Enhancing OpenMP With Features for Locality Control *

Barbara Chapman[a] *Piyush Mehrotra*[b] *Hans Zima*[c]

[a]Dept.Electronics and Computer Science
University of Southampton, Highfield, Southampton SO17 1BK, United Kingdom
E-Mail: bmc@ecs.soton.ac.uk

[b]ICASE, MS 403, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: pm@icase.edu

[c]Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
E-Mail: zima@par.univie.ac.at

Abstract

`OpenMP` is a set of directives extending Fortran and C which provide a shared memory programming interface for shared-address space machines. However, there are no directives for controlling the locality of data. Such control can critically affect performance on machines which exhibit non-uniform memory access times. In this paper, we present a set of extensions to `OpenMP` to provide support for locality control of data. These extensions are based on similar directives from `HPF`, a language which focusses on controlling the distribution of data. The integrated language is particularly suitable for hybrid architectures such as clusters of SMPs.

1 Introduction

Recently a proposal was put forth for a set of language extensions to Fortran and C based upon a fork-join model of parallel execution; called `OpenMP` [4], it aims to provide

*The work described in this paper was partially supported by the Special Research Program F011 "AURORA" of the Austrian Science Fund and by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 and NAS1-97046, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

a portable shared memory programming interface for shared memory and low latency systems. The OpenMP interface supports a well-established model of shared memory parallel programming. It extends earlier work done by the Parallel Computing Forum (PCF) as part of the X3H5 standardization committee [5] and the SGI directives for shared memory programming [6]. In addition to the parallel regions and constructs of PCF, it allows users to exploit shared memory parallelism at a reasonably coarse level by extending the scope of directives; thus the range of applicability of the paradigm is increased.

OpenMP is an explicitly parallel programming model in which the user is able to explicitly generate threads to utilize processors of a shared memory machine and is also able to control accesses to shared data in an efficient manner. Such a paradigm will work effectively as long as the memory is equidistant, i.e., the underlying hardware does not exhibit any memory locality. However, due to hardware scaling issues, a large number of shared address space machines are implemented using physically distributed memory, thus exhibiting non-uniform memory access times. This can critically affect the overall performance of the code. Yet, the OpenMP model does not allow the user to control the mapping of data to machines across a larger system, nor does it permit the explicit binding of computations to a machine based upon data locality.

In contrast to OpenMP, the focus of HPF has been NUMA machines, in particular systems with underlying distributed memory. It presents the user with a conceptual single thread of control and provides facilities for data mapping and work distribution so that the locality of the underlying memory can be exploited.

HPF does not have features for exploiting the availability of shared memory in a system; OpenMP does not support explicit control of data locality. Since many systems may benefit from both of these, it seems worthwhile to explore possibilities for interfacing the paradigms, enabling each of them to be used where appropriate.

Given that there is a potential performance gain if HPF-like data locality is used in conjunction with shared memory parallel loops, it could be appealing to combine features of HPF and OpenMP. In an earlier paper [1], two of us had proposed several paradigms for integrating HPF and OpenMP including using an extrinsic interface for invoking OpenMP routines from HPF code. In this paper, we discuss one of these options in greater detail; we describe how OpenMP can be extended with HPF-like directives to provide control over the locality of data. The resulting paradigm might enable the creation of a parallel program in which the underlying shared address space is exploited to the maximum, while data and work are mapped to processors in a coordinated fashion which respects the fact that the memory is physically distributed. The hybrid paradigm represented by the integrated language is particularly suitable for controlling the mapping of data and work in hybrid architectures such as clusters of SMPs.

This paper is organized as follows. We first briefly compare HPF and OpenMP in Section 2. Section 3 details the proposed extensions while Section 4 provides a more complete example using these directives. The paper concludes with a short summary.

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 [2]. 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[3].

OpenMP is a set of directives, with bindings for Fortran 77 and C/C++, for explicit shared memory parallel programming [4]. 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 loop-carried 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 different 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 parallel architectures, and provides functionality which cannot be expressed within the other.

3 Locality Control in OpenMP

The current OpenMP specification provides a number of features that are indirectly related to locality. For example, the **PRIVATE** clause and the **THREADPRIVATE** directive result in the creation of thread-specific instances of data objects for each thread in a parallel region. This can be exploited by the compiler to enforce locality. Similarly, the execution of a **REDUCTION** clause is based upon an implicit privatization of the reduction variable.

However, given a cluster of SMPs or a NUMA parallel architecture, there is no way for the user to specify the mapping of large data structures across the nodes of the machine, or to control the distribution of work in such a way that threads operating on a data structure execute on the same node on which the data reside.

In this section, we propose extensions to OpenMP that address this problem. We begin by introducing directives for HPF-style data mapping (Section 3.1). Based on these features, we describe features for the control of data mapping in the context of privatization (Section 3.2). After proposing independent loops as a new work sharing construct for OpenMP in Section 3.3, we discuss the on clause as a mechanism for data-driven scheduling of the work in parallel loops and parallel sections (Section 3.4).

3.1 Data Mapping

In the HPF model, data objects are first aligned and then distributed to a set of *abstract processors*, where the mapping from abstract to physical processors is implementation-dependent. These features were carefully defined to affect a program's performance without changing its semantics. As a consequence, data mapping directives are orthogonal to OpenMP semantics and can be integrated relatively easily.

In our integrated model, the HPF **PROCESSORS** directive is replaced by a **NODES** directive, which is used to specify the number of nodes available for the execution of the parallel program, and to make them accessible. The *abstract nodes* introduced by such a directive are 1-1 mapped to physical nodes of the underlying machine; the specification of this mapping is implementation dependent and is of no concern here. The difference between HPF processors and the nodes in our scheme is important: while each HPF processor is associated with exactly one sequential thread, many OpenMP threads may operate on a single node at any time, and the physical processors of a node share the data residing at that node.

Consider the following example (HPF-related directives in the integrated language will be prefixed by "!HOP$").

```
!HOP$ NODES  P(NUMBER_OF_NODES())
      REAL U(N,N)
!HOP$ DISTRIBUTE U (*, BLOCK) ONTO P
```

In this example, the extent of the node array, P, is defined by the number of nodes made available to the computation. The columns of the two-dimensional array U are block distributed across all nodes of P.

The mapping of an array can be modified in two ways: either implicitly, via procedure calls, or explicitly, by executing a redistribute or realign directive. Such a directive must be executed on all threads which have access to that array.

3.2 Privatization

A **PRIVATE** clause in an OpenMP parallel region, when applied to a shared variable, creates a thread-specific instance of the variable for each thread in the team associated with the parallel region. The related directives **FIRSTPRIVATE** and **LASTPRIVATE** respectively allow the initialization of the private instances, and a transfer of the "last" value of a private instance back to the shared variable at the end of the parallel region.

The *location* of a private instance associated with a thread is not an issue dealt with in the semantics of the OpenMP language: an implementation will usually put the object on the stack of the associated thread. Furthermore, we expect that data structures that are distributed will in most cases be declared as *shared*, due to the cost of memory resources caused by privatization.

However, assuming the possibility of nested parallelism (a feature which is explicitly allowed but not yet specified in OpenMP), the user may want to control the distribution of private data using mapping directives. If no explicit specification is made, the decision is referred to the implementation. We propose a set of features whose semantics resemble that of the HPF subroutine call, although less sophistication is needed here since private data must be "whole" data structures.

We illustrate this in the example below:

```
!HOP$ PROCESSORS  P(16)
      REAL A(N), B(N), C(N), D(N)
!HOP$ DISTRIBUTE A (BLOCK) ONTO P
!HOP$ DISTRIBUTE B (CYCLIC) ONTO  P
```

```
    ...
!$OMP PARALLEL PRIVATE(A,B,C,D), FIRSTPRIVATE(A), LASTPRIVATE(B)
!HOP$ INHERIT A
!HOP$ DISTRIBUTE B(*) ONTO MY_NODE
!HOP$ DISTRIBUTE C(BLOCK) ONTO P(1:8)
    ...
!$OMP END PARALLEL
    ...
```

The shared arrays A and B are respectively distributed by block and cyclic across all processors; no mapping is given for C and D which means that their mapping is decided by the compiler. All four variables are privatized.

- For A, a *transcriptive* specification is provided, meaning that each private instance is distributed in exactly the same way as the original variable. All instances are initialized with the actual value of the shared variable at the time the parallel region is entered.

- In contrast, a *prescriptive* specification is given for B: each private instance is allocated on a *single* node, the node on which the thread executes. No initialization is performed.

- Independent of the mapping specified (implicitly) for the shared variable C, each private instance is prescribed to be distributed by block across the first 8 nodes of P.

- The mappings for the shared variable D and its private instances are solely determined by the implementation.

The values of the shared arrays A, C, and D are undefined at the time of exit from the parallel region, with no value transfer taking place from private instances to the corresponding shared variable. However, as a result of B being declared as **LASTPRIVATE**, the "last" private instance of B (as defined in the OpenMP language specification) is transferred to the shared variable upon exit from the parallel region, which involves a redistribution to **CYCLIC**.

Common blocks that occur in a **THREADPRIVATE** directive of OpenMP can be dealt with in an analogous way. A distribution directive provided in the head of the parallel region for variables in such a common block is valid for all instances of the variable, throughout the parallel region (possibly excluding nested regions). For subsequent parallel regions that satisfy the OpenMP condition for persistence, data mappings also persist. However, the mappings can be changed upon entry to the new region.

All HPF and OpenMP restrictions regarding common blocks must be satisfied.

3.3 Independent Loops

OpenMP provides the **DO** directive, which is a work-sharing construct that can be used within the extent of a parallel region. The semantics specifies that "...the iterations of the ... do loop must be executed in parallel"; they are distributed across the existing threads according to some schedule. Note that this specification does *not* assert that the iterations of the loop are independent in the sense that they do not contain loop-carried dependences [7]. Rather, such dependences may or may not exist; if they exist, then it is the responsibility of the programmer to synchronize the threads so as to enforce correct behavior of the program execution.

In some situations, in particular where regular linear algebra computations are concerned, the compiler may assert independence of a loop. In general, however, for example in the presence of procedure calls or multiple subscript levels, as they occur in sparse matrix computations or sweeps over irregular meshes, this is not possible. In such cases the compiler must follow a conservative approach that may result in a loss of parallelism.

The HPF example below – which is an excerpt from a CFD code performing a sweep over the edges of an unstructured grid – illustrates such a situation. Since the values of the *EDGE* array are not known at compile time, the compiler cannot automatically determine the absence of loop-caried output dependences for the variable $V2$. The **INDEPENDENT** directive provides this missing piece of information by asserting independence of the loop, which in turn allows fully parallel execution.

```
!HPF$ INDEPENDENT, NEW (N1, N2, DELTAX), REDUCTION (V2)
      DO J = 1, M
       ...
       N1 = EDGE(J,1);     N2 = EDGE(J,2)
       ...
       DELTAV = F(V1(N1), V1(N2))
       V2(N1) = V2(N1) - DELTAV
       V2(N2) = V2(N2) + DELTAV
      ENDDO
```

3.4 Data-Driven Work Scheduling

The inclusion of directives for data distribution into OpenMP provides the user with control over the mapping of *data* to nodes. In order to allow full exploitation of locality it is useful to also allow the specification of the allocation of *work* operating on that data.

We shortly digress to discuss this issue in the context of HPF and OpenMP, as they are currently specified. In the HPF model, an *abstract processor* is characterized by (1) having a (local) memory where data may reside (this data are "owned" by the processor), and (2) being the locus of control for the execution of a single, sequential thread. The execution of an HPF program orchestrates the set of all threads in an SPMD fashion.

Given a data structure operated on in an HPF program, the compiler may schedule work on that structure in such a way that any modification of the data is performed on the processor which owns the data. This is called the *owner-computes rule*. For some kinds of computation that display a dynamic and irregular behavior – such as in the above example – the owner-computes rule may lead to excessive communication that can be avoided under a more flexible scheduling strategy. In the HPF Approved Extensions, the **ON** directive provides such control: attached to a statement, it allows a precise specification of the processor that is to execute a portion of work (such as a loop iteration) belonging to that statement. For example:

```
!HPF$ INDEPENDENT , ON HOME (A(I))
      DO I=...
        B(I)=A(I)*A(I)/K + ...
      ...
```

In this example, the on clause specifies a data item, $A(I)$; the HPF processor owning that piece of data – which is known at runtime when the independent loop is entered – is selected to execute iteration I of the loop.

This simple relationship between the location of data and a uniquely determined associated thread has been lost in our approach: each data item is mapped to a *node*, and a node cannot be identified with a single sequential thread, but may have arbitrarily many threads operating on it. Furthermore, the relationship of threads to nodes or physical processors is not specified in the OpenMP language.

One option for including an HPF-style on clause into OpenMP would make the binding of threads to nodes explicit and allow explicit scheduling of threads in the on clause. However, besides being a rather low-level approach this leads to interesting semantic problems which do not have an obvious solution. Consider the following example:

```
      ...
!$OMP PARALLEL DO PRIVATE (I)
      ...
!HOP$ ON THREAD (0)
      I = I+1
      ...
```

Without the on clause, the statement *I=I+1* would result in an incrementation of the private instance of I, performed on each thread separately. The on clause specifies execution on the master thread, but there is no simple rule to explain what happens to the set of private instances of I in the statement.

In order to avoid problems of this kind, our approach (1) avoids explicit thread scheduling, and (2) allows binding of the on clause only to work sharing constructs, effectively providing an additional, data-oriented variant of the OpenMP schedule directive.

We illustrate this by two small examples:

```
      REAL X(N), Y(N), Z(N)
!HOP$ DISTRIBUTE (CYCLIC) :: X
!HOP$ DISTRIBUTE (BLOCK) :: Y, Z
      ...
!$OMP PARALLEL, SHARED(X, Y, Z)
!$OMP DO, ON(HOME(Y(I)))
      DO I=1,N
         X(I) = SQRT(Y(I)*Y(I+1))/Y(I-1)+Z(I)
      END DO
!$OMP END DO
```

The on clause attached to the parallel loop specifies a schedule with the meaning that for each I, iteration I of the independent loop is executed on the home of *Y(I).* This means that only threads operating on that node may execute this iteration, without specifying the number and identity of these threads.

The example below shows the application of the on clause to a parallel section. The meaning is obvious.

```
!HOP$ PROCESSORS P(16)
      REAL X(N),...
!HOP$ DISTRIBUTE (*) ONTO P(1) :: X
      ...
!$OMP PARALLEL SECTIONS, DEFAULT(SHARED)
!$OMP SECTION ON(P(1))
      CALL Q1(X)
!$OMP SECTION ON(P(2))
      ...
!$OMP END PARALLEL SECTIONS
```

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.

Using OpenMP Directives to Express Parallelism

We map each grid to a single node (which may "own" several grids) by distributing the array *MBLOCK*. Such a mapping permits both 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.

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

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

Given the above distribution, the *IT_SOLVER* routine can be directly written with directives from `OpenMP` and the directives proposed in the last section:

```
SUBROUTINE IT_SOLVER( MBLOCK, SUM )
!HOP$ DISTRIBUTE MBLOCK( BLOCK )
        SUM = 0.0
        CALL GET_THREAD(NGRID, MBLOCK, N)

!$OMP CALL OMP_SET_NUM_THREADS(N)

!$HOP INDEPENDENT
!$OMP PARALLEL DO SCHEDULE (DYNAMIC), DEFAULT (SHARED),
&
   !$OMP  ON (HOME (MBLOCK(I)), RESIDENT
!$OMP REDUCTION (+: SUM)
        DO I = 1, NGRID
            CALL SOLVE_GRID( MBLOCK(I)%U, . . . )
            CALL RESID_GRID( ISUM, MBLOCK(I)%R, . . . )
!HPF$       END ON
          SUM = SUM + ISUM
        END DO
        RETURN
END
```

In the above code, first, the independent directive is specifying that there is no inter-iteration dependencies, i.e., the calls to *SOLVE_GRID* and *RESID_GRID* can be executed in parallel for each grid. Howvere, there is 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 the on-clause along with the `RESIDENT` directive, as shown.

Given the above code, each thread will perform the solution on each grid. If nested parallelism is turned on and the *SOLVE_GRID* is appropriately encoded, then the parallelism within the solver can also be exploited. This approach allows the user to specify that each grid has to be stored within a single node, thus avoiding any remote access of data during the solution process. Inter-node transfer of data will, thus, occur only during computation of *SUM* and during the *UPDATE_BDRY* routine (not shown here).

5 Conclusion

`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 primarily 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. 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 this paper we have described a set of extensions to the `OpenMP` directives to allow users to control data locality. The aim of such extensions is to expand the scope of applicability of `OpenMP` programs to architectures which exhibit memory locality.

References

[1] B. Chapman and P. Mehrotra, OpenMP and HPF: Integrating Two Paradigms. Proceedings of the 1998 EuroPar Conference, Southampton, 1998.

[2] High Performance Fortran Forum: High Performance Fortran Language Specification. Version 2.0, January 1997

[3] P.Mehrotra, J.Van Rosendale,and H.Zima. High Performance Fortran: History, Status and Future. In: Zapata,E. and Padua,D.(Eds.): *Parallel Computing*, Special Issue on Languages and Compilers for Parallel Computers, Vol.24, No.3-4,pp.325–354 (1998).

[4] OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October, 1997

[5] B. Leasure (Ed.): Parallel Processing Model for High Level Programming Languages, Draft Proposed National Standard for Information Processing Systems, April 1994

[6] Silicon Graphics, Inc.: MIPSpro Fortran 77 Programmer's Guide, 1996

[7] H.Zima with B.Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.