High-Level Data Mapping for Clusters of SMPs*

Siegfried Benkner[1] and Thomas Brandes[2]

[1] Institute for Software Science
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
sigi@ieee.org

[2] SCAI - Institute for Algorithms and Scientific Computing
GMD - German National Research Center for Information Technology
Schloß Birlinghoven, D-53754 St. Augustin, Germany
brandes@gmd.de

Abstract. Clusters of shared-memory multiprocessors (SMPs) have become the most promising parallel computing platforms for scientific computing. However, SMP clusters significantly increase the complexity of user application development when using the low-level application programming interfaces MPI and OpenMP, forcing users to deal with both distributed-memory and shared-memory parallelization details. In this paper we present extensions of High Performance Fortran for SMP clusters which enable the compiler to adopt a hybrid parallelization strategy, efficiently combining distributed-memory with shared-memory parallelism. By means of a small set of new language features, the hierarchical structure of SMP clusters may be specified. This information is utilized by the compiler to derive inter-node data mappings for controlling distributed-memory parallelization across the nodes of a cluster, and intra-node data mappings for extracting shared-memory parallelism within nodes. Additional mechanisms are proposed for specifying inter- and intra-node data mappings explicitly, for controlling specific SM parallelization issues, and for integrating OpenMP routines in HPF applications. The proposed features are being realized within the ADAPTOR and VFC compiler. The parallelization strategy for clusters of SMPs adopted by these compilers is discussed as well as a hybrid-parallel execution model based on a combination of MPI and OpenMP. Early experimental results indicate the effectiveness of the proposed features.

Keywords: parallel programming, HPF, OpenMP, MPI, SMP clusters, parallelization, hybrid parallelism.

1 Introduction

Many supercomputer centers have introduced clusters of shared-memory multiprocessors (SMPs) as their premier high performance computing platforms. Examples of such systems are multiprocessor clusters from SUN, SGI, IBM, a variety of multi-processor PC clusters, supercomputers like the NEC SX-5 or the future Japanese Earth Simulator and the ASCI White machine.

* This work was supported by NEC Europe Ltd. as part of the ADVICE and ADVANCE projects in cooperation with the NEC C&C Research Laboratories and by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.
Published in: Proceedings 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments, San Francisco, USA, 2001.

Clusters of SMP systems increase the complexity of user applications development by forcing programmers to deal with shared-memory programming issues such as multi-threading and synchronization, as well as with distributed-memory issues such as data distribution and explicit message-passing.

There are mainly two trends in parallel programming, depending on how the address space of parallel systems is organized. On the one hand, the standard application programming interface (API) for message-passing, MPI [13], is widely used, mainly for distributed-memory systems. Recently, a standard API for shared-memory parallel programming, OpenMP [17], has become available. While OpenMP is restricted to shared-memory architectures only, MPI programs can also be executed on shared-memory machines and clusters.

However, MPI programs which are executed on clusters of SMPs usually do not directly utilize the shared-memory available within nodes and thus may miss a number of optimization opportunities. A promising approach for parallel programming attempts to combine MPI and OpenMP in a single application. Such a strategy attempts to fully exploit the potential of SMP clusters by relying on data distribution and explicit message-passing between the nodes of a cluster, and on shared-memory and multi-threading within the nodes. While such an approach allows optimizing parallel programs by taking the hybrid architecture of SMP clusters into account, applications written in such a way tend to become extremely complex.

In contrast to MPI and OpenMP, High Performance Fortran (HPF) is a high-level parallel programming language which can be employed on both distributed-memory and shared-memory machines. The data mapping provides the necessary data locality to minimize communication and synchronization that is generated automatically by the compiler. Although HPF programs can be compiled for clusters of SMPs, the language does not provide features for exploiting the hierarchical structure of clusters. As a consequence, current HPF compilers usually ignore the shared-memory aspect of SMP clusters and treat such machines as pure distributed-memory systems.

In order to overcome these shortcomings of HPF and its compilers, we propose extensions of the HPF mapping mechanisms such that the hierarchical structure of SMP clusters can be taken into account. Based on these features, an HPF compiler can then adopt a hybrid parallelization strategy whereby distributed-memory parallelism based on message-passing, e.g. MPI, is exploited across the nodes of a cluster, while shared-memory parallelism is exploited within SMP nodes by relying on multi-threading, e.g. OpenMP.

Two sets of HPF extensions for clusters of SMPs are proposed in this paper. The first set of extensions is based on the concept of *processor mappings*, a simple mechanism for specifying the hierarchical structure of SMP clusters by introducing *abstract node arrays* onto which abstract processor arrays may be mapped by means of the basic HPF distribution mechanisms. The second set of extensions allows the specification of *hierarchical data mappings* in two separate steps, comprising an *inter-node data mapping* which maps data arrays onto abstract node arrays and an *intra-node data mapping* which maps node-local data onto processors within a node. Additional language mechanisms are provided for the integration of OpenMP routines or other node routines that operate only on data local to a node.

The rest of this paper is organized as follows. Section 2 proposes language features for optimizing existing HPF programs for clusters of SMPs by providing an explicit specification of the hierarchical structure of clusters. Section 3 sketches a hierarchical compilation and execution model for clusters of SMPs which is adopted for the parallelization of HPF programs that utilize the proposed language extensions. Section 4 describes a set of additional HPF extensions for specifying inter-node and intra-node data mappings explicitly, and a new `EXTRINSIC` model for integrating OpenMP routines into HPF applications. Related work is discussed in Section 5. Finally, conclusions and a discussion of future work are presented in Section 6.

2 Exploiting the Hierarchical Structure of SMP Clusters

HPF provides the concept of abstract processor arrangements to establish an abstraction of the underlying parallel target architecture in the form of one or more rectilinear processor arrays. Processor arrays are utilized within data distribution directives to describe a mapping of array elements to abstract processors. Array elements mapped to an abstract processor are *owned* by that processor. *Ownership* of data is the central concept for the execution of data parallel programs. Based on the ownership of data, the distribution of computations to abstract processors and the necessary communication and synchronization are derived automatically.

Consider now an SMP cluster consisting of NN nodes, each equipped with NPN processors. Currently, if an HPF program is targeted to an SMP cluster, abstract processors are either associated with the NN nodes of the cluster or with the $NN * NPN$ processors. In the first case, data arrays are distributed only across the NN nodes of the cluster and therefore only parallelism of degree NN can be exploited. In the second case, where abstract HPF processors are associated with all processors available in the cluster, potential parallelism of degree $NN * NPN$ can be exploited. However, by viewing an SMP cluster as a distributed-memory machine consisting of $NN * NPN$ processors, the shared-memory available within nodes is usually not exploited, since data distribution and communication are performed within nodes as well.

In the following we propose language features for optimizing existing HPF programs for clusters of SMPs by providing an explicit specification of the hierarchical structure of clusters.

2.1 Processor Mappings

The concept of *processor mappings* is introduced in order to describe the hierarchical structure of SMP clusters. A processor mapping specifies a mapping of an abstract processor array to an abstract node array. The `NODES` *directive* is introduced for declaring one or more abstract node arrays. For the specification of processor mappings a subset of the HPF data distribution mechanisms as provided by the `DISTRIBUTE` directive are utilized.

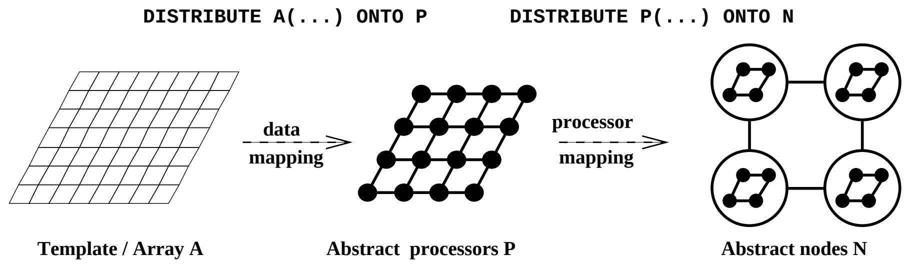
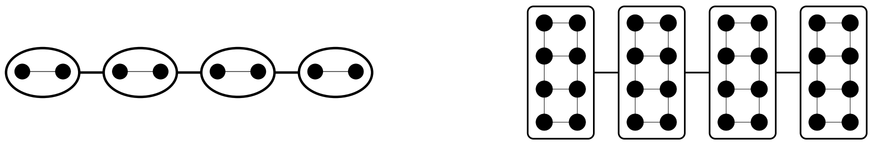


Fig. 1. Processor mappings. The hierarchical structure of SMP clusters is described by mapping processor arrays to node arrays using a subset of HPF's distribution mechanisms.

As a consequence of a processor mapping, each processor of an abstract processor array is mapped to a node of an abstract node array. Processor mappings are specified by using a subset of the HPF distribution mechanisms. For homogeneous clusters, the usual HPF `BLOCK` distribution format may be used. Heterogeneous clusters can be supported by means of the `GEN_BLOCK` distribution format of the Approved Extensions of HPF.



```
!hpf$ processors P(8)
!hpf$ nodes N(4)
!hpf$ distribute P(block) onto N

      real A(NA)
!hpf$ distribute A(block) onto P

              (a)
```

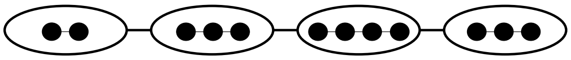
```
!hpf$ processors R(4,8)
!hpf$ nodes M(4)
!hpf$ distribute R(*,block) onto M

      real B(NB1,NB2)
!hpf$ distribute B(cyclic,block) onto R

              (b)
```

Fig. 2. Examples of processor mappings. On the left-hand side Figure (a) shows a 4x2 SMP cluster, while on the right-hand side Figure (b) illustrates a cluster of 4 nodes each with 8 processors, arranged in a 4x2 configuration.

Figure 1 illustrates the concept of processor mappings while examples of processor mappings are shown in Figures 2 and 3. Figure 2 (a) shows how the hierarchical structure of an SMP cluster, consisting of 4 nodes with 2 processors each, may be specified. Figure 2 (b) specifies an SMP cluster, consisting of 4 nodes with 8 processors each, where the processors within a node are arranged as a 4x2 array. Figure 3 specifies a heterogeneous SMP cluster, consisting of 4 nodes with 2, 3, 4, and 3 processors, respectively. Here the `GEN_BLOCK` distribution format of HPF is utilized to indicate that the number of processors within nodes varies.



```
      integer, dimension(4):: SIZE = (/2,3,4,3/)
!hpf$ processors R(12)
!hpf$ nodes M(4)
!hpf$ distribute R(gen_block(SIZE)) onto M
```

Fig. 3. A heterogeneous SMP cluster with four nodes. The `GEN_BLOCK` distribution format is used to specify the number of processors available in each node.

In order to support abstract node arrays whose sizes are determined upon start of a program, the new intrinsic function `NUMBER_OF_NODES` is provided. `NUMBER_OF_NODES` returns the actual number of nodes used to execute a program in the same way as the HPF intrinsic function `NUMBER_OF_PROCESSORS` returns the total number of processors used to execute an HPF program.

Processor mappings provide a simple means for optimizing HPF applications for SMP clusters. Using processor mappings, the hierarchical structure of SMP clusters may be specified explicitly, without the need to change existing HPF directives. Based on a processor mapping, an HPF compiler can adopt a cluster-specific parallelization strategy in order to exploit distributed-memory parallelism across the nodes of a cluster, and shared-memory parallelism within nodes.

2.2 Exploiting DM and SM Parallelism

If a dimension of an abstract processor array is distributed by the format `BLOCK` or `GEN_BLOCK`, contiguous blocks of processors are mapped to the nodes in the corresponding dimension of the specified abstract node array. As a consequence of such a processor mapping, both distributed-memory parallelism and shared-memory parallelism may be exploited for all data array dimensions that are mapped to that processor array dimension. On the other hand, if in a processor mapping a dimension of an abstract processor array is distributed by means of "*", all abstract processors in that dimension are mapped to the same node of an abstract node array, and thus only shared-memory parallelism may be exploited across array dimensions which have been mapped to that processor array dimension.

In the first example (Figure 2 (a)), both distributed- and shared-memory parallelism may be exploited for array `A`. In the other example (Figure 2 (b)), only shared-memory parallelism may be exploited across the first dimension of array `B`, while both shared-memory and distributed-memory parallelism may be exploited across the second dimension of `B`.

2.3 Deriving Inter- and Intra-Node Mappings

An HPF data mapping directive, e.g. `DISTRIBUTE A(BLOCK) ONTO P`, determines for each processor `P(I)` of the abstract processor array `P` those parts of the data array `A` that are owned by `P(I)`. If, in addition, a processor mapping for `P` with respect to a node array `N` is specified, an inter-node data mapping may be automatically derived from the data mapping and the processor mapping.

An *inter-node data mapping* determines for each node `N(J)` those parts of array `A` that are owned by node `N(J)`. The implicit assumption is that those portions of an array owned by a node are allocated in an unpartitioned way in the shared memory of this node, while data is distributed across the local memory of nodes, according to inter-node mapping.

In the same way, an *intra-node data mapping* may be derived from a data mapping and a processor mapping. Intra-node data mappings specify a mapping of the data allocated on a node of a cluster with respect to the processors within a node. Here, we assume that intra-node data mappings are utilized by the compiler to control the exploitation of shared-memory parallelism within SMP nodes. This is described in more detail in the next section.

3 Compilation and Execution Model

In this section we briefly sketch a hierarchical compilation and execution model for clusters of SMPs which is adopted for the parallelization of HPF programs that utilize the proposed language extensions. This model is currently being realized within the ADAPTOR HPF compilation system [4] and within the VFC [1] compiler.

3.1 Outline of the Compilation Strategy

Both the ADAPTOR and the VFC compiler are source-to-source translation systems which transform an HPF program into an explicitly parallel Fortran program which is then compiled by the Fortran compiler of the parallel target machine in order to obtain an executable parallel program. As opposed to the usual HPF compilation, where a single-threaded SPMD node program is generated, a multi-threaded node program is generated under the hierarchical execution model.

An extended HPF program is compiled in two phases for clusters of SMPs. In the first compilation phase the compiler analyzes the data mappings and processor mappings in order to determine an inter-node data mapping. Based on the inter-node data mapping, the compiler distributes data and work across nodes and inserts message-passing primitives in order to realize the communication of non-local data between the nodes of a cluster. The result of the first compilation phase is an SPMD Fortran/MPI message-passing program.

In Figure 4 (b) the first compilation phase is illustrated for a simple HPF code fragment showing a reduction operation on a two-dimensional array. Due to the specified processor mapping, the first dimension of `B` is classified as `shared` while the second dimension is classified as `distributed`. As a consequence, the second dimension of `B` is distributed across the 4 nodes, while the first dimension is not distributed. The `SUM` intrinsic function is transformed into a nested loop. The outer loop is strip-mined across the nodes of the cluster. The call to `HPFrt_DM_get_my_bounds` ensures that each node operates only on its local part of `B`. After the loop, communication bewteen nodes is performed by means of `mpi_allreduce` in order to combine the partial results of all nodes.

In the second phase the intermediate SPMD program is parallelized for shared-memory according to the intra-node data mapping derived by the compiler. Shared-memory parallelization is currently realized by inserting corresponding OpenMP directives in order to distribute the work of a node among multiple threads. Work distribution of loops and array assignments is derived from the intra-node data mapping of the accessed arrays and realized by corresponding OpenMP work-sharing constructs and/or appropriate loop transformations (e.g. strip-mining). Similarly, data consistency of shared data objects is enforced by inserting appropriate OpenMP synchronization primitives.

As shown in Figure 4 (c) during the second compilation phase the SPMD message-passing program is transformed by inserting OpenMP directives in order to exploit shared-memory parallelism within the nodes of the cluster. The OpenMP `parallel` directive ensures that multiple threads are generated on each node, and the call to `HPFrt_SM_get_my_bounds` enforces that each thread executes its own chunk of inner loop iterations.

3.2 The Hierarchical Execution Model

HPF programs compiled for clusters of SMPs as outlined above, are executed according to a *hierarchical execution model*. Within the hierarchical execution model an HPF program is executed by a set of parallel processes, each of which executes on a separate node of an SMP cluster within its own local address space. Usually, every abstract HPF node is associated with a separate MPI-process executed on a single node of the cluster. Process parallelism, data partitioning, and message-passing communication based on MPI is utilized across nodes.

Each node process generates a set of threads which emulate the abstract processors mapped to a node and which execute concurrently in the shared address space of a node. The data mapped to one node is allocated in a non-partitioned way in the shared memory, regardless of the intra-node mapping. Parallel execution of threads within nodes is organized on the basis of the derived intra-node data mapping which controls the distribution of computations among the threads. Consistency of shared data objects is guaranteed by automatically generated synchronization primitives [2].

```
!hpf$ processors P(6,4)
!hpf$ nodes N(4)
!hpf$ distribute P(*, block) onto N
      real B(NB1,NB2), S
!hpf$ distribute B(block, block) onto P
      ...
      S = sum(B)
```

(a) Original HPF program

```
      real B(NB1,NB2/4), S    ! for simplicity: NB2 is a multiple of 4
      integer B_DSP (...)     ! internal array descriptor for B
      integer B_LB2, B_UB2
      ...                     ! build array descriptor
      S = 0.0
      call HPFrt_DM_get_my_bounds (B_DSP, 2, B_LB2, B_UB2)
      do I = B_LB2, B_UB2
         do J = 1, NB1
            S = S + B(J,I)
         end do
      end do
      call mpi_allreduce (S, S, 1, MPI_REAL, MPI_SUM, MPI_COMM_WORLD, IERR)
```

(b) Phase 1: Intermediate SPMD Fortran/MPI message-passing node program.

```
      ...
      integer B_LB1, B_UB1, B_LB2, B_UB2
      ...                                 ! build array descriptor
      S = 0.0
      call HPFrt_DM_get_my_bounds (B_DSP, 2, B_LB2, B_UB2)
!$omp parallel, private (I,J,B_LB1,B_UB1), reduction(S)
      call HPFrt_SM_get_my_bounds (B_DSP, 1, B_LB1, B_UB1)
      do I = B_LB2, B_UB2
         do J = B_LB1, B_UB1
            S = S + B(J,I)
         end do
      end do
!$omp end parallel
      call mpi_allreduce (S, S, 1, MPI_REAL, MPI_SUM, MPI_COMM_WORLD, IERR)
```

(c) Phase 2: Generated hybrid-parallel MPI/OpenMP program.

Fig. 4. Compilation of a HPF program for clusters of SMPs.

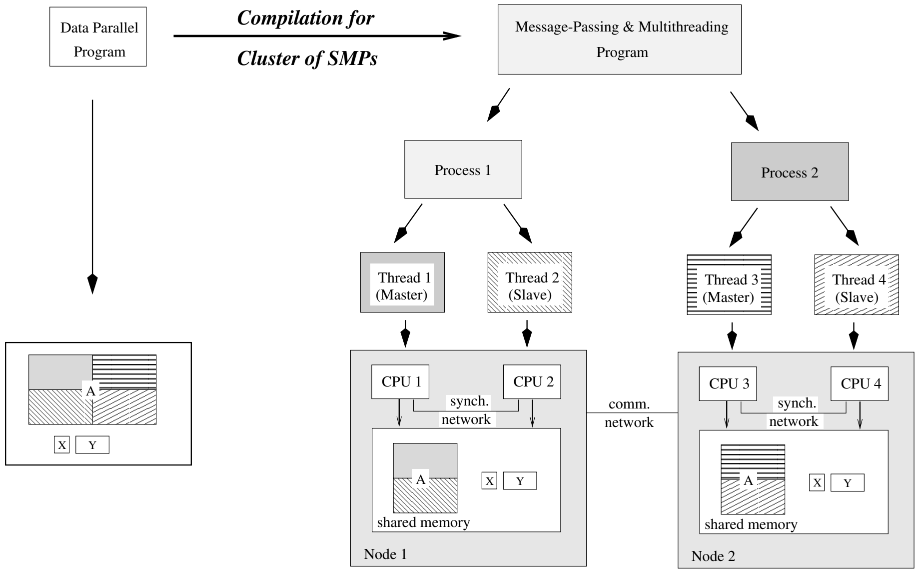


Fig. 5. Hierarchical HPF execution model.

3.3 Experiments

In this section we report on early experiments with the proposed language extensions and the hierarchical compilation and execution model on a dual-processor PC cluster. For the performance experiments, we used an existing HPF kernel from a numerical pricing module [14] developed in the context of the AURORA Financial Management System [7]. This kernel realizes a backward induction algorithm on a Hull and White interest rate tree. In the HPF code the interest rate tree is represented by 2-dimensional arrays and several index vectors which capture the structure of the tree. All these arrays have been distributed by usual HPF distribution mechanisms. The main computations are performed in an inner `INDEPENDENT` loop with indirect data accesses, which operates on a single level of the Hull and White tree. Due to the peculiarities of the algorithm, communication is required for each level of the tree, introducing a significant communication overhead.

For the performance experiments, we computed the price of a variable coupon bond with a maturity of 10 years on a one-day basis, resulting in a tree with 3650 levels. We compared the original HPF kernel to an extended kernel, where an additional processor mapping was used for specifying the hierarchical structure of the PC cluster. Both kernels have been parallelized with the VFC compiler [1] and executed on a Beowulf cluster consisting of 8 nodes, each equipped with two Pentium II (400MHz) processors, connected by Fast Ethernet. The pgf90 compiler from Portland Group Inc. was used as a back-end compiler of VFC for compiling the generated MPI/OpenMP program.

The original HPF kernel was compiled by VFC to Fortran/MPI, while the extended kernel was compiled to Fortran/MPI/OpenMP and executed according to the hierarchical execution model utilizing MPI process parallelism across the nodes of the cluster and OpenMP thread parallelism within nodes.

The original HPF kernel has been measured in two different ways which are labeled in Figure 6 `HPF/MPI-P` and `HPF/MPI-N`, respectively. `HPF/MPI-P` refers to the MPI code generated by VFC and executed on 2 to 16 processors, where both processors were utilized on each node. `HPF/MPI-N` refers to the MPI version of the generated code executed on 2 to 8 nodes, where only one processor was utilized on each node. `HPF-Cluster/MPI-OpenMP` refers to the hybrid parallel code generated by VFC from the extended HPF kernel.

As Figure 6 shows, the `HPF-Cluster` kernel significantly outperforms the original HPF kernel, if both processors in a node are used. One reason for this performance difference seems to be the communication overhead induced by MPI communication[1] within the nodes of the cluster. The times measured for the `HPF-Cluster/MPI-OpenMP` and `HPF/MPI-N` variants were almost the same, however, the `HPF/MPI-N` version required twice as many nodes.

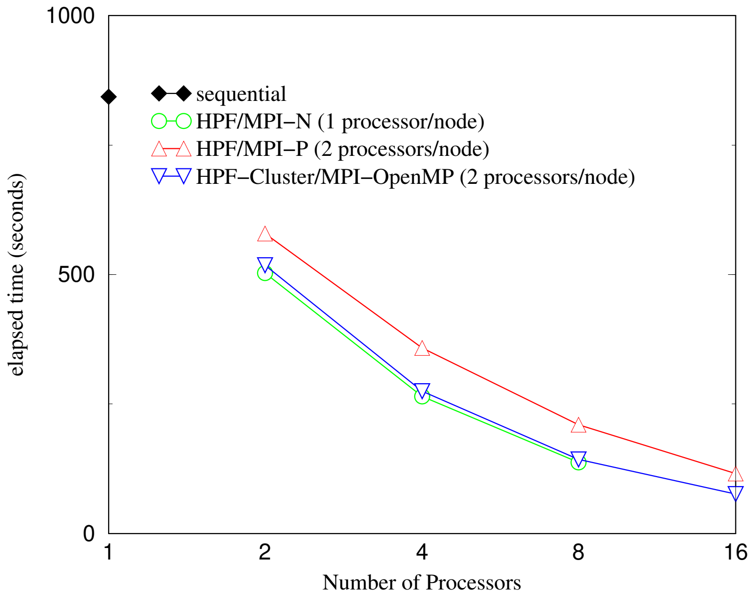


Fig. 6. Experimental Results

[1] The MPI version used on the PC cluster did not offer an optimized intra-node communication via shmem.

4 Additional Language Features

As outlined in the previous section, the introduction of abstract node arrays and processor mappings allows the specification of the hierarchical structure of SMP clusters. Based on these features, an HPF compiler may derive from existing HPF distribution an inter-node mapping which is utilized to distribute data across the nodes of a cluster, and an intra-node mapping in order to exploit shared-memory parallelism within nodes.

In this Section we describe a set of additional HPF extensions which to enable programmers to specify inter-node data mappings and intra-node data mappings explicitly. Such mappings are referred to as *hierarchical data mappings* and are defined in two steps. First, arrays are distributed to the nodes of a cluster, and second, node-local data is mapped to the processors within a cluster.

4.1 Hierarchical Data Mappings

A hierarchical data mapping may be specified explicitly by the user by first mapping data arrays onto abstract node arrays, and then specifying a mapping of node-local data with respect to the processors within a node. The first mapping is referred to as an *explicit inter-node data mapping*, and the second mapping is called *explicit intra-node data mapping*.

Inter-node data mappings may be specified explicitly by means of the HPF `DISTRIBUTE` directive which has been extended such that data arrays may be distributed directly onto abstract node arrays. The semantics of a distribution directive with an abstract nodes array as *distribution target* is equivalent to the semantics of a usual HPF distribution directive, except that data is now distributed only with respect to the nodes of a cluster. Processors within a node of a cluster usually have access to all data mapped to the shared memory of a node.

In addition to an inter-node mapping, an intra-node mapping may be explicitly specified by means of the `SHARE` directive. An intra-node data mapping specifies a mapping of node-local data to the abstract processors (threads) within a node. The syntax of the `SHARE` directive is similar to the `DISTRIBUTE` directive, but does not include an `ONTO` clause. The target of the mapping is the set of abstract processors available within a node.

The `SHARE` directive is used to control the distribution of work among the threads available on a node. In this context, the OpenMP work-sharing formats `DYNAMIC` and `GUIDED` may be employed in addition to the usual HPF distribution formats `BLOCK`, `CYCLIC`, and `GEN_BLOCK`.

4.2 Controlling the Work Sharing of Loops

An inter-node mapping explicitly specified by a `SHARE` directive for certain arrays is utilized in order to determine the work sharing of parallel loops or array assignments where such arrays are accessed. The user may override such an automatic strategy by appending a `SHARE` clause to an `INDEPENDENT` directive.

```
!hpf$ nodes N(8)
      real A(1000,1000)
!hpf$ distribute A(block,*) onto N !inter-node mapping (data distribution)
!hpf$ share A(*,block)             !intra-node mapping (work sharing)

...
!hpf$ independent, share(dynamic)
      do J = 1, N
         A(:,J) = f (A(:,J))
      end do
```

Fig. 7. Hierarchical data mappings. The first dimension of `A` is distributed by `block` onto the nodes of an SMP cluster, while for the second dimension of `A` a `block` work sharing strategy is specified by means of the `share` directive. Note that the work sharing information provided for an array in the specification part of a program may be overwritten at individual loops by attaching a `share` clause to an `independent` directive.

An example of a hierarchical mapping is shown in Figure 7. Assuming a homogeneous cluster of 8 nodes, each of which equipped with `NPN` processors, the inter-node mapping specified by means of the `DISTRIBUTE` directive distributes the first dimension of array `A` across the 8 nodes of the cluster, mapping a sub-array of shape (125,1000) to each node. The `SHARE` directive specifies a virtual distribution within each node. As a consequence, a sub-array of shape (125,1000/`NPN`) is mapped to each processor within a node. The `SHARE` clause attached to the `INDEPENDENT` directive specifies that a dynamic scheduling strategy should be adopted for scheduling the loop iterations of the `J`-loop among the threads executing on the processors within nodes. Without the `SHARE` clause a `BLOCK`-scheduling strategy would be adopted according to the specified intra-node mapping.

4.3 Integration of OpenMP Routines

The LOCAL extrinsic model of HPF [10] provides the possibility to write code that is executed on all active processors, where each processor has direct access only to its local data. We introduce a new extrinsic model, called `NODE_LOCAL`, that allows writing code which is executed on all active nodes, where on each node only node-local data according to the inter-node mapping is directly accessible. In `NODE_LOCAL` routines the potential shared-memory parallelism within nodes can be exploited independently of the HPF level.

A `NODE_LOCAL` subprogram can be written in one of the following languages:

- `'HPF'`, referring to the HPF language, where the HPF compiler can take advantage of the fact that only processors on the same node are involved.
- `'FORTRAN'`, referring to the ANSI/ISO standard Fortran language, in which the user can take advantage of thread parallelism, for example, via OpenMP.

```
      extrinsic (MODEL=NODE_LOCAL) subroutine Sub(A)
         real, dimension :: A(:,:)
         integer N

!$OMP    parallel
         ...
      end subroutine
```

Fig. 8. Example of a `NODE_LOCAL` EXTRINSIC routine.

Node-local HPF procedures can use any HPF intrinsic or library procedure. Similar to the HPF Local Routine Library, a new HPF Node Local Routine Library is provided for the use within HPF node-local routines, and a Fortran Node Local Routine Library which realizes the HPF Node Local Routines with a Fortran interface.

Node-local Fortran routines are very similar to local Fortran routines. The only difference is that the programmer now sees in the subprogram only the data mapped to a node instead of the data mapped to a processor. It is the programmers responsibility to exploit parallelism across multiple processors within a node, e.g. via OpenMP or via any other thread library.

5 Related Work

Several researchers have investigated the advantages of a hybrid programming model based on MPI and OpenMP against a unified MPI-only model. Cappelo [5] et al. investigated a hybrid-parallel programming strategy in comparison with a pure message passing approach using the NAS benchmarks on IBM SP systems. In their experiments the MPI-only approach is better than a hybrid strategy for most codes. They conclude that a hybrid-parallel strategy becomes superior when fast processors make the communication performance significant and the level of parallelization is sufficient. Henty [9] reports on experiments with a Discrete Element Modeling code on various SMP clusters. He concludes that current OpenMP implementations are not yet efficient enough for hybrid parallelism to outperform pure message-passing. Haan [8] performed experiments with a matrix-transpose showing that a hybrid-parallel approach can significantly outperform message-passing parallelization.

On the Origin2000, the SGI data placement directives [16] form a vendor specific extension of OpenMP. Some of these extensions have similar functionality as the HPF directives, e.g. "affinity scheduling" of parallel loops is the counterpart to the `ON` clause of HPF. Compaq has also added a new set of directives to its Fortran for Tru64 UNIX that extend the OpenMP Fortran API to control the placement of data in memory and the placement of computations that operate on that data [3].

Chapman, Mehrotra and Zima [6] propose a set of OpenMP extensions, similar to HPF mapping directives, for locality control, but they do not provide a execution model or implementation scheme.

Portland Group, Inc. proposes a high-level programming model [12] that extends the OpenMP API with additional data mapping directives, library routines and environment variables. This model extends OpenMP in order to control data locality with respect to the nodes of SMP clusters. In contrast to this model, HPF, with the extensions proposed in this paper, supports locality control across nodes as well as within nodes.

All these other approaches introduce data mapping features into OpenMP in order to increase locality, but still utilize the explicit work distribution via the `PARALLEL` and `PARDO` directives of OpenMP. Our approach is based on HPF and relies on an implicit work distribution which is usually derived from the data mapping but which may be explicitly controlled by the user within nodes by means of OpenMP-like extensions.

A number of researches have addressed the issues of implementing OpenMP on clusters of SMPs relying on a distributed-shared memory (DSM) software layer. Hu et al. [11] describe the implementation of OpenMP on a network of shared-memory multiprocessors by means of a translating OpenMP directives into calls to a modified version of the TreadMarks software distributed memory system. Sato et al. [15] describe the design of an OpenMP compiler for SMP clusters based on a compiler-directed DSM software layer.

6 Conclusions and Future Work

The concept of processor mappings enables HPF programmers to provide the compiler with an explicit description of the hierarchical structure of SMP clusters. The corresponding directives for the definition of abstract node arrays and the mapping of abstract processor arrays to abstract node arrays can also be considered as implementation-dependent directives that go conform with the current HPF language definition and therefore enable an easy way to port existing HPF programs to SMP clusters.

Additional extensions are provided for the explicit specification of inter-node and intra-node data mappings. These features give users more control over exploiting shared-memory parallelism within a node, by using the `SHARE` directive and `SHARE` clause, or the new `NODE_LOCAL` extrinsic model.

The directives, in either case, do not add significant complexity to the source code. Default processor arrays and node arrays in missing `ONTO` clauses can be chosen in such a way that many HPF programs do not need any modification at all. But the new extensions will be very helpful when the shared-memory parallelism is more advantageous than the distributed-memory parallelism for certain dimensions of the involved arrays.

A complete specification of the syntax and semantics of the proposed language extensions is subject of ongoing work. Major issues are the provision of data mapping mechanisms with respect to subsets of nodes as well as an integration of the new features with the HPF and/or OpenMP tasking model. Moreover, the evaluation of the new execution model in comparison with a pure message-passing model and the investigation of its relevance for a larger range of scientific applications will be addressed in future work.

References

1. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, 1999.
2. S. Benkner and T. Brandes. Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs. In *Euro-Par 2000 Parallel Processing, Munich*, pages 647–657. Lecture Notes in Computer Science (1900), September 2000.
3. J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA Machines. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, Dallas, November 2000.
4. T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, April 1994.
5. F. Cappello and D. Etieble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, Dallas, November 2000.
6. B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with Features for Locality Control. In *Proc. ECWMF Workshop "Towards Teracomputing - The Use of Parallel Processors in Meteorology"*, 1998.
7. E. Dockner, H. Moritsch, G. Ch. Pflug, and A. Swietanowski. AURORA financial management system: From Model Design to Implementation. Technical report AURORA TR1998-08, University of Vienna, June 1998.
8. O. Haan. Matrix Transpose with Hybrid OpenMP / MPI Parallelization. Technical Report Presentation given at SCICOMP 2000, http://www.spscicomp.org/2000/userpres.html#haan, 2000.
9. D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, Dallas, November 2000.
10. High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, January 1997.
11. Y. Hu, H. Lu, A. Cox, and W. Zwaenepel. Openmp for networks of smps. In *Proceedings of IPPS.*, 1999.
12. M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP – A Programming Model for SMP Clusters. In *Eighth International Workshop on Compilers for Parallel Computers*, pages 229–238, Aussois, France, January 2000.
13. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Vers. 1.1, June 1995. MPI-2: Extensions to the Message-Passing Interface, 1997.
14. H. Moritsch and S. Benkner. High Performance Numerical Pricing Methods. In *4-th Intl. HPF Users Group Meeting*, Tokyo, October 2000.
15. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of openmp compiler for an smp cluster. In *Proceedings EWOMP '99, pp.32-39.*, 1999.
16. Silicon Graphics Inc. MIPSpro Power Fortran 77 Programmer's Guide: OpenMP Multriprocessing Directives. Technical Report Document 007-2361-007, 1999.
17. The OpenMP Forum. OpenMP Fortran Application Program Interface. Version 1.1, November 1999. `http://www.openmp.org`.