISTRIBUTE` or `REALIGN` directive. In order to facilitate remapping within an `OpenMP` program, we must impose the following constraint: *Any code which implicitly or explicitly remaps a data objects must be encountered by all the threads that share that object.* This ensures that if the object is remapped, the new mapping is visible to all the threads which have access to the data.

Additional HPF Constructs `HPF` provides the `ON` directive to specify the locus of computation of a block of statements, a loop iteration or subroutine call. This might be used as an alternative to the `OpenMP` scheduling options to bind the execution of code in a work-sharing construct to one or more nodes, or

logical HPF processors. An ON clause enables the user to specify, for example, the node where a SINGLE region or a SECTION of a parallel sections construct is to be executed. Similarly, it may bind each iteration of a DO construct to the node where a specific data item is stored.

Finally, in contrast to the INDEPENDENT directive of HPF, parallel loops in OpenMP may have inter-iteration data dependencies. Thus an additional work-sharing construct could be based upon the INDEPENDENT loop, which permits the prefetching of data and may eliminate expensive synchronizations.

4 An Example

Scientific and engineering applications sometimes use a set of grids, instead of a single grid, to model a complex problem domain. Such *multiblock* codes use tens, hundreds, or even thousands of such grids, with widely varying shapes and sizes.

Structure of the Application The main data structure in such an application stores the values associated with each of the grids in the multiblock domain. It is realized by an array, *MBLOCK*, whose size is the number of grids *NGRID* which is defined at run time. Each of its elements is a Fortran 90 derived type containing the data associated with a single grid.

```
TYPE GRID
  INTEGER NX, NY
  REAL, POINTER :: U(:, :), V(:, :), F(:, :), R(:, :)
END TYPE GRID
TYPE (GRID), ALLOCATABLE :: MBLOCK(:)
```

The decoupling of a domain into separate grids creates internal grid boundaries. Values must be correctly transferred between these at each step of the iteration. The connectivity structure is also input at run time, since it is specific to a given computational domain. It may be represented by an array *CONNECT* (not shown here) whose elements are pairs of sections of two distinct grids.

The structure of the computation is as follows:

```
DO ITERS = 1, MAXITERS
  CALL UPDATE_BDRY (MBLOCK, CONNECT)
  CALL IT_SOLVER (MBLOCK, SUM)
  IF ( SUM .LT. EPS) THEN finished
END DO
```

The first of these routines transfers intermediate boundary results between grid sections which abut each other. We do not discuss it further. The second routine comprises the solution step for each grid. It has the following main loop:

```
SUBROUTINE IT_SOLVER( MBLOCK, SUM )
```

```
    SUM = 0.0
    DO I = 1, NGRID
     CALL SOLVE_GRID( MBLOCK(I)%U, MBLOCK(I)%V, ... )
     CALL RESID_GRID( ISUM, MBLOCK(I)%R, MBLOCK(I)%V, ... )
     SUM = SUM + ISUM
    END DO
  END
```

Such a code generally exhibits two levels of parallelism. The solution step is independent for each individual grid in the multiblock application. Hence each iteration of this loop may be invoked in parallel. If the grid solver admits parallel execution, then major loops in the subroutine it calls may also be parallelized.

Finally the routine also computes a residual; for each grid, it calls a procedure *RESID_GRID* to carry out a reduction operation. The sum of the results across all grids is produced and returned in *SUM*. This would require a parallel reduction operation if the iterations are performed in parallel.

Distributing the Grids We again presume that an HPF processor maps onto a node of the underlying machine rather than to an individual physical processor. We map each grid to a single node (which may "own" several grids) by distributing the array *MBLOCK*. Such a mapping permits levels of parallelism to be exploited. It implies that the boundary update routine may need to transfer data between nodes, whereas the solver routine can run in parallel on the processors of a node using the shared memory in the node to access data. A simple BLOCK distribution, as shown below, may not ensure load balance.

```
!HPF$ DISTRIBUTE MBLOCK( BLOCK )
```

The INDIRECT distribution of HPF may be used to provide a finer grain of control over the mapping.

4.1 Using an OpenMP Extrinsic Routine

This version calls an *OpenMP* extrinsic to perform the solution step on each grid as well as compute local contributions to the residual. The boundary updates and the final residual summation are performed in the calling HPF program.

An interface declaration is required for the *OpenMP* extrinsic routine:

```
EXTRINSIC('OPENMP', 'LOCAL')
  SUBROUTINE OPEN(MBLOCK, M, LSUM)
    INTEGER M(:)
    REAL LSUM(:)
    TYPE (GRID), ALLOCATABLE :: MBLOCK(:)
```

Only the *local* segment of *MBLOCK* will be passed to the local procedure, *M* is its size and *LSUM* is the residual value for the local grids on the node.

The extrinsic subroutine is invoked on each node independently. The master thread creates the required number of threads locally. It will return when each of the threads has terminated. The calling HPF routine waits until all extrinsic procedures have terminated.

```
SUBROUTINE OPEN( LOC_MBLOCK, NLOC, LOC_SUM )
      SUM = 0.0
      CALL GET_THREAD(NLOC, LOC_MBLOCK, N)
!$OMP CALL OMP_SET_NUM_THREADS(N)
!$OMP PARALLEL DO SCHEDULE (DYNAMIC), DEFAULT(SHARED)
!$OMP REDUCTION (+: LOC_SUM)
      DO I = 1, NLOC
       CALL SOLVE_GRID( LOC_MBLOCK(I)%U, ... )
       CALL RESID_GRID( ISUM, LOC_MBLOCK(I)%R, ... )
       LOC_SUM = LOC_SUM + ISUM
      END DO
      RETURN
END
```

This approach is suitable for systems where the latency between nodes is comparatively high. It permits direct experimentation with, and exploitation of, threads in order to balance work despite large differences in grid sizes.

4.2 Combining HPF and OpenMP

The combined model is very similar to the above; however, it is simpler to write in the sense that there is no need to construct an extrinsic interface. The data declarations and distribution are identical to those of the HPF program. The *IT_SOLVER* routine can be directly written with OpenMP directives in a manner very similar to the code shown above:

```
SUBROUTINE IT_SOLVER( MBLOCK, SUM )
!HPF$ DISTRIBUTE MBLOCK( BLOCK )
      SUM = 0.0
      CALL GET_THREAD(NGRID, MBLOCK, N)
!$OMP CALL OMP_SET_NUM_THREADS(N)
!$OMP PARALLEL DO SCHEDULE (DYNAMIC), DEFAULT(SHARED)
!$OMP REDUCTION (+: SUM)
      DO I = 1, NGRID
!$HPF   ON (HOME (MBLOCK(I)), RESIDENT
          CALL SOLVE_GRID( MBLOCK(I)%U, ... )
          CALL RESID_GRID( ISUM, MBLOCK(I)%R, ... )
!$HPF   END ON
        SUM = SUM + ISUM
      END DO
      RETURN
END
```

This version again deals with the full array *MBLOCK*, rather than with local segments. Thus, there may be a need to specify the locus of computation so that each iteration is performed on the processor which stores its data. This can be done using an HPF-style ON-block along with the RESIDENT directive, as shown.

The execution differs. This time, the initialization of *SUM* will be performed once only, on a single master thread which controls the entire machine. The number of threads, N, is computed for the entire system, and the work is distributed across all processors. This may enable an improved load balance. We have avoided a two-step computation of *SUM*.

This approach is appropriate for a system with comparatively low latency, which allows a single master thread on the machine and a global scheduling policy, permitting a compiler/runtime system to do global load balancing if appropriate.

5 Summary

OpenMP was recently proposed for adoption in the community by a group of major vendors. It is not yet clear what level of acceptance it will find, yet there is certainly a need for portable shared-memory programming. The functionality it provides differs strongly from that of HPF, which considers the needs of distributed memory systems. Since many current architectures have requirements for both data locality and efficient shared memory parallel programming, there are potential benefits in combining elements of these two paradigms. We have shown in this paper that the HPF extrinsic mechanism enables usage of each within a single program. This may facilitate the programming of workstation clusters with standard interconnection technology, for example. It is less clear if a large low latency machine will benefit from a simple interface. For these, there appears to be added value in providing some level of integration. Given the orthogonality of many concepts in the two programming models, they are fundamentally highly compatible and a useful integrated model can be specified. In particular, a combination of HPF data locality directives with OpenMP parallelization constructs extends the scope of applicability of each of them. Since at least one major vendor has already provided data mapping options together with a shared memory parallel programming model [4], we expect that this avenue will be closely explored by the OpenMP consortium in the near future.

References

1. High Performance Fortran Forum: High Performance Fortran Language Specification. Version 2.0, January 1997
2. OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October, 1997
3. B. Leasure (Ed.): Parallel Processing Model for High Level Programming Languages, Draft Proposed National Standard for Information Processing Systems, April 1994
4. Silicon Graphics, Inc.: MIPSpro Fortran 77 Programmer's Guide, 1996