Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs*

Siegfried Benkner[1] and Thomas Brandes[2]

[1] Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
sigi@ieee.org
[2] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
brandes@gmd.de

Abstract. OpenMP offers a high-level interface for parallel programming on scalable shared memory (SMP) architectures providing the user with simple work-sharing directives while relying on the compiler to generate parallel programs based on thread parallelism. However, the lack of language features for exploiting data locality often results in poor performance since the non-uniform memory access times on scalable SMP machines cannot be neglected. HPF, the de-facto standard for data parallel programming, offers a rich set of data distribution directives in order to exploit data locality, but has mainly been targeted towards distributed memory machines. In this paper we describe an optimized execution model for HPF programs on SMP machines that avails itself with the mechanisms provided by OpenMP for work sharing and thread parallelism while exploiting data locality based on user-specified distribution directives. This execution model has been implemented in the ADAPTOR HPF compilation system and experimental results verify the efficiency of the chosen approach.

1 Introduction

There is now an emerging class of multiprocessor architectures with scalable hardware support for cache coherence. These are generally referred to as (scalable) Shared Memory Multiprocessor (SMP) architectures. Most of these these machines are built via physically distributed memory (ccNUMA) resulting in non-uniform memory access times and therefore the exploitation of data locality is a crucial issue for many applications.

The OpenMP Application Programming Interface [13] is intended as a portable shared memory programming model to be used on SMP architectures. It defines a set of program directives and a library for runtime support that augment standard C/C++ and Fortran 77/90. OpenMP is based on thread parallelism

* The work described in this paper was supported by NEC Europe Ltd. as part of the ADVICE project in cooperation with the NEC C&C Research Laboratories.

allowing users to exploit shared memory parallelism at a reasonable coarse level. But OpenMP does not provide any directives for controlling the locality of data. As a consequence, the user is responsible for achieving high data locality by enforcing an appropriate work and data distribution which results in a significantly higher programming effort.

High Performance Fortran (HPF) [8] is a well established language extension of Fortran supporting the data parallel programming model. While Fortran array statements and the `FORALL` statement are already natural ways of specifying data parallel computations, HPF provides additional directives to assert independent computations and to advise the compiler how to assign array elements to processor memories in order to reduce data movements on machines with Non-Uniform-Memory-Access (NUMA). The HPF mapping directives define a mapping of data objects (arrays) to abstract processors. Data objects that have been mapped to a certain abstract processor are said to be *owned* by that processor. *Ownership* of data is the central concept for the execution of HPF programs. Based on the ownership of data (owner-computes rule), the distribution of computations to the abstract processors and the necessary communication and synchronization between processors is derived automatically. Up to now, the compilation and execution of HPF programs is mainly considered for distributed memory (DM) machines based on a DM execution model, illustrated in Figure 1(b). In this model, the parallel program generated by the compiler is executed by a set of (abstract) processors where each processor executes the same program in its local address space operating only on its own data. Any two processors communicate by exchanging messages. In accordance with the SPMD (single-program-multiple-data) paradigm, a HPF compiler has to ensure that all processors executing the target program follow the same control flow in a loosely synchronous style. Scalar data and data without mapping directives are allocated on each processor, i.e. replicated.

In this paper, we present and discuss a highly-efficient execution model for SMP architectures. It is illustrated in Figure 1(c). In contrary to the DM execution model that can also be emulated on SMP machines, it takes advantage of the global address space provided on these machines in order to generate more efficient code. While the work distribution implied by the mapping directives remains the same as in the DM model, the model uses thread parallelism instead of process parallelism and keeps the global layout of the mapped data in the global address space to reduce the overhead for non-local data accesses. Scalar data and data without mapping directives have only one incarnation in the shared memory in order to reduce memory overheads.

The SM execution model described in this paper has been implemented within the public domain HPF compilation system ADAPTOR [1]. This compilation systems already supported the DM execution model for a longer time. It has been redesigned to support both execution models in such a way that most of the compiler modules can be exploited for both models. The user can select the execution model by specifying a flag [5]. For the SM execution model, ADAPTOR generates Fortran code with embedded shared memory parallelization directives

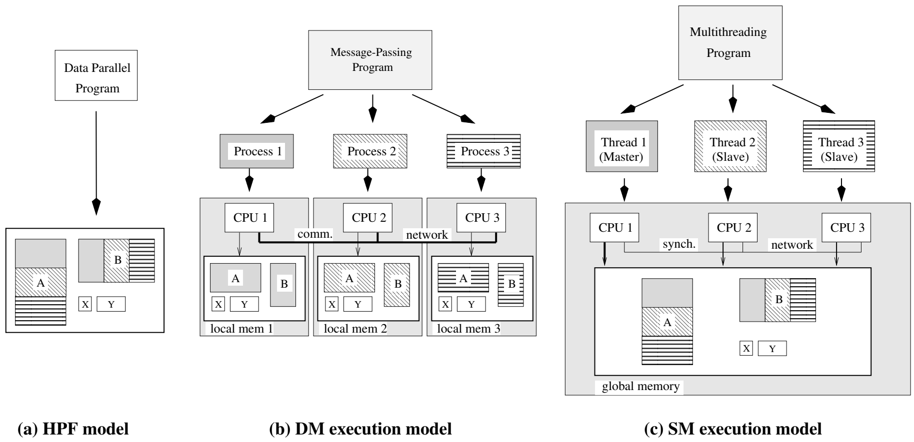

Fig. 1. HPF execution model for distributed and shared memory machines.

exploiting thread parallelism (e.g. OpenMP, NEC/SX or SGI Fortran directives). Additional runtime support is available to support data locality.

The experimental results for some applications show that on SMP architectures the SM execution model is more efficient than the DM execution model.

Related Work

On the Origin2000, the SGI data placement directives [12] form a vendor specific extension of OpenMP. Some of the directives have similar functionality as the HPF directives, e.g. the "affinity scheduling" of parallel loops is the counterpart to the `ON` clause of HPF.

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

Portland Group, Inc. proposes a new high-level programming model [9] that extends the OpenMP API with additional data mapping directives, library routines and environment variables. This model offers more capabilities to direct locality of data and is also applicable for distributed memory systems and SMP clusters. In contrary to this model, we rely on a uniform data parallel programming model suited for all kinds of architectures.

2 Thread Parallelism vs. Process Parallelism

For the DM execution model, the HPF compiler generates a program to be executed by a set of processes where each process executes the same program in its local address space. This corresponds to the Single-Program-Multiple-Data (SPMD) paradigm that is also used in the message passing programming model.

All processes follow the same control flow in a loosely synchronous style. Usually there is a one-to-one mapping of abstract processors (declared at the HPF level by means of processors directives) to processes and each process is executed on an individual processor of the parallel target architecture.

Light-weighted processes (threads) are based on the idea of separating the characteristics of program execution (program counter, stack) from system resources (memory, table of open files). One process can split up in many threads that have their own stack and program counter, but share all the other system resources. Operations on threads (creation, context switching) are executed at least one order of magnitude faster than the corresponding operations on processes. Threads are the first choice when using the master/slave paradigm that is also followed within the OpenMP programming model.

For our SM execution model we chose to generate programs based on thread parallelism. The main reason for this decision is the fact that the global address space is available by default and has not to be allocated via special system calls. Nevertheless, we follow the SPMD paradigm as far as possible to avoid the overhead caused by the creation/termination of threads. Attention has been paid to efficient memory allocation during parallel execution as it must be thread-safe.

3 Data Mapping and Data Layout

For the SM execution model, we changed the default mapping strategy for scalar data and non-mapped data. While in the DM model every processor is an owner and gets an own copy of this data, we have now only one incarnation in the global address space that will be owned by a dedicated processor (master thread). On SMP systems, the conflict between memory overhead and data movement is better solved by reducing memory overhead.

For the mapped data, the HPF mapping directives define a mapping data to the abstract processors. This mapping defines the ownership of data. In the DM execution model, mapped data objects are allocated in a partitioned manner, such that each processor only allocates those parts of a data object that are *owned* by it. The part of a distributed array owned by a processor is referred to as its *local section*. Since the size of the local section of a distributed array on a particular processor usually cannot be determined at compile time, a dynamic allocation strategy has to be adopted. Since each processor allocates only a subsection of a distributed array, global addresses of distributed arrays have to be translated into local addresses. In particular for cyclic and indirect distributions, these additional local address calculations may add significant overhead.

On shared memory machines, the availability of a global address space obviates the need to allocate mapped HPF data objects in a partitioned way. As a consequence, the original global Fortran layout can be left unchanged. The global address space offers several advantages regarding the execution of data parallel programs: translation of global addresses to local addresses is superfluous and remapping of data changes only the ownership of data but does not imply any reallocation of data or data movements in memory.

In certain situations, however, the global data layout might decrease the cache performance due to false sharing. In such cases, a reorganization of distributed arrays such that all array elements belonging to the same processor are stored contiguously might be more efficient. We provide a new directive to select this data layout for mapped arrays.

4 Work Distribution

Parallel execution in the HPF execution model is achieved by distributing the computations to the processors. This task, referred to as work distribution, is usually based on the owner-computes rule where an assignment to an element of a distributed array is executed by the processor that owns this element. Thus, work distribution is derived automatically from the data distribution. Not in all situations the owner computes rule is the best strategy for work distribution. Therefore alternative strategies may be adopted by an HPF compiler. Moreover, HPF-2 provides the `ON` clause that allows the user to explicitly control work distributions.

For the SM execution model, we propose the same work distribution strategy as in the DM execution model, i.e. the work sharing of parallel loops is derived automatically from the data distribution of arrays accessed within the loop and implemented by means of appropriate OpenMP work sharing constructs. There is only one difference due to the changed default allocation strategy for scalars and non-mapped arrays. Assignments to such data objects are no longer executed by all processors (replicated) but only by one dedicated processor (master thread).

5 Communication and Synchronization

By analyzing the data distribution and work distribution, non-local data accesses can be determined. These non-local data accesses imply data movements between the involved processors.

In the DM execution model, non-local accesses are implemented by means of message-passing. As a consequence, each processor has to determine both the data to be sent to other processors and the data to be received from other processors. Since all cross-processor dependencies are satisfied in this manner, no explicit synchronization is necessary. In the context of parallel loops, communication is extracted from the loop body combining single-element messages into larger messages whenever possible in order to minimize latency (message vectorization). In case of indirect addressing, an inspector/executor strategy [11] should be exploited to derive a communication schedule. Temporary arrays and buffers become necessary to keep non-local data. The introduction of shadow edges reduces this memory overhead and simplifies the generated code.

In the SM execution model, a thread can access non-local data directly via the global address space. Only appropriate synchronization becomes necessary to ensure that the correct version of the shared data is accessed. This synchronization can be realized either by point-to-point synchronization between the accessing and the owning processor or by global synchronization via barriers. Synchronization is extracted from parallel loops utilizing similar techniques as applied for communication optimization in the DM model. The easiest way for avoiding point-to-point communication within loops is to insert a barrier before and after independent computations (e.g. see Figure 2). Temporary arrays to store copies of non-local data and expensive computations of communication schedules are no longer necessary in this model.

Independent computations accessing only local data do not require any synchronization. The `RESIDENT` directive of HPF allows to avoid redundant synchronization in the SM model, in the same way as it is utilized in the DM model to avoid redundant communication.

```
      real, dimension (N) :: A, B
!hpf$ distribute (block)  :: A, B


      ...
      B(2:N) = A(1:N-2)+A(3:N)
      B = B - A
      A = A + B
      ...
```

(a) original HPF code.

```
!$omp parallel, private (I,LB,UB)
      LB = ...; UB = ...! local range
      ...
!$omp barrier
      do I = max(LB,2),min(UB,N-1)
         B(I) = A(I-1)+A(I+1)
      end do
!$omp barrier
      do I = LB, UB
         B(I) = B(I) - A(I)
         A(I) = A(I) + B(I)
      end do
      ...
!$omp end parallel
```

(b) generated OpenMP code.

Fig. 2. Synchronization of non-local accesses.

6 Private and Reduction Variables

Private variables specified by the `NEW` clause of HPF become private data of the threads in the SM execution model. Thus every thread will get its own local incarnation of the variable. In the DM execution model, every process has its own incarnation by default and the `NEW` clause only guarantees that the different incarnations need not to be consistent and can therefore have different values.

Reduction variables specified by the `REDUCTION` clause of HPF are treated like private reductions when they are not mapped. Every processor gets an own copy and the values are accumulated at the end of the independent computations. This is the same strategy for both models, only the accumulation is handled differently, either by message passing via a global reduction or by accumulating the result to the shared incarnation within a critical section.

Reduction variables that are mapped arrays are treated differently. In the DM execution model, the processors allocate non-local copies for those items of the reduction array that they update. The non-local copies will be accumulated after the independent computation. In the SM execution model, mapped reduction arrays will have only one shared incarnation where the reductions on it are synchronized by locks to ensure that a specific memory location is updated atomically and not accessed simultaneously by multiple writing threads.

In order to minimize synchronization overheads for the reduction array, the concept of *exclusive ownership* has been introduced [3]. An element of the reduction array is exclusively owned by an abstract processor (thread) if it is owned by that processor and not updated by any other processor. Synchronization is only necessary for those elements of the reduction array that are not exclusively owned while exclusively owned elements can be handled like private data requiring no synchronization. The concept of exclusive ownership is especially important for unstructured reductions that are employed in many scientific applications to implement computations on unstructured meshes or sparse matrices.

7 Experiments and Results

ADI-Kernel with Redistributions

The three-dimensional ADI kernel is a data parallel HPF program which utilizes five three-dimensional arrays of size $64 \times 64 \times 64$. The HPF code uses dynamic arrays and redistributes the five arrays from `(*,*,BLOCK)` to `(*,BLOCK,*)` and vice versa for every of the 10 iterations. Only the redistribution requires communication in the DM execution model. Table 1 shows the performance results of the ADI kernel on the NEC SX-4. It compares the data parallel HPF code compiled by ADAPTOR for the different execution models. The DM execution model shows some small speed-ups, but the gain of parallelism in the computations is mainly lost by the time needed for the redistributions. The SM model utilizes a shared global layout of the arrays, therefore the redistributions only require a synchronization, but no communication and no data movement in memory.

	NP = 1	NP = 2	NP = 4	NP = 8
`HPF-DM`	1.12 s	1.07 s	0.85 s	0.80 s
`HPF-SM`	1.14 s	0.57 s	0.30 s	0.16 s

Table 1. Execution times of ADI Kernel on NEC SX-4.

Relaxation on an Unstructured Grid

The next example is a relaxation kernel that uses indirect addressing on an unstructured grid. An integer array specifies for every grid point its four neighbors.

The grid data is block distributed, the numbering of the grid points exploits a high data locality. Table 2 gives the execution times for a fixed problem size and different number of processors. Both versions scale, but the SM version is nearly two times faster than the DM version. The DM version requires complex compile time and run time support to compute the communication schedules implied by the indirect addressing. Even when the overhead amortizes over the iterations by reusing the communication schedule, it remains non-negligible. This is especially true on the NEC SX-4 where indirect access to shared arrays can be vectorized but not the computation of the communication schedule.

	NP = 1	NP = 2	NP = 4	NP = 8
`HPF-DM`	4.327 s	1.972 s	1.021 s	0.545 s
`HPF-SM`	1.940 s	0.983 s	0.500 s	0.264 s

Table 2. Unstructured relaxation on NEC SX-4 (1M grid points, 73 iterations).

Crash Simulation Kernel

For the evaluation of a scientific computation with unstructured reductions, we used a kernel from an industrial crash simulation code [7]. The kernel is based on a time-marching scheme to perform stress-strain calculations on a finite-element mesh consisting of 4-node shell elements. In each time-step elemental forces are calculated for every element of the mesh and added back to the forces stored at nodes by means of unstructured reduction operations. Besides the computation of elemental forces, the unstructured reduction operations to obtain the nodal forces represent the most important contribution to the overall computational costs. Table 3 shows the elapsed times measured on an SGI Origin 2000 (MIPSpro Fortran compiler, version 7.30) for different variants of a crash kernel performing 100 iterations on a mesh consisting of 25600 elements and 25760 nodes. In the table the entry `HPF-DM` refers to the HPF version compiled by ADAPTOR for the DM model using the inspector/executor strategy, `HPF-SM` to the HPF version compiled for the SM execution model using the exclusive-ownership technique, and *OpenMP* to an OpenMP version which synchronizes all updates on the reduction array.

The irregular mesh used in this evaluation exhibits a high locality. As a consequence, both the DM version and the SM version that exploit data locality show very satisfying results. The version using atomic updates for all assignments to the node array scales but exhibits an overhead of about a factor of two.

8 Summary and Conclusion

In this paper we presented an execution model for data parallel programs that takes advantage of threads and the global address space provided on SMPs.

NP	1	2	4	8	16	32
`HPF-DM`	6.39	3.58	1.76	0.99	0.61	0.40
`HPF-SM`	5.10	2.79	1.47	0.74	0.55	0.34
OpenMP	11.39	6.51	3.48	2.06	1.41	1.27

Table 3. Execution times (secs) for crash simulation kernel on the SGI Origin 2000.

Compared to the DM execution model being emulated on SMPs, it avoids the memory overhead implied by replication of data and by non-local copies of data and it yields better performance for a range of applications.

In comparison to the OpenMP programming model, data locality specified in the data parallel HPF program can be exploited directly to achieve better scalability on larger SMPs. Though OpenMP also allows SPMD style parallelism in which the programmer explicitly partitions data and work to achieve comparable performance with optimized message passing versions, this style results in higher programming effort. Furthermore, data parallelism within array statements and `FORALL` statements is used within HPF, but only considered for the next version 2.0 of OpenMP. Efficient OpenMP reduction is currently only defined for scalars, reductions on an array must be carried out "by hand" using other synchronization mechanisms (e.g. a critical section for private reductions or the `atomic` directive for shared reductions).

OpenMP still offers some features that make this programming model more attrative than HPF for a certain range of applications: OpenMP allows to create tasks dynamically and provides features to deal with fluctuating workload, and in OpenMP, tasks might interact with each other in non-trivial ways. But recent developments show that these issues can also be addressed in HPF [2, 4]. Nevertheless, dynamic scheduling strategies show non-negligible overhead, e.g. see [10], and non-trivial task interaction reduces scalability.

In contrast to other approaches where the HPF and OpenMP programming model are combined, we rely on a uniform programming model that is appropriate for all architectures. By a hierarchical mapping of data, this programming model can be also exploited on clusters of SMPs. The first hierarchy defines the mapping of data to the clusters, the second one defines the ownership for the processors within the cluster. The implementation of such an execution model coupling the DM and SM execution model hierarchically is currently under work.

References

1. ADAPTOR. High Performance Fortran Compilation System. WWW documentation, Institute for Algorithms and Scientific Computing (SCAI, GMD), 1999. `http://www.gmd.de/SCAI/lab/adaptor`.
2. G. Antoniu, L. Bougé, R. Namyst, and C. Perez. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. In *The 1999 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 4*, pages 1756–1762, Las Vegas, NV, June 1999.

3. S. Benkner and T. Brandes. Efficient Parallelization of Unstructured Reductions on Shared Memory Parallel Architectures. In *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, Lecture Notes in Computer Science, Cancun, Mexico, May 2000. Springer Verlag. Paper accepted for Irregular'2000 held in conjunction with IPDPS'2000.
4. T. Brandes. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. In Amestoy, P. and Berger, P. and Dayde, M. and Duff, I. and Giraud, L. and Frayssé, V. and Ruiz, D. (Eds.), editor, *Euro-Par'99 Parallel Processing, Toulouse*, pages 833–844. Lecture Notes in Computer Science (1685), Springer-Verlag Berlin Heidelberg, Sept. 1999.
5. T. Brandes and R. Höver-Klier. ADAPTOR User's Guide (Version 7.0). Technical documentation, GMD, Dec. 1999. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/uguide.ps`.
6. B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with Features for Locality Control. Technical report TR99-02, Inst. for Software Technology and Parallel Systems, U. Vienna, Feb. 1999. `www.par.univie.ac.at`.
7. J. Clinckemaillie, B. Elsner, and G. L. et al. Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.
8. High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, Jan. 1997.
9. 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, Jan. 2000.
10. M. Resch, I. Loebich, and B. Sander. A comparison of OpenMP and MPI for the parallel CFD test case. In *Workshop on OpenMP (EWOMP'99) at Lund/Sweden, September 30 - October 1 1999*, Oct. 1999.
11. J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.
12. Silicon Graphics, Inc. MIPSpro (TM) Power Fortran 77 Programmer's Guide. Document 007-2361-007, SGI, 1999.
13. The OpenMP Forum. OpenMP Fortran Application Program Interface. Proposal Ver 1.0, SGI, Oct. 1997. `http://www.openmp.org`.