Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library

Thomas Brandes

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. As task parallelism has been proven to be useful for applications like real-time signal processing, branch and bound problems, and multidisciplinary applications, the new standard HPF 2.0 of the data parallel language High Performance Fortran (HPF) provides approved extensions for task parallelism that allow nested task and data parallelism. Unfortunately, these extensions allow the spawning of tasks but do not allow interaction like synchronization and communication between tasks during their execution and therefore might be too restrictive for certain application classes. E.g., they are not suitable for expressing the complex interactions among asynchronous tasks as required by multidisciplinary applications. They do not support any parallel programming style that is based on non-deterministic communication patterns.
This paper discusses the extension of the task model provided by HPF 2.0 with a task library that allows interaction between tasks during their lifetime, mainly by message passing with an user-friendly HPF binding. The same library with the same interface can also be used for single processors in the local HPF model. The task model of HPF 2.0 and the task library have been implemented in the ADAPTOR HPF compilation system that is available in the public domain. Some experimental results show the easy use of the concepts and the efficiency of the chosen approach.
Keywords: Data Parallelism, Task Parallelism, High Performance Fortran

1 Introduction

High Performance Fortran (HPF) [10] is a data parallel, high level programming language for parallel computing that might be more convenient than explicit message passing and that should allow higher productivity in software development. With HPF, programmers provide directives to specify processor and data layouts, and express data parallelism by array operations or by directives specifying independent computations.

Some users are reluctant to use HPF because many applications do not completely fit into the data parallel programming model. The applications contain

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 833–844, 1999.

data parallelism, but task parallelism is needed to represent the natural computation structure or to enhance performance. Many results verify that a mixed task/data parallel computation can outperform a pure data parallel version (e.g. see [7]) if the granularity of the data is not sufficient; a typical example is the pipelining of data parallel tasks for image and signal processing. Multiblock codes are more naturally programmed as interacting tasks, and applications that interact with external devices. Another important application area is the coupling of different simulation codes (multidisciplinary applications).

Some features supporting task parallelism are available as approved extensions in HPF 2.0[10]. The `TASK_REGION` construct provides the means to create independent coarse-grain tasks, each of which can itself execute a data-parallel or nested task-parallel computation. This kind of task parallelism has been implemented and evaluated within the public domain HPF compilation system ADAPTOR [3]. Currently, ADAPTOR is the only HPF compiler supporting the task model as defined in the new HPF 2.0 standard.

But the HPF task model does not allow interaction between the independent tasks during their execution. The introduction of a task library that allows interaction of HPF data parallel tasks during their lifetime enhances the possibilities of the current model. This paper describes the functionality of such a library and shows that it goes conform with the existing language concepts. The implementation of the task library in an existent HPF compiler should be rather straightforward like it was in the ADAPTOR compilation system.

The rest of this paper is organized as follows. Section 2 describes related work regarding the integration of task and data parallelism. The current HPF task model and its restrictions are outlined in Section 3. Section 4 introduces the HPF task library for interaction of data parallel tasks. Some experimental results in Section 5 show the applicability of the advanced tasking model.

2 Related Work

The need of task parallelism and its benefits have been discussed by many authors and at many places (e.g. in [6]). The promising possibility of integrating task parallelism within the HPF framework has attracted much attention [13, 9, 10, 4]

An integrated task and data parallel model has been implemented in the Fx compiler at Carnegie Mellon [14]. It is mainly based on the specification of subgroups and the assignment of arrays, variables and computations to the subgroups. Communication between the tasks must be visible at the coordination level specified as a `TASK_REGION`. The execution model is not based on message passing, programs can still be executed in the serial model. A variation of this model has become an approved extension for HPF 2.0 [10].

Kohr, Foster et al. [7] developed a coordination library based on MPI to exchange distributed data structures between different HPF programs. The data parallel language has not to be extended at all. Unfortunately, their implementation supports only coupling of different HPF programs, but not of different

HPF tasks created within a task region. But the ideas can be generalized for nested task parallelism.

Zima, Mehrotra et al. [4] propose an interaction mechanism using shared modules with access controlled by a monitor mechanism. A form of remote procedure call (RPC) is used to operate on data in the shared module. The monitor mechanism ensures mutual exclusion of concurrent RPC's to the same module. This concept enhances modularity and is particularly good for multidisciplinary applications. Due to the use of an intermediate space, it appears less well suited for fine-grained or communication-intensive applications.

Orlando and Perego [11] provide run-time support for the coordination of concurrent and communicating HPF tasks. $COLT_{HPF}$ provides suitable mechanisms for starting distinct data-parallel tasks on disjoint group of processors. It also allows the specification of the task interaction on a high level from which they generate automatically corresponding code skeletons. For the implementation of communication between the data parallel tasks they use the pitfalls algorithm [12].

3 Support of Task Parallelism in HPF

The introduction of task parallelism in High Performance Fortran is strongly connected with the `ON` directive that allows the user to control explicitly the distribution of computations among the processors of a parallel machine. It allows dividing processors into subgroups which is essential for task parallelism. The `RESIDENT` directive tells the compiler that only local data is accessed and no communication has to be generated. A code block guided by the `ON` and `RESIDENT` directive is called a *lexical task*. An execution instance of a lexical task is called an *execution task*. Every execution task is associated with a set of *active processors* on which the task is executed.

Though the `ON` and `RESIDENT` directive on their own allow task parallelism, HPF 2.0 provides the `TASK_REGION` construct. A task region surrounds a certain number of lexical tasks where other statements in the `TASK_REGION` can be used to specify data transfers and task interaction between these tasks.

3.1 The `ON` Directive

In the `HOME` clause of the `ON` directive, the user can specify a processor array or a processor subset or an array (template) or a subsection of an array (template). The `ON` directive restricts the active processor set for a computation to those processors named in the home, or to the processors that own at least one element of the specified array or template. As HPF allows the mapping of arrays to processor subsets, the exploitation of task parallelism is more convenient.

It should be noted that the `ON` directive only advises the compiler to use the corresponding processors to perform the `ON` statement or block. Certain statements cannot be executed by an arbitrary processor subset, e.g. allocation, deallocation and redistribution of arrays must include all processors involved in

```
        real, dimension (N) :: A1, A2
!hpf$   processors PROCS(8)
!hpf$   distribute A1 (block) onto PROCS(1:4)
!hpf$   distribute A2 (block) onto PROCS(5:8)
        ...
        do IT = 1, NITER
!hpf$     task_region
!hpf$     on (PROCS(1:4)), resident
           call TASK1 (A1,N)
          if (mod (IT,2) == 0) A1 = A2     ! task interaction
!hpf$     on home (A2), resident
           call TASK2 (A2,N)
!hpf$     end task_region
        end do
```

Fig. 1. Mixed task and data parallelism in High Performance Fortran.

it. But the compiler should inform the user if it overrides the user's advice. Not respecting the `ON` directive can suppress the task parallelism intended by the user.

3.2 The `RESIDENT` Directive

If a statement or a block should be executed by a processor subset, the compiler must make sure that all data is mapped onto the corresponding active processors. This data transfer can involve other processors that are not part of the active processors. Unfortunately, compilers are conservative and can also introduce synchronization or communication where it is not really necessary. The `RESIDENT` directive tells the compiler that only local data is accessed and no communication has to be generated. This guarantees that only the specified processors are involved and the code can be skipped definitively by the other processors. The `RESIDENT` directive is very useful for task parallelism where subroutines are called. It gives the compiler the important information that within the routine only resident data is accessed. This might also allow the compiler to respect the specified `HOME` where it was not possible before.

3.3 The `TASK_REGION` Construct

Though the `ON` and `RESIDENT` directive on their own allow task parallelism, HPF 2.0 provides the `TASK_REGION` construct. A `TASK_REGION` surrounds a certain number of lexical tasks. The `TASK_REGION` construct specifies clearly where task parallelism appears, it provides syntactical restrictions (every `ON` directive must be combined with the `RESIDENT` directive), and the user guarantees no I/O interferences between the different execution tasks. Data dependencies within a `TASK_REGION` can result in a serial execution of tasks. Nevertheless, parallelism might be achieved due to the outer loop around the `TASK_REGION` resulting in a pipelined execution of the tasks (see example in Fig. 1).

The HPF model allows nested task and data parallelism. There is no restriction that execution tasks within one `TASK_REGION` are executed on disjoint processor subgroups. The compiler can ignore the `ON` directive without changing the semantic of the program. Task interaction must be visible at the coordination level which is the code within the task region outside the lexical tasks.

The main disadvantage is the lack of any possibility for task interaction during the execution of the tasks. Communication between tasks in the `TASK_REGION` is deterministic and does not allow any self-scheduling.

4 The HPF Task Library

The HPF task library is intended to allow interaction between different data parallel tasks via message passing. The use of new language constructs has not been considered as it would complicate the whole development.

4.1 Problems and Design Issues

Any task interaction can only be useful if the concurrent execution of the tasks is guaranteed. All execution tasks must be mapped to disjoint processor subsets that is not mandatory for the task concept of HPF. With task interaction, an HPF program might no longer run on a serial machine. Furthermore, task interaction requires the unique identification of tasks, e.g. by a task identifier, that is used to specify the source and destination of message passing.

In the first place, it might have been useful to define the message passing routines between data parallel tasks in analogy to MPI [8], but an own HPF binding of such routines offers a lot of advantages. It allows to pass whole arrays or subsection of arrays to the routines without specifying the data type, the distribution, or the size of the data. The data must not be contiguous and the use of derived data types like in MPI is not really necessary. The routines do not require a communicator as the communicator is implicitly given by the current task nesting level (see also Section 4.4).

The task library is not intended to be realized compiler-independently but as a part of the HPF compiler. Most of the necessary functionality must be available in the HPF runtime system of an HPF compiler. As internal descriptors for arrays and their mappings and internal representations for communication schedules are far away from any standardization, the task library is most efficiently implemented by the compiler vendor itself.

The routines of the `HPF_TASK_LIBRARY` are available in global, local and serial HPF programs by the `USE` statement of Fortran. Two subroutines, `HPF_TASK_INIT` and `HPF_TASK_EXIT` support the initialization and termination of data parallel tasks. They should verify at runtime that the tasks of the current context are really mapped to disjoint processor subgroups. Furthermore, at the end it should be verified that there are no pending messages between the tasks. The two subroutines `HPF_TASK_SIZE` and `HPF_TASK_RANK` return, similar to their MPI counterparts, the size (number of data parallel tasks in the current context) and the rank of the calling task ($1 \leq rank \leq size$).

There are no mechanisms for the creation or termination of task processes at runtime. Tasks can only be defined as subtasks within the current context. Support of task migration can be an option of the runtime system but is not part of the language or library.

4.2 Spawning of HPF Tasks

Tasks can be spawned within a `TASK_REGION`. Their concurrent execution requires that the execution of tasks does not depend on any values computed within the tasks and that there are no dependencies at the coordination level. The user has to assert this property by the keyword `INDEPENDENT`. The following example shows how to invoke data parallel tasks for pipelined data parallelism as described in section 5.2.

```
!hpf$ processors PROCS (20)
      ...
!hpf$ independent task_region
!hpf$ on (PROCS(1:4)), resident            ! will be task 1
         call STAGE1 ()
!hpf$ on (PROCS(5:10)), resident           ! will be task 2
         call STAGE2 ()
!hpf$ on (PROCS(11:16)), resident          ! will be task 3
         call STAGE2 ()
!hpf$ on (PROCS(17:20)), resident          ! will be task 4
         call STAGE3 ()
!hpf$ end task_region
```

All execution tasks within the `INDEPENDET TASK_REGION` get a task identifier starting with 1. Processor subsets in the `ON` directive must not be known at compile time. Therefore the user can implement algorithms on its own to compute the processor subsets for the scheduling of his data parallel tasks.

The HPF Task Library allows also the coupling of separately compiled data parallel programs. Some parallel architectures provide the possibility of loading distinct executables on distinct nodes. Then the data parallel programs will be executed on disjoint processor subsets that are specified on an outer level. The task initialization within the HPF runtime system guarantees that all data parallel tasks know their current task context.

HPF allows the use of local routines via the `EXTRINSIC` mechanism. A local routine allows to write single-processor code that works only on data that is mapped to a given physical processor. The processors executing the local subprogram can be viewed as single processor tasks where task interaction can be used in exactly the same way as for data parallel tasks. The routines in the local model have the same syntax as the corresponding routines for communication between data parallel tasks.

4.3 Communication between Data Parallel Tasks

For the sending of data (scalars, arrays or array sections of any type), the task identifier of the destination task `dest` must be specified. For the receiving of data, the specification of the source task is optional to allow the receiving of data from any task. Every send must have a matching receive.

```
subroutine HPF_SEND (data, dest [,tag] [,order])
subroutine HPF_RECV (data [,source] [,tag])
```

The optional `ORDER` argument for the sending of data must be of type integer, rank one, and of size equal to the rank of `DATA`. Its elements must be a permutation of $(1, 2, ..., n)$, where n is the rank of the data. If the `ORDER` argument is available, the axes of the sending data will be permuted similar to the extended `TRANSPOSE` routine of HPF 2.0.

Due to the HPF/Fortran 90 binding, the routines `HPF_SEND` and `HPF_RECV` can be called with array arguments and the arguments can be named. This makes the use of the routines easier and better readable.

Figure 2 shows the communication pattern between the single processors if `TASK 1` runs on a 2×2 processor subset and `TASK 2` on a processor subset of three processors. The implementation of point-to-point communication between data parallel tasks results in communication between the processors of the two processor subgroups that are involved.

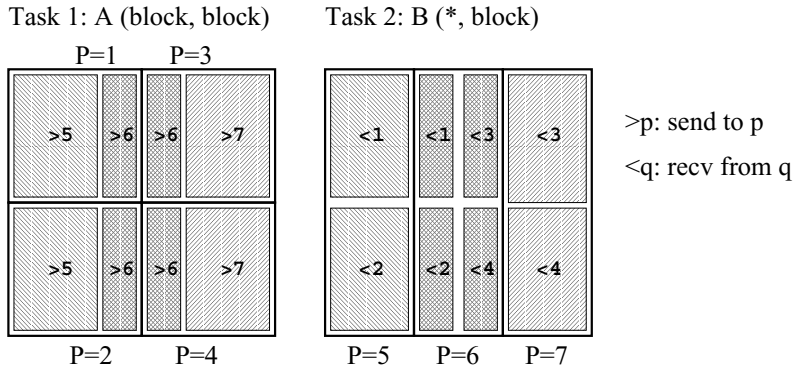

Fig. 2. Example of point-to-point communication between data parallel tasks.

In fact, the implementation of the message passing routines should be straightforward for all HPF compilers as the send and receive operation together correspond to the HPF array assignment between the two arguments. The only difference is that due to the local allocation of the arrays in the processor subset the exchange of the mapping information (*descriptor exchange*) is necessary.

Sending and receiving of distributed data must be assumed to be blocking. When executing shift operations across a chain of tasks or when two tasks are

exchanging data, one needs to order the sends and receives correctly (e.g. even tasks send, then receive, odd tasks receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When using a send-receive routine, the system takes care of these issues.

```
subroutine HPF_SEND_RECV (send_data, dest, recv_data, source
                          [,send_tag] [,recv_tag] [,order])
```

The point-to-point communication between two data parallel tasks causes a certain overhead due to the exchanging of the mapping information. This overhead as well as the computation of the schedule between the single processors of the tasks can be reduced by introduction of internal handles (*request*). The use of these routines might cause serious problems when the distribution or sizes of data has changed.

```
subroutine HPF_SEND_INIT (data, dest, request [,tag] [,order])
subroutine HPF_RECV_INIT (data, source, request [,tag])
subroutine HPF_TASK_COMM (request)
```

Collective communication like in MPI might also be useful for HPF tasks. Especially the broadcast of data and the barrier proved to be useful. It should be observed that the context of these operations is given by the current task context. A barrier synchronizes the tasks of the current context, not the processors within this task.

```
subroutine HPF_BCAST (data [,root])
subroutine HPF_BARRIER ()
```

4.4 Nested Task Parallelism

Task parallelism can be nested. Every independent `TASK_REGION` defines a set of data parallel tasks, in each task new subtasks can be created hierarchically.

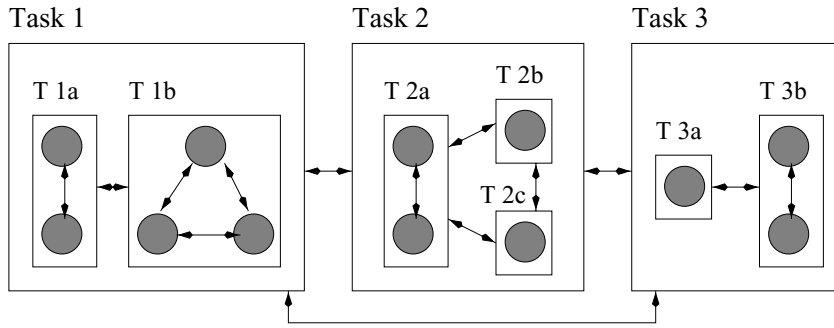

Fig. 3. Task interaction for nested task parallelism.

Task interaction is only possible between the tasks within one context. In the example of Figure 3, task communication is possible between the tasks `TASK 1`,

`TASK 2` and `TASK 3` as long as they are not spawned into subtasks. When `TASK 2` is divided into the three subtasks `TASK 2a`, `TASK 2b` and `TASK 2c`, only communication between these subtasks is possible. Task communication between e.g. `TASK 1a` and `Task 2a` is not possible.

5 Experiments and Results

This section presents some typical patterns for using the HPF task library. More examples regarding task parallelism in HPF and more detailed results can be found in [2].

5.1 Recursive Tree Structured Algorithms

Many problems can be solved recursively by splitting the problem into two partial problems of the half size. Typical examples are the Fast Fourier Transformation (FFT) that plays a key role in many areas of computational science and engineering. Another recursive tree structured algorithm is the Barnes-Hut algorithm for N-body problems [1]. The recursive splitting can be used to split the active processors into two processor subsets via task parallelism. The two subtasks can exchange data via the task library. When the recursion ends up in a single processor, it might be more efficient to call a serial and iterative algorithm.

Table 1 shows the execution times (in seconds) of an one-dimensional FFT ($N = 2^{19}$) that has been implemented in this way. The task parallel version shows reasonable speed-ups when compared with the serial version of the FFT based on the efficient Cooley-Tukey algorithm [5]. It outperforms a data parallel version that uses indirect addressing.

	P=1	P=2	P=4	P=8	P=16	P=32
serial	1.506	n.a.	n.a.	n.a.	n.a.	n.a.
data	5.399	3.363	1.955	1.157	0.671	0.444
task	1.784	1.144	0.762	0.493	0.295	0.184

Table 1. Results for one-dimensional FFT ($N = 2^K, K = 19$) on IBM SP2.

5.2 HPF Task Farming

The HPF task library allows the realization of task farming within a pipeline of data parallel tasks. Let the pipeline have three stages. While the first and the last stage are exactly one task, there are a certain number of worker tasks for the second stage. The load is scheduled to available tasks on this second stage. A worker task sends a ready signal to the first stage when it is free for doing work. It receives the data, works on it and sends it at the end to the final stage NT in the pipeline where `NT` is the number of tasks. Figure 5 shows the principle of message passing between the different tasks.

```
      recursive subroutine DO_IT (A, N, NP)
      use HPF_TASK_LIBRARY
      integer :: N, NP, N2, NP2, SIZE, RANK, IP
      real, dimension(N) :: A, H
!hpf$ processors PROCS(1:NP)
!hpf$ distribute (block) onto PROCS :: A, H
      call HPF_TASK_INIT ()
      call HPF_TASK_RANK (rank=RANK)
      IP = 3 - RANK
      call HPF_SEND_RECV (send_data=A, dest=IP, recv_data=H, source=IP)
      ...
      if (NP .eq. 1) then
         call DO_IT_SERIAL (A, N); return
      end if
      NP2 = NP/2; N2  = N/2
!hpf$ independent task_region
!hpf$ on (PROCS(1:NP2)), resident
         call DO_IT (A(1:N2), N2, Np2)
!hpf$ on (PROCS(NP2+1:NP)), resident
         call DO_IT (A(N2+1:N), N2, Np2)
!hpf$ end task_region
      ...
      call HPF_TASK_EXIT ()
      end subroutine DO_IT
```

Fig. 4. Example of a recursively nested task and data parallel program.

Task farming becomes especially useful when running data parallel programs on heterogeneous architectures.

6 Conclusions

The HPF task model allows the coupling of data parallel tasks in a simple way as long as the interaction between the tasks is completely visible at the coordination level in the `TASK_REGION`. As all information about the mapping of the arrays is available, no exchange of descriptors or distribution information is necessary. The deterministic communication allows the program still to be run on a serial machine. This task model is relatively easy to implement in an HPF environment if the `ON` directive and related clauses are supported as well as the mapping of data to processor subsets.

Moving task interaction within the tasks is rather straightforward. An assignment of data from one task to data of another task becomes a send in the first and a receive in the other task. New allocated data within the tasks (e.g. local variables) can now also be exchanged. Furthermore, only task interaction within the task allows the receiving of values from any other task resulting in non-deterministic behavior. Many well established parallel programming styles like farming can be used in a data parallel framework. Task parallelism can now also be exploited for separately compiled HPF programs. The task library combines

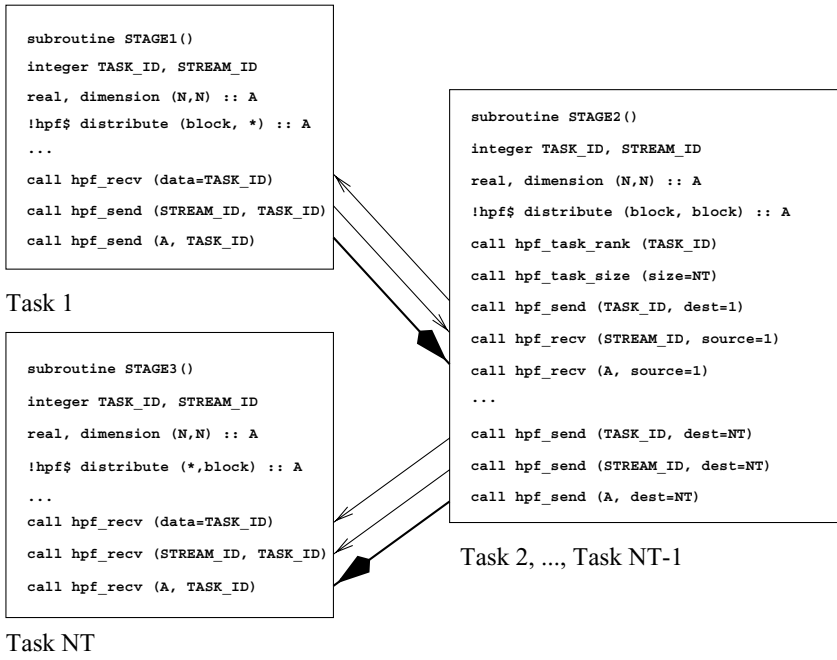

Fig. 5. Task farming for data parallel tasks.

the advantages of the HPF task model based on processor subsets and a sequential semantic with the advantages of the great flexibility when using message passing. The concept of the task library goes conform with the other extrinsic models of HPF and therefore allows the combination with other programming models, e.g. MPI.

The use of communication via send and receive in a language like HPF seems to destroy the high level intention. But data parallel computations still do not need any message passing, it is restricted only to the interaction of running data parallel tasks where it is indeed quite natural. The high level nature of HPF is taken into account by providing a high level HPF binding of the communication routines (no arguments for size, data type, distribution, context, etc.).

The ADAPTOR HPF compilation systems provides task parallelism as specified in HPF and the task library as described in this paper. This library or a more standardized library with a similar and improved functionality could be provided by any HPF compiler vendor as it can be implemented rather easily by using the HPF runtime system that has to support redistributions in any case. Experiments so far have shown that the use of task parallelism in HPF is very user-friendly, no more limited, and, most important, efficient enough to be a very attractive alternative. Its support became also necessary as ADAPTOR is intended to support nested process and thread parallelism for hierarchical systems in future.

Acknowledgements

Most thanks are due to Salvatore Orlando (Universita di Venezia) and Raffaele Perego (CNUCE, Italy) for many valuable discussions and a lot of technical hints. I am indebted to Mike Delves (NA Software, Liverpool) for the great idea to make the task library also available in the local HPF model.

References

[1] J. Barnes and P. Hut. A hierarchical $O(NlogN)$ force calculation algorithm. *Nature 4*, 324:446–449, 1986.

[2] T. Brandes. Implementation and Evaluation of Nested Task and Data Parallelism for High Performance Fortran within the ADAPTOR Compilation System. Working paper (unpublished), GMD, 1998. available via anonymous ftp as `ftp.gmd.de:/GMD/adaptor/docs/tasking.ps`.

[3] T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K. Decker and R. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, Apr. 1994.

[4] B. Chapman, M. Haines, P. Mehrotra, H. Zima, and J. Van Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(4):345–361, 1997.

[5] J. W. Cooley and J. W. Tukey. An Algorithm for the machine calculation of complex Fourier series. *Mathematical Computing*, 19:297–301, 1965.

[6] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, E. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Mar. 1994.

[7] I. Foster, D. Kohr, K. R., and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF via the Message Passing Interface. In P. Pittsburgh, editor, *Supercomputing '96*, Nov. 1996.

[8] W. Groop, E. Lusk, and A. Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, 1994.

[9] T. Gross, D. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(2):16–26, 1994.

[10] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, Jan. 1997.

[11] S. Orlando and R. Perego. $COLT_{HPF}$, a Coordination Layer for HPF Tasks. Technical Report Series on Computer Science CS-98-4, Universita ca' Foscari di Venezia, Mar. 1998. Paper submitted to Concurrency: Practice and Experience.

[12] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computations (FRONTIERS'95)*, pages 342–394, Feb. 1995.

[13] S. Ramaswamy, S. Spatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transaction on Parallel and Distributed Systems*, 8(11):1098–1116, Nov. 1997.

[14] J. Subhlok and B. Yang. A New Model for Integrated Nested Task and Data Parallel Programming. In *PPOPP 97*, 1997.