Exploiting Distributed-Memory and Shared-Memory Parallelism on Clusters of SMPs with Data Parallel Programs

Siegfried Benkner[1] and Viera Sipkova[1]

Received August, 2002; revised October, 2002

Clusters of SMPs are hybrid-parallel architectures that combine the main concepts of distributed-memory and shared-memory parallel machines. Although SMP clusters are widely used in the high performance computing community, there exists no single programming paradigm that allows exploiting the hierarchical structure of these machines. Most parallel applications deployed on SMP clusters are based on MPI, the standard API for distributed-memory parallel programming, and thus may miss a number of optimization opportunities offered by the shared memory available within SMP nodes. In this paper we present extensions to the data parallel programming language HPF and associated compilation techniques for optimizing HPF programs on clusters of SMPs. The proposed extensions enable programmers to control key aspects of distributed-memory and shared-memory parallelization at a high-level of abstraction. Based on these language extensions, a compiler can adopt a hybrid parallelization strategy which closely reflects the hierarchical structure of SMP clusters by automatically exploiting shared-memory parallelism based on OpenMP within cluster nodes and distributed-memory parallelism utilizing MPI across nodes. We describe the implementation of these features in the VFC compiler and present experimental results which show the effectiveness of these techniques.

KEY WORDS: Hybrid parallelization; SMP clusters; HPF; OpenMP; MPI.

[1] Institute for Software Science, University of Vienna, Liechtensteinstrasse 22, A-1090 Vienna, Austria. E-mail: {sigi,sipka}@par.univie.ac.at

0885-7458/03/0200-0003/0 © 2003 Plenum Publishing Corporation

1. INTRODUCTION

Clusters of (symmetric) shared-memory multiprocessors (SMPs) are playing an increasingly important role in the high-performance computing arena. Examples of such systems are multiprocessor clusters from SUN, SGI, IBM, a variety of multi-processor PC clusters, supercomputers like the NEC SX-6 or the Japanese Earth Simulator and the ASCI White machine. SMP clusters are hybrid-parallel architectures that consist of a number of nodes which are connected by a fast interconnection network. Each node contains multiple processors which have access to a shared memory, while the data on other nodes may usually be accessed only by means of explicit message-passing. Most application programs developed for SMP clusters are based on MPI,(1) a standard API for message-passing which has been designed for distributed-memory parallel architectures. 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 with OpenMP,(2) a standardized shared-memory API, in a single application. This strategy aims to fully exploit SMP clusters by relying on data distribution and explicit message-passing between the nodes of a cluster, and on data sharing and multi-threading within nodes.(3–6) Although combining MPI and OpenMP allows optimizing parallel programs by taking into account the hybrid architecture of SMP clusters, applications written in this way tend to become extremely complex.

In contrast to MPI and OpenMP, High Performance Fortran (HPF)(7) is a high-level parallel programming language which can be employed on both distributed-memory and shared-memory machines. HPF programs can also be compiled for clusters of SMPs, but the language does not provide features for directly exploiting their hierarchical structure. Current HPF compilers usually ignore the shared-memory aspect of SMP clusters and treat such machines as pure distributed-memory systems.

In order to optimize HPF for clusters of SMPs, we have extended the mapping mechanisms of HPF by high-level means for controlling the key aspects of distributed and shared-memory parallelization. The concept of *processor mappings* enables the programmer to specify the hierarchical structure of SMP clusters by mapping abstract processor arrays onto abstract node arrays. The concept of *hierarchical data mappings* allows the separate specification of *inter-node data mappings* and *intra-node data mappings*. Furthermore, new intrinsic and library procedures and a new local extrinsic model have been developed. By using node-local extrinsic procedures, hybrid parallel programs may be constructed from OpenMP routines within an outer HPF layer. Based on these extensions, the VFC compiler[8] adopts a hybrid parallelization strategy which closely reflects the hierarchical structure of SMP clusters. VFC compiles an extended HPF program into a hybrid parallel program which exploits shared-memory parallelism within nodes relying on OpenMP and distributed-memory parallelism across nodes utilizing MPI.

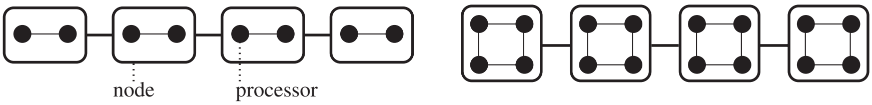
This paper is organized as follows: In Section 2 we describe language extensions for optimizing HPF programs for SMP clusters. In Section 3 we give an overview of the main features of the VFC compiler and outline its parallelization strategy adopted for SMP clusters. Section 4 presents an experimental evaluation of the hybrid parallelization strategy. Section 5 discusses related work, followed by conclusions and a brief outline of future work in Section 6.

2. HPF EXTENSIONS FOR SMP CLUSTERS

HPF has been primarily designed for distributed-memory machines, but can also be employed on shared-memory machines and on clusters. However, HPF lacks features for exploiting the hierarchical structure of SMP clusters. Available HPF compilers usually ignore the shared-memory aspect of SMP clusters and treat such machines as pure distributed-memory systems. These issues have been the main motivation for the development of cluster-specific extensions.

2.1. Abstract Node Arrays and Processor Mappings

HPF offers the concept of abstract processor arrays for defining an abstraction of the underlying parallel architecture. Processor arrays are used as the target of data distribution directives, which specify how data arrays are to be distributed to processor arrays. Although suitable for distributed-memory machines and shared-memory machines, processor arrays are not sufficient for describing the structure of SMP clusters. In order to



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

```
!hpf$ processors Q(2,8)
!hpfC nodes N(4)
!hpfC distribute Q(*,block) onto N
```

Fig. 1. Examples of processor mappings: (left) 4 × 2 SMP cluster, (right) 4 × 4 cluster.

specify the hierarchical topology of SMP clusters we introduce abstract *node arrays* and *processor mappings* (see Fig. 1).

Processor mappings may be specified using the extended DISTRIBUTE directive with a processor array as distributee and a node array (declared by means of the NODES directive) as distribution target. Within processor mappings the HPF distribution formats BLOCK, GEN_BLOCK or "*" may be used. For example, the processor mapping directive DISTRIBUTE P(BLOCK) ONTO N, maps each processor of the abstract processor array P to a node of the abstract node array N according to the semantics of the HPF BLOCK distribution format.

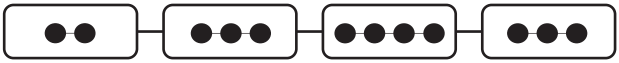
The new intrinsic function NUMBER_OF_NODES is provided in order to support abstract node arrays whose sizes are determined upon start of a program. NUMBER_OF_NODES returns the actual number of nodes used to execute a program while the HPF intrinsic function NUMBER_OF_PROCESSORS returns the total number of processors in a cluster. Using these intrinsic functions, programs may be parallelized regardless of the actual number of nodes and processors per nodes.

2.1.1. Heterogeneous Clusters

While for homogeneous clusters the BLOCK distribution format is used in processor mappings, heterogeneous clusters, e.g., clusters where the number of processors per node varies, can be supported by means of the GEN_BLOCK distribution format of the Approved Extensions of HPF. Figure 2 shows 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 specify that the number of processors within nodes varies.

2.1.2. Semantics of Processor Mappings

A processor mapping specifies for each processor array dimension whether distributed-memory parallelism, shared-memory parallelism or both may be exploited according to the following rules:



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

Fig. 2. Specification of a heterogeneous SMP cluster using the GEN_BLOCK distribution format within a processor mapping directive.

(1) If a dimension of a processor array is distributed by BLOCK or GEN_BLOCK, contiguous blocks of processors are mapped to the nodes in the corresponding dimension of the specified node array. As a consequence, both distributed-memory parallelism and shared-memory parallelism may be exploited for all array dimensions that are mapped to a distributed processor array dimension.

(2) If for a dimension of a processor array a "*" is specified as distribution format, only shared-memory parallelism may be exploited across array dimensions that are mapped to that processor array dimension.

For example, in Fig. 1(b) only shared-memory parallelism may be exploited across the first dimension of Q, while both shared-memory and distributed-memory parallelism may be exploited across the second dimension.

By combining data distributions and processor mappings, *inter-node mappings* and *intra-node mappings* can be derived by the compiler.

An inter-node mapping determines for each node those parts of A that are owned by it. 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. Inter-node mappings are used by the compiler to control distributed-memory parallelization, i.e., data distribution and communication across nodes.

An intra-node mapping determines a mapping of the local part of an array assigned to a node of a cluster with respect to the processors within the node. Intra-node mappings are utilized by the compiler to organize shared-memory parallelization, i.e., work scheduling (work sharing) across concurrent threads within nodes. These issues are described in more detail in Section 3.

2.2. Hierarchical Data Mappings

In this section we describe additional extensions which allow users to specify hierarchical mappings for data arrays. A hierarchical data mapping comprises an *explicit inter-node mapping* and an *explicit intra-node mapping*, each specified by a separate directive. Compared to the basic concept of processor mappings as described previously, hierarchical data mappings provide a more flexible mechanism for the distribution of data on clusters of SMPs. However, in order to take advantage of this increased flexibility, it will be usually necessary to modify the mapping directives of existing HPF programs.

2.2.1. Explicit Inter-Node Data Mappings

In order to specify a mapping of data arrays to the nodes of an SMP cluster, the DISTRIBUTE directive is extended by allowing node arrays to appear as distribution target. Such a mapping is referred to as *explicit inter-node mapping*. It maps data arrays to abstract nodes in exactly the same way as an original HPF distribute directive maps data arrays to abstract processors. Inter-node mappings are utilized by the compiler to organize distributed-memory parallelization, i.e., data distribution and communication across nodes. In the following example, array A is mapped to an abstract node array N.

```
!hpfC nodes N(2,2)
      real, dimension (8,8) :: A
!hpfC distribute A (block,block) onto N ! inter-node mapping
```

As a consequence of the extended distribute directive, the section A(1:4, 1:4) is mapped to node N(1, 1), A(5:8, 1:4) to N(2, 1), A(1:4, 5:8) to N(1, 2), and A(5:8, 5:8) is mapped to node N(2, 2).

2.2.2. Explicit Intra-Node Data Mappings

In order to specify a mapping of node-local data with respect to the processors within a node, the SHARE directive has been introduced. A mapping defined by the SHARE directive is referred to as *explicit intra-node data mapping*. As the name of the directive suggests, an intra-node mapping controls the work sharing (scheduling) of threads running within nodes. Besides the usual HPF distribution formats BLOCK, CYCLIC, and GEN_BLOCK, the OpenMP work-sharing formats DYNAMIC and GUIDED may be employed for this purpose.

The information provided by means of the share directive is propagated by the compiler to parallel loops. Various code transformations ensure that loops can be executed by multiple threads which are scheduled according to the specified work sharing strategy.

Hierarchical data mappings may be specified regardless of whether a processor mapping has been specified or not. For example, in the code fragment

```
!hpfC nodes N(4)
      real, dimension (32,16) :: A
!hpfC distribute A(*, block) onto N ! inter-node mapping
!hpfC share A (block,*)             ! intra-node mapping

!hpf$ independent
  do i=1, 32
    a(i,:)=...
  end do
```

the extended distribute directive specifies that the second dimension of A is distributed by BLOCK to the nodes of a cluster. The SHARE directive specifies that computations along the first dimension of A should be performed in parallel by multiple threads under a BLOCK work-scheduling strategy. If we assume that each node is equipped with four processors, the loop iteration space would be partitioned into four blocks of eight iterations and each block of iterations would be executed by a separate thread.

Note that although in some cases a hierarchical mapping can also be expressed by means of a usual HPF data distribution directive and a processor mapping, this is not true in general.

3. HYBRID PARALLELIZATION STRATEGY

In this section we outline how HPF programs that make use of the proposed extensions are compiled with the VFC compiler for clusters of SMPs according to a hybrid parallelization strategy that combines MPI and OpenMP.

3.1. Overview of VFC

VFC(8) is a source-to-source parallelization system which translates HPF+ programs into explicitly parallel programs for a variety of parallel target architectures. HPF+(9) is an extension of HPF with special support for an efficient handling of irregular codes. In addition to the basic features of HPF, it includes generalized block distributions and indirect distributions, dynamic data redistribution, language features for *communication schedule reuse*(10) and the *halo concept*(11) for controlling irregular non-local data access patterns. VFC provides powerful parallelization strategies for a large class of non-perfectly nested loops with irregular runtime-dependent access patterns which are common in industrial codes. In this context, the concepts of communication schedule reuse and halos allow the user to minimize the potentially large overhead of associated runtime compilation strategies.

Initially, VFC has been developed for distributed-memory parallel machines. For such machines, VFC translates HPF programs into explicitly parallel, single-program multiple-data (SPMD) Fortran 90/MPI message-passing programs. Under the distributed-memory execution model, the generated SPMD program is executed by a set of processes, each executing the same program in its local address space. Usually there is a one-to-one mapping of abstract processors to MPI processes. Each processor only allocates those parts of distributed arrays that have been mapped to it according to the user-specified data distribution. Scalar data and data without mapping directives are allocated on each processor. Work distribution (i.e., distribution of computations) is mainly based on the owner-computes rule. Access to data located on other processors is realized by means of message-passing (MPI) communication. VFC ensures that all processors executing the target program follow the same control flow in a loosely synchronous style.

In the following we focus on the extensions of VFC for clusters of SMPs.

3.2. Hybrid Execution Model

The parallelization of HPF+ programs with cluster-specific extensions relies on a hybrid-parallel execution model. As opposed to the usual HPF compilation where a single-threaded SPMD node program is generated, a multi-threaded SPMD node program is generated under the hybrid execution model.

Under the hybrid model, an HPF program is executed on an SMP cluster by a set of parallel processes, each of which runs usually on a separate node. Each process allocates data it owns in shared memory, according to the derived (or explicitly specified) inter-node data mapping. Work distribution across node processes is usually realized by applying the owner computes strategy, which implies that each process performs only computations on data elements owned by it. Communication across node processes is realized by means of appropriate MPI message-passing primitives. In order to exploit additional shared-memory parallelism, each MPI node process generates a set of OpenMP threads which run concurrently in the shared address space of a node. Usually the number of threads spawned within node processes is equal to the number of processors available within nodes. Data mapped to a node is allocated in a non-partitioned way in shared memory, regardless of intra-node mappings. Intra-node data mappings are however utilized to organize parallel execution of threads by applying code transformations and inserting appropriate OpenMP directives.

3.3. Outline of the Hybrid Parallelization Strategy

The parallelization of extended HPF programs for clusters of SMPs can be conceptually divided into three main phases (1) inter-node and intra-node mapping analysis, (2) distributed-memory parallelization, and (3) shared-memory parallelization.

3.3.1. Deriving Inter-Node and Intra-Node Data Mappings

After the conventional front-end phases, the VFC compiler analyzes the distribution directives and processor mapping directives. As a result of this analysis, each dimension of a distributed array is classified as DM, SM, DM/SM, or SEQ, depending on the type of parallelism that may be exploited. Then for each array dimension an inter-node data mapping and an intra-node data mapping is determined.

Assuming the following declarations

```
!hpf$ processors P(4)               ! usual HPF processor array
!hpfC nodes N(2)                    ! abstract node array
!hpfC distribute P(block) onto N ! processor mapping
      real, dimension (100) :: A
!hpf$ distribute A(block) onto P ! usual HPF distribution
```

the inter-node and intra-node mapping of A derived by the compiler are equivalent to those explicitly specified by the following directives:

```
!hpfC distribute A(block) onto N ! "derived inter-node mapping"
!hpfC share A(block)               ! "derived intra-node mapping"
```

As a consequence, both distributed and shared-memory parallelism will be exploited for array A.

On the basis of inter-node and intra-node mappings, the ownership of data both with respect to processes (nodes), and with respect to threads (processors within nodes) is derived and represented in symbolic form. Ownership information is then propagated to all executable statements and, at an intermediate code level, represented by means of ON_HOME clauses which are generated for assignment statements and loops accessing distributed arrays. Each loop is then classified as DM, SM, DM/SM or SEQ depending on the classification of the corresponding array dimension in the associated ON_HOME clause.

3.3.2. Distributed-Memory Parallelization

During the distributed-memory parallelization phase VFC generates an intermediate SPMD message-passing program based on inter-node data mappings. Array declarations are modified in such a way that each MPI process allocates only those parts of distributed arrays that are owned by it according to the inter-node mapping. Work distribution is realized by strip-mining all loops which have been classified as DM across multiple MPI processes. If access to non-local data is required, appropriate message-passing communication primitives are generated and temporary data objects for storing non-local data are introduced. Several communication optimizations are applied, including elimination of redundant communication, extraction of communication from loops, message vectorization, and the use of collective communication instead of point-to-point communication primitives.

The intermediate SPMD message-passing program generated after this phase could already be executed, however it would exploit only a single processor on each node of the cluster.

3.3.3. Shared-Memory Parallelization

The intermediate message-passing program is parallelized for shared memory according to the intra-node data mapping derived by VFC. The shared-memory parallelization phase makes use of OpenMP 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 inserting corresponding OpenMP work-sharing directives and/or appropriate loop transformations (e.g., strip-mining). In this context, variables which have been specified as NEW in an HPF INDEPENDENT directive are specified as PRIVATE within the generated OpenMP directives.

Consistency of shared data objects is enforced by inserting OpenMP synchronization primitives (critical sections or atomic directives).

Furthermore, various optimizations are performed in order to avoid unnecessary synchronization. The potential overheads of spawning teams of parallel threads is reduced by merging parallel regions. Most of these optimization steps are conceptually similar to the communication optimizations performed during distributed-memory parallelization. Special optimizations are applied to loops performing irregular reductions on arrays for which a halo has been specified In order to minimize synchronization overheads for loops that perform irregular reductions on arrays for which a halo has been specified, special optimization techniques are applied.(12)

Note that our current implementation of the hybrid parallelization strategy ensures that only the master thread performs MPI communication.

In Figs. 3 to 5 the hybrid parallelization strategy as realized by VFC is sketched. Figure 3 shows the original HPF code, Fig. 4 sketches the

```
!hpf$ processors P(number_of_processors())
!hpfC nodes N(number_of_nodes())            ! abstract node array
!hpfC distribute P(block) onto N            ! processor mapping
      real, dimension (N) :: A
!hpf$ distribute A(block) onto P
      ...
!hpf$ independent
      do i = NL, NU
         a(i) = ... a(...) ...
      end do
```

Fig. 3. Original HPF program fragment with cluster-specific extensions (!hpfC)

```
      ...
!hpfC share A(block)                        ! derived inter-node mapping
      real, allocatable :: A(:)
      type(rt_dsc), pointer :: A_dsc        ! runtime descriptor of A
      ...                                   ! set up runtime descriptors
      allocate(A(vhpf_extent(A_dsc,1)))     ! allocate node-local part of A
      ...                                   ! compute node-local bounds
      call vhpf_loc_bounds_SM(A_dsc,NL,NU,lb_DM,ub_DM)
      call vhpf_comm(...)                   ! perform MPI communication
      do i = lb_DM, ub_DM                   ! process-local iterations
         a(i) = ...
      end do
      ...
```

Fig. 4. Intermediate (pseudo-)code after mapping analysis and DM parallelization for the program fragment shown in Fig. 3.

```
      ...
      real, allocatable :: A(:)
      type(rt_dsc), pointer :: A_dsc        ! runtime descriptor of A
      ...                                   ! set up runtime descriptors
      allocate(A(vhpf_extent(A_dsc,1)))     ! allocate node-local part of A
      ...
      call vhpf_comm(...)                   ! perform MPI communication
      ...                                   ! spawn parallel threads
!$omp parallel, private(i,A_loc_lb_SM,A_loc_ub_SM)
                                            ! compute thread-local bounds
      call vhpf_loc_bounds_SM(A_dsc,lb_DM,ub_DM,lb_SM,ub_SM)
      do i = lb_SM, ub_SM                   ! thread-local iterations
         a(i) = ...
      end do
!$omp end parallel
      ...
```

Fig. 5. Final (pseudo-)code after DM and SM parallelization (OpenMP) for the program fragment shown in Fig. 3.

intermediate message-passing program obtained after DM parallelization, and Fig. 5 the final (pseudo-)code after DM and SM parallelization.

As shown in Figs. 4 and 5, during shared-memory parallelization the intermediate 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. Inside this parallel region the runtime routine vhpf_loc_bounds_SM computes from the iterations assigned to a node (i.e., the iterations from lb_DM to ub_DM) the chunk of iterations executed on each individual thread according to the derived intra-node data mapping.

3.3.4. Potential Advantages of Hybrid Parallelization

The hybrid parallelization strategy offers a number of advantages compared to the usual distributed-memory parallelization strategy as realized by most HPF compilers. The hybrid strategy reflects closely the topology of SMP clusters by exploiting distributed-memory parallelism across the nodes and shared-memory parallelism within nodes. It allows a direct utilization of the shared memory within nodes and usually requires less total memory than the DM parallelization strategy. For example, replicated data arrays, which are mostly accessed in a read only way, have to be allocated only once per node, while in the distributed-memory model replicated arrays have to be allocated by each process, resulting in multiple copies on each node.

Moreover, the hybrid model usually allows a more efficient handling of communication. Since data is distributed only across nodes and communication is performed only by the master thread, the hybrid model results in less messages as well as larger messages. However, on certain architectures, such a strategy, where only the master thread performs communication, may reduce the overall communication bandwidth.

Another important issue are unstructured computations, where data arrays are accessed indirectly by means of array subscripts. Under the distributed-memory model, an HPF compiler has to apply expensive runtime techniques to parallelize loops with indirect array accesses. However, if array dimensions which are accessed indirectly are mapped to the processors within a node, the overheads of runtime parallelization can be avoided due to the shared access.

Despite these differences both models usually result in the same degree of parallelism. Thus, for many applications only minor performance differences can be observed. In particular, this is true for codes which are characterized by a high degree of locality and independent computations.

In the following we present an experimental evaluation of the new language extensions and the hybrid parallelization strategy as provided by VFC using two benchmark codes on a Beowulf-type SMP PC cluster.

4. EXPERIMENTAL RESULTS

For the performance experiments, we used a kernel from a crash-simulation code which originally has been developed for HPF+, and kernel from a numerical pricing module(13) developed in the context of the AURORA Financial Management System.(14) Both kernels are based on an iterative computational scheme with an outer time-step loop. The main computational parts are performed in nested loops where large arrays are accessed by means of vector subscripts. In order to minimize the overheads that would be caused by usual runtime parallelization strategies (e.g., inspector/executor), non-local data access patterns are explicitly specified at runtime based on the concept of halos. Moreover, in both kernels the reuse of runtime generated communication schedules for indirectly accessed arrays is enforced by means of appropriate language features for communication schedule reuse.(10, 8)

Both kernels have been parallelized with the VFC compiler(8) and executed on a Beowulf cluster consisting of 16 nodes connected via Myrinet. Each node is equipped with four Pentium III Xeon processors (700 MHz) and 2GB RAM.

For both kernels we compared a pure MPI version to a hybrid-parallel version based on MPI/OpenMP. Both versions have been generated from an HPF+ source program, in the latter case, with additional cluster-specific extensions. The pgf90 compiler from Portland Group Inc. which also supports OpenMP has been used as a backend-compiler of VFC.

All kernels have been measured on up to 16 nodes, where on each node either four MPI processes (MPI only) or four OpenMP threads (MPI/OpenMP) were run. Speed-up numbers have been computed with respect to the sequential version of the code (i.e., HPF compiled with the Fortran 90 compiler).

In Fig. 6 the speedup curves of the financical optimization kernel are shown on the left hand side. For up to 8 processors (2 nodes) the pure MPI version and the hybrid MPI/OpenMP version are almost identical. However, on more than 8 processors the MPI/OpenMP version becomes superior. The main reason for this performance difference is that the computation/communication ratio of the pure MPI version decreases faster than for the MPI/OpenMP version. In the pure MPI version the total number of messages is approximately four times larger than in the MPI/OpenMP version. Also the overall memory requirements of the pure

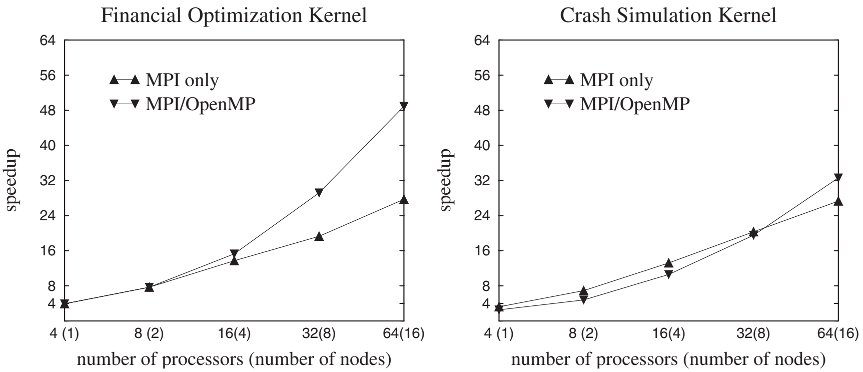


Fig. 6. Performance comparison of kernels parallelized with VFC under the distributed-memory parallelization strategy (MPI only) and under the hybrid parallelization strategy (MPI/OpenMP).

MPI version are higher since the kernel contains several replicated arrays which are allocated in the pure MPI version in each process, resulting in four copies per node, while in the MPI/OpenMP version only one copy per node is required.

For the crash simulation kernel (see Fig. 6, right hand side) the situation is similar. Here the pure MPI version is superior on up to 32 processors (8 nodes). On 64 processors, however, the MPI/OpenMP version achieves a better speedup.

Note that for both kernels we could not achieve any speedups with commercial HPF compilers due to an inadequate handling of loops with irregular (vector-subscripted) data accesses.

5. RELATED WORK

Some of the extensions for SMP clusters described in this paper have been implemented also in the ADAPTOR compilation system.(16, 15) ADAPTOR supports also the automatic generation of hybrid-parallel programs from HPF based on appropriate default conventions for processor mappings.

Several researchers have investigated the advantages of a hybrid programming model based on MPI and OpenMP against a unified MPI-only model. Cappelo *et al.*(3) 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 has provided better results 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(17) 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(4) 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(18) 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.(19) Chapman, Mehrotra, and Zima(20) have proposed a set of OpenMP extensions, similar to HPF mapping directives, for locality control. PGI proposes a high-level programming model(21, 22) 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 control data 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 studies have addressed the issues of implementing OpenMP on clusters of SMPs relying on a distributed-shared memory (DSM) software layer. Hu *et al.*(23) 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.*(24) describe the design of an OpenMP compiler for SMP clusters based on a compiler-directed DSM software layer.

6. CONCLUSIONS

Processor mappings provide a simple but convenient means for adapting existing HPF programs with minimal changes for clusters of SMPs.

Usually only a node array declaration and a processor mapping directive have to be added to an HPF program. Based on a processor mapping, an HPF compiler can adopt a hybrid parallelization strategy that exploits distributed-memory parallelism across nodes, and shared-memory parallelism within nodes, closely reflecting the hierarchical structure of 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 the shared-memory parallelization within nodes, by using the SHARE directive.

As our experimental evaluation has shown, using these features performance improvements for parallel programs on clusters of SMPs can be achieved in comparison to a pure message-passing parallelization strategy. Another potential advantage is that with the hybrid-parallelization strategy shared memory within nodes can be exploited directly, often resulting in lower memory requirements.

For the future we plan to extend the hybrid compilation strategy of VFC by relaxing the current restriction that only the master threads on each process can perform MPI communication. This implies that a thread-safe MPI implementation supporting the features of MPI-2 for thread-based parallelization (cf. Section 8.7 of the MPI-2 Specification) must be available on the target SMP cluster. However, currently this is not the case on most clusters.

ACKNOWLEDGMENTS

This work was supported by the Special Research Program SFB F011 “AURORA” of the Austrian Science Fund and by NEC Europe Ltd. as part of the NEC/Univ. Vienna ADVANCE project in co-operation with NEC C&C Research Laboratories.

REFERENCES

1. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Vers. 1.1, June 1995, MPI-2: Extensions to the Message-Passing Interface, 1997.
2. The OpenMP Forum, OpenMP Fortran Application Program Interface, Version 1.1, November 1999, http://www.openmp.org.
3. 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).
4. O. Haan, Matrix Transpose with Hybrid OpenMP/MPI Parallelization. Technical Report, http://www.spscicomp.org/2000/userpres.html#haan, 2000.
5. R. D. Loft, S. J. Thomas, and J. M. Dennis, Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models. In *Proceedings SC2001*, Denver (November 2001).

6. P. V. Luong, C. P. Breshears, and L. N. Ly, Costal Ocean Modeling of the U.S. West Coast with Multiblock Grid and Dual-Level Parallelism. In *Proceedings of SC2001*, Denver (November 2001).
7. High Performance Fortran Forum, High Performance Fortran Language Specification, Version 2.0, Department of Computer Science, Rice University (1997).
8. S. Benkner, VFC: The Vienna Fortran Compiler, *Scientific Programming*, **7**(1):67–81 (1999).
9. S. Benkner, HPF+-High Performance Fortran for Advanced Scientific and Engineering Applications, Future Generation Computer Systems, Vol. 15 (3) (1999).
10. S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima, High-Level Management of Communication Schedules in HPF-like Languages, In *Proceedings of the International Conference on Supercomputing* (ICS'98), pp. 109–116, Melbourne, Australia, ACM Press (July 13–17, 1998).
11. S. Benkner, Optimizing Irregular HPF Applications Using Halos, *Concurrency: Practice and Experience*, Wiley (2000).
12. S. Benkner and T. Brandes, Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs, In *Euro-Par 2000 Parallel Processing*, Lecture Notes in Computer Science 1900, Munich, Germany (September 2000).
13. H. Moritsch and S. Benkner, High Performance Numerical Pricing Methods, In *Fourth Int'l. HPF Users Group Meeting*, Tokyo (October 2000).
14. 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).
15. S. Benkner and T. Brandes. High-Level Data Mapping for Clusters of SMPs, In *Proceedings 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, San Francisco, Springer-Verlag (April 2001).
16. T. Brandes and F. Zimmermann, ADAPTOR—A Transformation Tool for HPF Programs, In K. M. Decker and R. M. Rehmann (eds.), *Programming Environments for Massively Parallel Distributed Systems*, Birkhäuser Verlag (1994).
17. 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).
18. Silicon Graphics Inc. MIPSpro Power Fortran 77 Programmer's Guide: OpenMP Multiprocessing Directives, Technical Report Document 007-2361-007 (1999).
19. 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).
20. 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).
21. 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*, pp. 229–238, Aussois, France (January 2000).
22. J. Merlin, D. Miles, and V. Schuster, Extensions to OpenMP for SMP Clusters. In *Proceedings of the Second European Workshop on OpenMP*, EWOMP (2000).
23. Y. Hu, H. Lu, A. Cox, and W. Zwaenepel, Openmp for networks of smps, In *Proceedings of IPPS*. (1999).
24. 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).