Implementation and Evaluation of Nested Task and Data Parallelism for High Performance Fortran within the ADAPTOR Compilation System

(Working Paper, unpublished)

Thomas Brandes
Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
Tel.: +49-2241-142492, Fax: +49-2241-142181
e-mail: brandes@gmd.de

Abstract

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 task 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. Widely accepted parallel programming paradigms like farming are not expressible.

This paper describes the support of task parallelism in High Performance Fortran and its realization within the ADAPTOR HPF compilation system. This system supports task parallelism as specified in the HPF 2.0 standard, but also allows interaction between tasks during their lifetime by providing a task library. This task library offers routines for exchanging distributed data between different data parallel tasks. Some example programs show the easy use of the concepts and the efficiency of this approach.

Keywords: Data Parallelism, Task Parallelism, High Performance Fortran

1 Introduction

High Performance Fortran (HPF) [13, 14] 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.

Users are very reluctant to use HPF because many applications do not completely fit into the data parallel programming model. The applications contain data parallelism, but task parallelism is needed to represent the natural computation structure or to enhance performance. Typical examples are the pipelining of data parallel tasks for image and signal processing to improve performance, multiblock codes are more naturally programmed as interacting tasks, and applications that interact with external devices. Many results verify that a mixed task/data parallel computation can outperform a pure data parallel version (e.g. see [10]) if the granularity of the data is not sufficient.

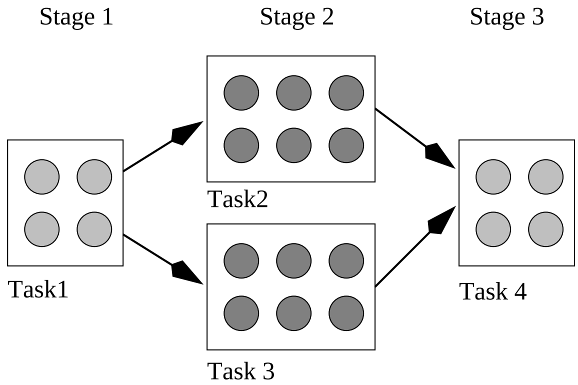

Figure 1: Pipelined execution of data parallel tasks.

Another important application area is the coupling of different simulation codes. Examples are simulations in the areas of fluid-structure interaction, magneto-hydro-dynamics, acoustics, vibrations of structures due to electromagnetic forces, and sono-chemistry. Such applications combine a number of programs representing different disciplines into an integrated system of interacting processes. Coupling communication libraries like COCOLIB [2] have been developed to couple different simulation codes written in MPI. In the same way, it should be possible to couple different HPF simulation codes.

Different approaches for supporting task parallelism are possible. Some approaches try to identify task parallel structures automatically, but this complicates compiler development and performance tuning. Language extensions can provide the necessary execution model for task parallelism but have the need of standardization. Library support on its own is very convenient as a starting point, but more effective if embedded in the language.

Some features supporting task parallelism are available as approved extensions in HPF 2.0[14]. 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 [5]. 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 presents related work. Section 3 gives a short introduction to the ADAPTOR HPF compilation system. Section 4 shows the use of the `ON` directive in HPF that is needed as a basis for task parallelism. The execution of subprograms by only a processor subset involves some problems that are discussed in Section 5. Section 6 summarizes the support of task parallelism in HPF 2.0 and discusses its limitations regarding task interaction. Section 7 describes the HPF task library as our approach for introducing interactions between disjoint tasks. This HPF task library can also be used for communication between the processors when a local routine is executed as Section 8 will point out. The example in Section 11 describes in detail how to realize a load-driven pipelined task farm with our task library.

2 Related Work

The need of task parallelism and its benefits have been discussed by many authors and at many places (e.g. in [9]). The promising possibility of integrating task parallelism within the HPF framework has attracted much attention [17, 12, 14, 7, 6]

An integrated task and data parallelism model has been implemented in the Fx compiler at Carnegie Mellon [19]. 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 [14].

Kohr, Foster et al. [10] 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 are very similar when generalized for nested task parallelism.

Banerjee et al. [17] have an approach where the programmer has to specify an input/output list for all the HPF tasks involved. Tasks can be either `PURE` HPF-like subroutines or simple statements. Since the user does not specify neither the allocation of tasks on specific processor subsets nor explicit communications between tasks, the compiler has to extract corresponding information and to guarantee proper allocations and communications. The type of task interaction that can be specified is still deterministic.

Zima, Mehrotra et al. [7, 6] 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 [15] 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 [16].

The coupling communication library COCOLIB [2] has been developed to couple different simulation codes written in MPI. The ideas are similar to the communication between two or more different data parallel programs but here the data parallel programs are also implemented in MPI. Also the functionality of this library is very high by providing features like inter- and extrapolation between the data structures of the different applications.

3 The ADAPTOR HPF Compilation System

ADAPTOR (Automatic Data Parallelism Translator) is a public domain HPF compilation system developed at GMD for compiling data parallel HPF programs to equivalent message passing programs [5].

3.1 Overview of the System

The latest release of ADAPTOR supports nearly the full HPF 2.0 base language and many of the approved extensions [3]. Support of the `ON` directive and related clauses exists already for a longer time.

By means of a source-to-source transformation, ADAPTOR translates the data parallel program to an equivalent SPMD program (single program, multiple data) that runs on all available nodes. Beside the translation system, a runtime system called DALIB (distributed array library) has been developed that will be linked with the generated SPMD program (see Figure 2).

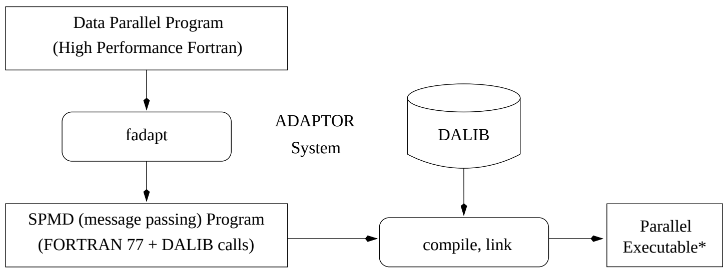

Figure 2: Overview of ADAPTOR.

3.2 Availability

The source files of ADAPTOR, documentation files in PostScript and a number of example programs are available via `ftp` [4]. The latest version 6.1 of ADAPTOR supports task parallelism as presented in this paper.

4 The ON Directive of 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.

4.1 Syntax and Semantic

There are two flavors of the ON directive: a single-statement form and a multi-statement form.

```
!hpf$ processors PROCS(4)
      real, dimension (N) :: A
!hpf$ distribute A(block) onto PROCS(3:4)
      ...
!hpf$ on (PROCS(1:2))
        call SUB1()
!hpf$ on home (A) begin
        call SUB2()
        call SUB3()
!hpf$ end on
```

In the HOME clause, 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. It should be noted that the ON directive only advises the compiler to use the corresponding processors to perform the ON statement or block. 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.

4.2 HOME Specifications

All HOME specifications as proposed in the HPF 2.0 standard can be used within ADAPTOR.

- If processors are named in the home, ADAPTOR takes exactly the specified processor subset as active processors. Values in the subscript must not be known at compile time.

```
      subroutine (NP, IP1, IPS)
      integer :: NP, IP1, IPS
!hpf$ processors P(1:NP)
       ...
!hpf$ on (P(IP1:IPS))
         <block>
```

- If the home is specified by an array or template, scalar subscripts are specifying exactly one processor while slices as subscripts usually take all processors of this dimension in the active processor subset.

```
      real, dimension (N,N) :: A
!hpf$ processors P(NP1,NP2)
!hpf$ distribute (cyclic(5),cyclic(10)) onto P :: A
      ...
!hpf$ on home (A(11:14,12))
         <block>              ! on P(1:NP1,IP), IP owner of 12 in 2nd dim
```

- In contrary to the HPF 2.0 standard, ADAPTOR also allows vector-subscripts for processor subsets. This gives more flexibility in mapping data to certain processors and for the selection of processors executing a task.

```
!hpf$ processors P (6)
      integer, dimension (4) :: IND = (/ 1, 3, 4, 6 /)
      real, dimension (N)    :: A1, A2
!hpf$ distribute A1 (block) onto P
!hpf$ distribute A2 (block) onto P(IND)
      ...
!hpf$ on (P(IND))
         call TASK (A2, N)
```

4.3 Data Transfers with the ON 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.

- All dummy arguments must be mapped onto the active processors. By this way, dummy arrays are local within the subprogram that is only executed by a processor subset.

  ```
        integer, parameter :: N = 100
  !hpf$ processors PROCS(4)
        real, dimension (N) :: A
  !hpf$ distribute A(block) onto PROCS
        ...
  !hpf$ on (PROCS(1:2))
        call TASK (A,N)
  ```

  While the dummy argument N is available on the active processors, the array A must be redistributed. The compiler might create implicitly the following code:

  ```
  !hpf$ redistribute A(block) onto PROCS(1:2)
  !hpf$ on (PROCS(1:2))
        call TASK (A,N)
  !hpf$ redistribute A(block) onto PROCS
  ```

- All local objects of the called subprogram must be mapped onto the active processors.

The compiler will create temporary data and inserting copy-in and copy-out communication for non-local data. Any replicated data changed on a processor subset has to be made consistent afterwards. All processors that were not active must get copies of the new values.

As HPF allows the mapping of arrays to processor subsets, the exploitation of task parallelism is more convenient. Unfortunately, the mapping of scalars to processor subsets is only possible via an alignment.

```
        real S
!hpf$   processors PROCS (4)
!hpf$   template T (2)
!hpf$   distribute T(block) onto PROCS (1:2)
!hpf$   align S with T(*)
```

A special directive is very convenient to tell the compiler that certain replicated variables should have only an incarnation on a processor subset. By this way, the compiler will not have to guarantee consistency between all processors.

```
       real S, A(10)
!hpf$  processors PROCS (4)
!hpf$  onto PROCS (1:2) :: S, A
```

4.4 Restrictions for the `ON` Directive

Certain statements cannot be executed by a given processor subset, e.g.:

- Allocation and deallocation of an array is only possible if all data is only mapped to the active processors.

  ```
  !hpf$  processors PROCS (4)
         real, dimension (:), allocatable :: A
  !hpf$  distribute A(block) onto PROCS (1:2)
         ...
  !hpf$  on (PROCS(1:2))
         allocate (A)           ! is allowed
  !hpf$  on (PROCS(1:2))
         deallocate (A)         ! not possible
  ```

- Any redistribution must include all processors involved in it.

4.5 Coupling of the `ON` and `RESIDENT` Directives

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.

```
!hpf$  on (PROCS(1:2)), resident
       call TASK1 (A1,N)
!hpf$  on (PROCS(3:4)), resident
       call TASK2 (A2,N)
```

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.

4.6 Parallelism with the ON Directive

The `ON` directive on its own inserts already task parallelism in a natural way. If the data is available on the specified processor set and no communication is required, the statement can be skipped by all the other processors. Code blocks mapped to disjoint processor sets will be executed in parallel.

```
       real, dimension (N) :: A1, A2
!hpf$  processors PROCS(4)
!hpf$  distribute A1 (block) onto PROCS(1:2)
```

```
!hpf$   distribute A2 (block) onto PROCS(3:4)
        ...
!hpf$   on (PROCS(1:2))
        call TASK1 (A1,N)
!hpf$   on (PROCS(3:4))
        call TASK2 (A2,N)
```

The code blocks might not be executed simultaneously if communication is involved. This will be the case if one of the code blocks uses data that has no incarnation on the specified processor subset or if it defines data that has also incarnations on other processors.

5 Compiling Subprograms for Processor Subsets

With the introduction of task parallelism, every subprogram in HPF must be compiled in such a way that it can be executed on any processor subset. The compiler cannot make any assumptions on which processors the code is executed at runtime. It might also be possible that the code will be executed several times on completely different processor subsets.

```
      subroutine TASK (A, B, N)
      real, dimension (N) :: A, B
!hpf$ distribute (block) :: A, B
      real, dimension (N) :: C
!hpf$ distribute (block) :: C
      ...
      end subroutine TASK
```

This implies the following strategies for the HPF compiler:

- Local arrays will be allocated only on the active processors, and not on all physical processors. So local arrays are always resident on the active processor subset.
- Dummy arrays are assumed to be available on the active processors. The calling routine is responsible for remapping arguments to the active processor subset so that the data is resident on the active processors (see Section 4.3).
- Theoretically, global arrays can be allocated on any processor subset. In the following example, the global array A might be allocated onto all processors, while the local array B is only allocated on a processor subset.

```
      module DATA
      integer, parameter :: N = 10000
      real, dimension (N,N) :: A
!hpf$ distribute A (block, block)
      end module DATA

      subroutine SUB (..)
      use DATA
      real, dimension (N,N) :: B
!hpf$ distribute B (block, block)
      B = A
      end subroutine SUB
```

In the current version of ADAPTOR, the compiler assumes that all global data will be resident on the active processors.

- The global use of the PROCESSORS directive causes troubles when the corresponding processor array includes processors that are not in the active processor subset.

```
      module DATA
!hpf$ processors P(3,3)   ! universal processor arrangement
      end module DATA

      subroutine SUB (..)
      use DATA
      real, dimension (N,N) :: B
!hpf$ distribute B (block, block) onto P    ! be careful
      ...
      end subroutine SUB
```

The HPF standard provides the subset directive to define processor arrangements that are not universal. This feature is not supported within ADAPTOR yet.

The access to global arrays causes serious problems for HPF compilers generating SPMD code based on message passing. Data can be on processors that are not in the active processor subset and these processors cannot send data needed by other processors or receive data defined by other processors. The correct implementation would require one-sided communication.

ADAPTOR always assumes that access to global data is resident. Unfortunately, this implies a certain responsibility for the user.

6 Task Parallelism in HPF

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.

6.1 The TASK_REGION Directive

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.

```
        real, dimension (N,N) :: A1,A2

!hpf$   processors PROCS(4)
!hpf$   distribute A1 (*,block) onto PROCS(1:2)
!hpf$   distribute A2 (*,block) onto PROCS(3:4)

!hpf$   task_region
!hpf$     on home (A1), resident
            call TASK1 (A1,N)
!hpf$     on home (A2), resident
            call TASK2 (A2,N)
!hpf$   end task_region
```

The task region has some advantages:

- clear specification where task parallelism appears,
- it provides syntactical restrictions (every `ON` directive must be combined with the `RESIDENT` directive),
- the user guarantees no I/O interferences between the different execution tasks.

6.2 Pipelined Execution of Data Parallel Tasks

In the following example, the data dependencies due to the array assignment result in a serial execution of the two tasks. Nevertheless, parallelism is achieved due to the outer loop around the task region (see also example in Section 10).

```
        real, dimension (N,N) :: A1,A2

!hpf$   processors PROCS(4)
!hpf$   distribute A1 (*,block) onto PROCS(1:2)
!hpf$   distribute A2 (*,block) onto PROCS(3:4)

        ! define a task region, otherwise home will be ignored

        do ITER = 1, NITERS
!hpf$   task_region
!hpf$     on home (A1), resident
            call TASK1 (A1,N)
          A1 = A2
!hpf$     on home (A2), resident
            call TASK2 (A2,N)
!hpf$   end task_region
        end do
```

6.3 Discussions

The 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.

7 The HPF Task Library

This section introduces the HPF task library that is intended to allow interaction between different data parallel tasks via message passing.

7.1 Problems and Design Issues

Though HPF provides already features for task-parallel computations, task interaction involves some new problems that have to be addressed:

- Any task interaction can only be useful if the parallel execution of the tasks is guaranteed. For this reason, all execution tasks must be executed on disjoint processor subsets. This is not mandatory for the task concept of HPF. With task interaction, an HPF program might no longer run on a serial machine.
- 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.

Providing task interaction via a Fortran 90 library has the following advantages:

- The embedding of task interaction in the language would complicate the whole development.
- The Fortran 90 binding allows to pass whole arrays or subsection of arrays to the routines.
- Compared to the MPI standard [11] that provides routines for the exchanging of data between serial processes, the exchanging of data between data parallel processes must be more synchronous.

In the first place, it might have been useful to define the message passing routines between data parallel tasks in analogy to MPI. But there is not even a standardized Fortran 90 binding for the MPI routines. The advantage of optional arguments as well as of array and array section arguments is enormous.

The follwoing differences exist when comparing the concepts of MPI and the HPF task library:

- The use of derived datatypes is not necessary due to the HPF/Fortran 90 binding that also allows non-contiguous data in their arguments.
- The routines do not require a communicator as the communicator is implicitly given by the current task nesting level.
- Sending of distributed data must be synchronous to exchange distribution information.

Furthermore, 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 for communication schedules are far away from any standardization, the task library is most efficiently implemented by the compiler vendor.

7.2 Task Numbering in a `TASK_REGION`

The implementation of a task numbering requires that the execution of tasks does not depend on any values computed within the tasks. The user has to assert this property by the keyword `INDEPENDENT`.

It also guarantees that there are no dependencies at the coordination level and all tasks will be really invoked simultaneously. All execution tasks within a parallel task region get a task identifier starting with 1. All tasks together build a context that is known for every task.

The following example shows how to invoke data parallel tasks for pipelined data parallelism as presented in Figure 1.

```
!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
```

The implementation of the independent TASK_REGION must observe that every processor has to define all tasks before it branches into the code of the task to which it belongs. The definition of tasks in a loop might be possible.

```
!hpf$ processors PROCS (20)
      ...
!hpf$ independent task_region
      do I = 0, 3
!hpf$ on(PROCS (I*5+1:I*5+5)), resident
        call TASK ()
      end do
!hpf$ end task_region
```

7.3 Task Initialization and Identification

Within any data parallel task, the HPF_TASK_LIBRARY can be used to have access to the routines for interaction between different tasks.

```
     subroutine TASK ()
     use HPF_TASK_LIBRARY
     ...
     end subroutine TASK
```

The following subroutines support the initialization and termination of data parallel tasks. They might already called implicitly when the data parallel tasks are invoked.

```
subroutine HPF_TASK_INIT ()

subroutine HPF_TASK_EXIT ()
```

The call of these routines is not mandatory but might assert additional runtime checks. They could verify at runtime that the tasks of the current context are really mapped to disjoint processor subgroups. Furthermore, at the end it could be verified that there are no pending messages between the tasks.

The following subroutines return the size (number of data parallel tasks in the current context) and the rank of the calling task ($1 \leq rank \leq size$).

```
subroutine HPF_TASK_SIZE (size)
integer, intent(out) :: size

subroutine HPF_TASK_RANK (rank)
integer, intent(out) :: rank
```

7.4 Coupling of Data Parallel Programs

The HPF Task Library allows also the coupling of separately compiled data parallel programs.

```
program TASK1                      program TASK2
use HPF_TASK_LIBRARY               use HPF_TASK_LIBRARY
...                                ...
call HPF_TASK_INIT ()              call HPF_TASK_INIT ()
call HPF_TASK_SIZE (SIZE)          call HPF_TASK_SIZE (SIZE)
call HPF_TASK_RANK (RANK)          call HPF_TASK_RANK (RANK)
!  RANK = 1, SIZE = 2              !  RANK = 2, SIZE = 2
...                                ...
call HPF_TASK_EXIT ()              call HPF_TASK_EXIT ()
end program TASK1                  end program TASK2
```

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 guarantees that all data parallel tasks know the current task context.

The initialization in the HPF runtime system has to verify that different executables are loaded. Otherwise all data parallel tasks would deliver `SIZE=1` and `RANK=1`. The usual way to guarantee the parallel execution of HPF tasks is to create a task file (e.g. `TASKS`) that contains the corresponding task names and task sizes.

The ADAPTOR HPF compiler uses the environment variable `TASK_FILE` that must be set with the name of the task file. This environment variable will be checked at runtime and the corresponding creation of HPF tasks initiated. The parallel machine itself must provide a mechanism to start the different executables on the nodes of the parallel machine. The numbering and execution of the executables must correspond to the entries in the task file.

```
setenv TASK_FILE TASKS

# content of task file TASKS
STAGE1  2
STAGE2  3
STAGE2  3
STAGE3  2
```

7.5 Dynamic Tasking

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.

```
!hpf$ processors PROCS (number_of_processors())
      integer P, P1, P2, P3, P4
      ...
      P = number_of_processors()
      call schedule (P, P1, P2, P3, P4)
!hpf$ independent task_region
!hpf$ on (PROCS(1:P1))                    ! will be task 1
         call STAGE1 ()
!hpf$ on (PROCS(P1+1:P1+P2))              ! will be task 2
         call STAGE2 ()
!hpf$ on (PROCS(P1+P2+1:P1+P2+P3)         ! will be task 3
         call STAGE2 ()
!hpf$ on (PROCS(P-P4+1:P)                 ! will be task 4
         call STAGE3 ()
!hpf$ end task_region
```

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. During the task execution the processor subset is fixed.

7.6 Nested Task Parallelism

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

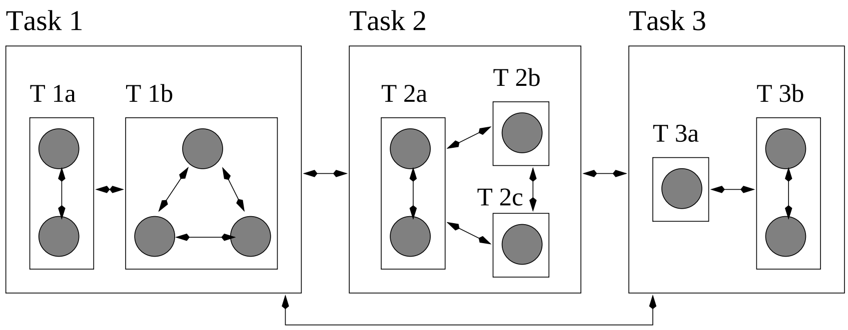

Figure 3: Task interaction for nested task parallelism.

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.

7.7 Point-to-Point Communication between Data Parallel Tasks

For the sending of data (scalars, arrays or array sections), the task identifier of the target task must be specified.

```
subroutine HPF_SEND (data, dest, tag, order)
<type>,  dimension <>, intent(in)           :: data
integer, intent (in)                        :: dest
integer, intent (in), optional              :: tag
integer, dimension(:), intent(in), optional  :: order
```

The ORDER argument 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 the rank of the data. If the order argument is available, the axes of the data will be permuted.

```
call HPF_SEND (data=arr, dest=pid, order = (/1, 3, 2/))
call HPF_SEND (data=TRANSPOSE (arr, order = (/1, 3, 2/)), dest=pid)
```

The receiving of data is similar, but without the ORDER argument. Every send must have a matching receive.

```
subroutine HPF_RECV (data, source, tag)
<type>,  intent(out)          :: data
integer, intent (in), optional :: source
integer, intent (in), optional :: tag

integer, parameter :: HPF_ANY_SOURCE = -1
integer, parameter :: HPF_ANY_TRAG   = -1
```

Due to the 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.

```
      subroutine TASK1 (N)                     subroutine TASK2 (N)
      use HPF_LIBRARY                          use HPF_LIBRARY
      integer, intent (in) :: N                integer, intent (in) :: N
      real, dimension (N,N) :: A               real, dimension (N,N) :: B
!hpf$ distribute A (block,block)          !hpf$ distribute B (*,block)
      ...                                      ...
      call HPF_SEND (data=A, dest=2)           call HPF_RECV (data = B, source=1)
      ...                                      ...
```

Figure 4 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. In fact, the implementation of these routines should be straightforward for all HPF compilers. For the above example, the communication patterns are exactly the same as for the following HPF program:

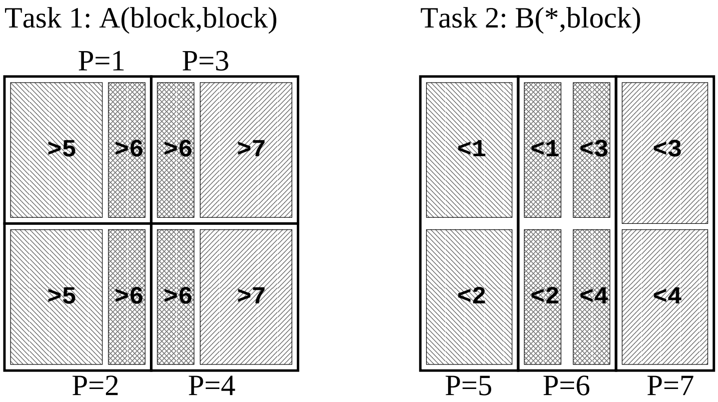

Figure 4: Example of point-to-point communication between data parallel tasks.

```
      real, dimension (N,N) :: A, B
!hpf$ processors Q(1:7)
!hpf$ processors P(2,2)
!hpf$ distribute A (block,block) onto P
!hpf$ distribute B (*,block) onto Q(5:7)
      ...                                       ...
      B = A
```

The only difference is that due to the local allocation of the arrays in the subgroups the exchange of the mapping information (*descriptor exchange*) is necessary.

The following restrictions are given:

- If the tasks are not really executed concurrently, the code might result in a deadlock.
- Every send must have a corresponding receive as otherwise the communication will conflict with compiler generated communication outside the task region.
- The sending and receiving of arrays is always blocking.
- The source argument is optional. By this way, it is possible to receive a message from an arbitrary task. But only scalar data can be received from any processor.

7.8 Persistent Communication Requests

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.

```
subroutine HPF_SEND_INIT (data, dest, request, tag, order)
integer, intent (in)  :: dest
<type>,  intent (in)  :: data
integer, intent (out) :: request
```

```
integer, intent (in), optional               :: tag
integer, dimension(:), intent(in), optional  :: order

subroutine HPF_RECV_INIT (data, source, request, tag)
integer, intent (in)  :: source
<type>,  intent (out) :: data
integer, intent (out) :: request
integer, intent (in), optional :: tag

subroutine HPF_TASK_COMM (request)
integer, intent (in)  :: request
```

The use of these routines might cause serious problems when the distribution or sizes of data has changed.

7.9 Other Routines

- Collective communication like in MPI might also be useful for HPF tasks. Especially the broadcast of data and the barrier proved to be very 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)
  <type>,  intent (inout)        :: data
  integer, intent (in), optional :: root

  subroutine HPF_BARRIER ()
  ```

- 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)
  integer, intent (in)  :: dest, source
  <type>,  intent(in)   :: send_data
  <type>,  intent(out)  :: recv_data
  integer, intent (in), optional :: send_tag
  integer, intent (in), optional :: recv_tag
  integer, dimension(:), intent(in), optional  :: order
  ```

7.10 Comparison with Task Coordination in a TASK_REGION

There is a strong and absolutely intended relation between task interaction within the tasks via the HPF task library and task coordination within a TASK_REGION. The following example realizes a typical pipelined computation using the TASK_REGION construct of HPF. The subroutines Task1 and Task2 have only interaction with each other on the coordination level through their arguments.

```
      do IT = 1, NITER              subroutine TASK1(A)
!hpf$ task_region                   < work on A>
!hpf$ on home (A)                   end subroutine TASK1
         call TASK1 (A)
         B = A                      subroutine TASK2(B)
!hpf$ on home (B)                   < work on B>
         call TASK2 (B)             end subroutine TASK2
!hpf$ end task_region
      end do
```

The same computation can be realized with independent tasks that interact with each other during their lifetime.

```
!hpf$ independent task_region
!hpf$ on home (A)
         call TASK1 (A)
!hpf$ on home (B)
         call TASK2 (B)
!hpf$ end task_region
```

```
      subroutine TASK1 (A)             subroutine TASK2 (B)
      use HPF_TASK_LIBRARY             use HPF_TASK_LIBRARY
      do IT = 1, NITER                 do IT = 1, NITER
         <work on A>                      recv (data=B, source=1)
         send (data=A, dest=2)            <work on B>
      end do                           end do
```

Comparing these two concepts, the following differences should be observed:

- Task interaction within the TASK_REGION allows the program to be run also on a serial machine.
- Within a TASK_REGION, all information about the mapping of the arguments is available. No exchange of descriptors or distribution information is necessary.
- Within a TASK_REGION no new allocated data of the tasks (e.g. local variables) can be exchanged.
- Communication between tasks in the TASK_REGION is deterministic. Only task interaction within the task allows the receiving of values from any other task.
- HPF does not really allow to map scalar variables to processor subsets. Assignments to scalar variables within a TASK_REGION can destroy parallel execution as processors will be blocked even if they do not really need the value.

```
!hpf$ task_region
!hpf$ on home (A)
         call TASK1 (A)
         B = A                      ! might also block processors of TASK3
!hpf$ on home (B)
         call TASK2 (B)
         C = B                      ! might also block processors of TASK1
!hpf$ on home (C)
         call TASK3 (C)
!hpf$ end task_region
```

Generally, task interaction should be visible in the task region for efficiency and portability reasons. In certain cases, this is not possible but then the HPF Library can be used. Both kinds of interactions can be combined without any problems.

8 Local Routines

HPF provides the `EXTRINSIC` mechanism to couple the data parallel programming model with other models. This section outlines that the local model goes conform with the task model and the HPF task library can be used in a similar way.

8.1 Tasks in `LOCAL` Subprograms

HPF allows the use of local routines. A local routine allows to write single-processor code that works only on data that is mapped to a given physical processor. In this sense, a local routine contains only local computations. Within the local subroutine, the active processor subset is restricted to a single processor.

```
interface
  extrinsic (HPF_LOCAL) subroutine SUB (A)
  real, dimension (:) :: A
  end subroutine SUB
end interface

real, dimension (N) :: A
...
call SUB (A)
```

The processors executing the local subprogram can be viewed as single processor tasks where task interaction is useful in the same way as for data parallel tasks.

```
!hpf$ independent task_region
!hpf$ on (P(1))
         call sub (A(lb(1):ub(1))
!hpf$ on (P(2))
         call sub (A(lb(2):ub(2))
      ...
!hpf$ on (P(k))
         call sub (A(lb(k):ub(k))
!hpf$ end task_region
```

Tasking for local subroutines is supported in the same way.

- The subroutine `TASK_SIZE` returns the number of processors executing the same local subroutine. This number corresponds to the value of the global `HPF_LIBRARY` function `ACTIVE_NUM_PROCS` as if it has been called before the call of the local routine (within the local routine the number of active processors is 1). The subroutine `TASK_RANK` returns the corresponding id $1 \leq id \leq P$ where P is the value return by `TASK_SIZE`.
- Any task interaction provided for data parallel tasks is available in the same way for the tasks (processors) executing a local subprogram. So the routines in the local model have the same syntax as the corresponding routines for communication between data paralllel tasks. As every task consists of exactly one processor, no descriptor exchange is necessary.

8.2 Local Routines in HPF Tasks

When a local routine is called within a code section where not all processors are active, some attention must be paid to the following topics:

- The HPF local routine library identifies each processor by an integer in the range 0 to $n-1$ where n is the value returned by the global `HPF_LIBRARY` function `NUMBER_OF_PROCESSORS`. More convenient would be the value returned by the global `HPF_LIBRARY` function `ACTIVE_NUM_PROCS`.
- As no longer all processors might be involved, message passing is restricted. If e.g. MPI is used, a special routine should provide the last HPF context for the global HPF task.
- The other routines of the `HPF_LOCAL_LIBRARY` should work fine. The term `GLOBAL` refers to the last global HPF task context.

9 Example of Nested Task Parallelism: FFT

The Fast Fourier Transformation (FFT) plays a key role in many areas of computational science and engineering. The Fourier transformations input is an N-vector of complex numbers representing some discretized function and it computes the coefficients of the constituent function yields that provide a great deal of information about the function. Cooley and Tukey presented in 1965 [8] an efficient algorithm that has been the basis for nearly all optimizations on all kind of computer architectures. The structure of the algorithm is similar to other tree structured algorithms loke Barnes-Hut for N-body problems [1].

9.1 The FFT Algorithm

Let $a(x)$ be a given polynom:

$$a(x) = \sum_{i=0}^{N-1} a_i x^i$$

The Fourier transformation computes the values of this polynom for the complex roots of unity $\omega = e^{2\pi i/N}$, where $i = \sqrt{-1}$. We are looking for the values of the roots:

$$A_j := a(\omega^j) \quad 0 \leq j < m \quad m = N = 2^k$$

Viewed merely as a linear system, $O(N^2)$ time is needed to compute the result coefficients A_i. The well-known Fast Fourier Transform technique requires only $O(NlogN)$ time, using the following identity:

$$\omega^{j+n} = -\omega^j \quad N = 2n$$

Now the polynom can be split by and odd/even transformation into two polynoms $b(y)$ and $c(y)$:

$$a(x) = b(y) + xc(y) \quad y = x^2 \quad b(y) = \sum_{i=0}^{n-1} a_{2i} y^i \quad c(y) = \sum_{i=0}^{n-1} a_{2i+1} y^i$$

The problem is splitted into two problems that have half the size. The solution of b and c can be combined to a solution of the original problem:

$$\begin{array}{lll} A_j & = a(w^j) & = b(w^{2j}) + w^j c(w^{2j}) \\ A_{j+n} & = a(w^{j+n}) & = b(w^{2j}) - w^j c(w^{2j}) \end{array}$$

This recursion continued up to problems of size 1 result in an algorithm that has the complexity $O(NlogN)$.

9.2 The Recursive FFT Program

Taking the idea of the recursive splitting results directly in the following data and task parallel HPF program:

```
      recursive subroutine FFT (A, W, N, N2)
      integer, intent (in) :: N, N2
      complex, dimension (0:N2-1), intent(in) :: W
      complex, dimension (0:N-1), intent(inout) :: A
!hpf$ inherit :: A
!hpf$ range (block()) :: A

      complex, dimension (0:N2-1) :: B, C    ! temporary arrays
!hpf$ align B(I) with A(I)
!hpf$ align C(I) with A(I+N2)

      if (N == 1) return     !  we are done now

      B (0:N2-1) = A(0:N-1:2)
      C (0:N2-1) = A(1:N-1:2)

      if (N2 > 1) then
!hpf$ on home (B)
         call FFT (B, W(0:N2-1:2), N2, N2/2)
!hpf$ on home (C)
         call FFT (C, W(0:N2-1:2), N2, N2/2)
      end if

      A(0:N2-1) = B(0:N2-1) + C(0:N2-1) * W(0:N2-1)
      A(N2:N-1) = B(0:N2-1) - C(0:N2-1) * W(0:N2-1)

      end subroutine FFT
```

- Active processors branch into two different subroutine calls of the recursive FFT. The compiler should not need the `RESIDENT` clause to identify the locality of the arguments.
- Most efficient only when the number of processors is a power of two. Otherwise one processor might participate in both calls.
- High reality on compiler that two disjoint processor subgroups are built up (not even done by ADAPTOR yet).
- ADAPTOR does not support alignment with a value not known at compile time.

- Inefficient due to many subroutine calls, also for the smaller arrays on one processor.
- Passing of the array section of `W` with the roots can result in copy-in and copy-out.

9.3 Presorting

The first part of the FFT is the reordering of the elements of the input array `A`.

```
recursive subroutine PRESORT (A, N, K)
integer, intent (in) :: N, K
complex, dimension (0:N-1), intent(inout) :: A
complex, dimension (:), allocatable :: H
if (N > 2) then
   N2 = N/2
   allocate (H(0:N-1))
   H = A
   A(0:N2-1) = A(0:N-1:2)
   A(N2:N-1) = H(1:N-1:2)
   deallocate (H)
   call PRESORT (A(0:N2-1), N2, K-1)
   call PRESORT (A(N2:N-1), N2, K-1)
end if
end subroutine PRESORT
```

Figure 5 shows the corresponding communication pattern for this reordering.

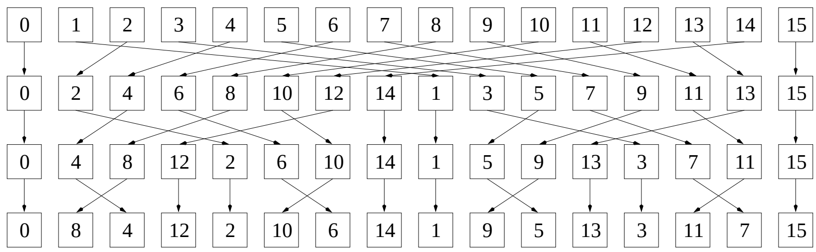

Figure 5: FFT presorting by bit reversing.

The presorting can be implemented more efficiently if the final permutation (bit reversion) is computed before the final data movement is done.

```
      complex, dimension (0:N-1) :: A, H
      integer, dimension (0:N-1) :: INDEX
      ...
!     part 1 : compute the bit reversion table
      INDEX(0)=0
      STRIDE=1
      OFFSET=N2
      do L = 1, K
         do I = 0, STRIDE - 1
            INDEX(I+STRIDE)=INDEX(I)+OFFSET
```

```
         end do
         OFFSET=OFFSET/2
         STRIDE=STRIDE*2
      end do
      ...
!     part 2: do the rerodering
      H = A(INDEX)
      A = H
```

The following algorithm computes the index array completely independently for all values.

```
       N2 = N/2
!hpf$ independent, new (HI, KN, IN, HL)
       do I = 0, N-1
          HI = I; KN = N2; IN = 0
          do L = 0, K-1
             HL = iand (HI,1)             ! HI mod 2, is bit L of I
             if (HL > 0) IN = ior (IN,KN) ! IN = IN + HL * KN
             HI = ishft (HI,-1)           ! HI / 2
             KN = ishft (KN,-1)           ! KN / 2
          end do
          INDEX (I) = IN
       end do
```

	K = 16	K = 17	K = 18	K = 19
serial+index	0.020	0.023	0.055	0.161
serial	0.008	0.013	0.035	0.125
data, serial+index	0.041	0.084	0.223	0.589
data, serial	0.008	0.018	0.074	0.273
data, P=1	0.091	0.189	0.452	1.701
data, reuse, P=1	0.015	0.039	0.116	0.358
task, P=1	0.011	0.023	0.088	0.288
data, P=2	0.046	0.095	0.211	0.480
data, reuse, P=2	0.009	0.022	0.049	0.122
task, P=2	0.014	0.042	0.072	0.212
data, P=4	0.026	0.049	0.126	0.252
data, reuse, P=4	0.007	0.012	0.026	0.065
task, P=4	0.015	0.030	0.070	0.172

Table 1: Presorting on SGI Origin

9.4 Transformation

```
      recursive subroutine TRANSFORM (A, W, N, N2, K)

      integer, intent (in) :: N, N2
      complex, dimension (0:N2-1), intent(in) :: W
      complex, dimension (0:N-1), intent(inout) :: A
```

```
!hpf$ distribute A (block) onto *

      complex, dimension (:) :: H   ! temporary array
!hpf$ align H with A

      if (N == 1) return     !  we are done now

      if (N2 > 1) then
         call TRANSFORM (A(0:N2-1), W(0:N2-1:2), N2, N2/2, K-1)
         call TRANSFORM (A(N2:N-1), W(0:N2-1:2), N2, N2/2, K-1)
      end if

      allocate (H(0:N-1))

      H(0:N2-1) = A(N2:N-1)
      H(N2:N-1) = A(0:N2-1)

      A(0:N2-1) = A(0:N2-1) + H(0:N2-1) * W(0:N2-1)
      A(N2:N-1) = H(N2:N-1) - A(N2:N-1) * W(0:N2-1)

      deallocate (H)

      end subroutine TRANSFORM
```

	K = 16	K = 17	K = 18	K = 19
serial	0.078	0.173	0.392	0.860
data, P=1	0.181	0.411	1.097	3.112
task, P=1	0.078	0.172	0.380	0.864
data, P=2	0.127	0.305	0.633	1.766
task, P=2	0.049	0.106	0.433	0.954
data, P=4	0.081	0.179	0.371	1.075
task, P=4	0.028	0.064	0.141	0.332

Table 2: Transformation on SGI Origin

9.5 Results

	K = 14	K = 15	K = 16	K = 17	K = 18	K = 19
Cooley, xlf	0.040	0.080	0.170	0.370	0.800	1.790
Cooley, hpf	0.036	0.076	0.161	0.350	0.744	1.694
Cooley1, hpf	0.043	0.113	0.244	0.525	1.140	2.521
data, xlf	0.160	0.400	0.840	1.780	3.790	8.140
data, hpf	0.125	0.376	0.695	1.489	3.241	6.969
rec, xlf	0.100	0.220	0.470	1.010	2.240	4.950
rec, hpf	error	error	error	error	error	error

Table 3: Serial results for one-dimensional FFT on IBM SP2

	K = 14	K = 15	K = 16	K = 17	K = 18	K = 19
serial	0.125	0.376	0.695	1.489	3.241	6.969
P=1	0.413	0.930	1.981	4.235	10.828	22.551
P=2	0.268	0.595	1.241	2.634	5.491	13.368
P=4	0.160	0.322	0.679	1.391	2.901	6.289
P=8	0.108	0.197	0.375	0.771	1.541	3.525
P=16	0.116	0.157	0.245	0.444	0.585	1.686

Table 4: Data parallel results for one-dimensional FFT on IBM SP2.

	K = 14	K = 15	K = 16	K = 17	K = 18	K = 19
serial	0.043	0.113	0.244	0.525	1.140	2.521
P=1	0.044	0.114	0.245	0.524	1.135	2.526

Table 5: Task parallel results for one-dimensional FFT on IBM SP2.

	K = 16	K = 17	K = 18	K = 19	K = 20	K = 21
bserial	0.151	0.326	0.698	1.506	3.270	6.950
P=1	0.550	1.177	2.502	5.399	12.962	42.537
P=2	0.356	0.731	1.534	3.363	7.751	18.007
P=4	0.230	0.464	0.938	1.955	4.496	10.519
P=8	0.153	0.295	0.569	1.157	2.546	6.021
P=16	0.101	0.195	0.362	0.671	1.409	3.213
P=32	0.092	0.140	0.252	0.444	0.836	1.742
P=1	0.163	0.348	0.765	1.712	3.666	7.972
P=2	0.145	0.293	0.601	1.260	2.680	5.692
P=4	0.124	0.235	0.443	0.866	1.794	3.737
P=8	0.096	0.181	0.325	0.643	1.212	2.447
P=16	0.075	0.141	0.267	0.464	0.878	1.739
P=32	0.071	0.121	0.222	0.407	0.705	1.299
P=1	0.173	0.369	0.784	1.784	4.079	8.000
P=2	0.133	0.267	0.554	1.144	2.483	5.210
P=4	0.097	0.193	0.384	0.762	1.545	3.278
P=8	0.064	0.122	0.242	0.493	0.965	1.975
P=16	0.045	0.082	0.143	0.295	0.564	1.153
P=32	0.035	0.053	0.129	0.184	0.337	0.642

Table 6: Results for one-dimensional FFT on IBM SP2

10 Example of HPF Task Pipelining: 2D FFT

The 2D FFT takes an $n_1 \times n_2$ input array A, performs n_2 independent n_1-point FFTs on the columns of A, followed by n_1 independent n_2-point 1D FFTs on the rows of A. The row-wise FFTs are replaced by a transpose followed by a set of column-wise FFTS and another transpose. Figure 6 shows the task graph for the 2D FFT. This example has been adapted from the Carnegie Mellon task parallel program suite [18].

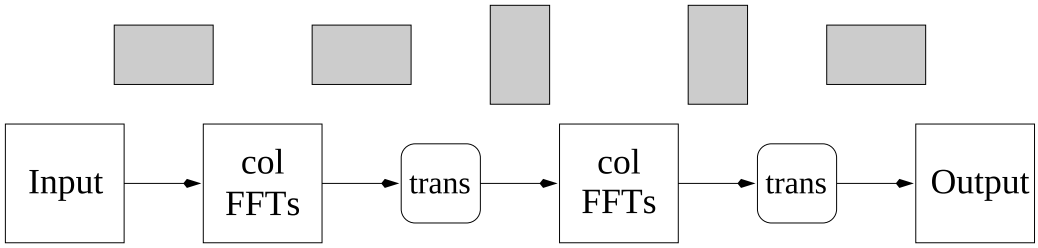

Figure 6: 2D FFT task graph for one input array.

10.1 The Data Parallel Version

```
      real, dimension (2,N, N) :: A, B   ! input/intermediate images
!hpf$ distribute (*,*,block) :: A, B

      do K=1,ITERS
         call DGEN(A,N)                   ! get the input image
         call CFFTS(A,BRT,W,N,LOGN,N/2)  ! FFT's on each column
         call TPOSE(A,B,N)                ! transpose the image
         call CFFTS(B,BRT,W,N,LOGN,N/2)  ! FFT's on each column again
         call TPOSE(B,A,N)                ! final transpose
         call CHKMAT(A,N)                 ! print/check the results
      enddo
```

10.2 The HPF Task Version

The HPF task version needs four arrays, one for every task. The mapping of these arrays to processor subsets specifies the mapping of the tasks to processor subsets

```
      real, dimension (2,N, N) :: A, B, A1, A2

!hpf$ processors P(1:NP)

!hpf$ distribute (*,*,block) onto P(P1L:P1U):: A1
!hpf$ distribute (*,*,block) onto P(P2L:P2U):: A
!hpf$ distribute (*,*,block) onto P(P3L:P3U):: B
!hpf$ distribute (*,*,block) onto P(P4L:P4U):: A2

!hpf$ task_region
```

```
!hpf$ on home (A), resident
        call DGEN(A1,N)                  ! get the input image

        A = A1

!hpf$ on home (A1), resident
        call CFFTS(A,BRT,W,N,LOGN,N/2) ! FFT's on each column

        call TPOSE(A,B,N)              ! transpose the image

!hpf$ on home (B), resident
        call CFFTS(B,BRT,W,N,LOGN,N/2) ! FFT's on each column again

        call TPOSE(B,A2,N)             ! final transpose

!hpf$ on home (A2), resident
        call CHKMAT(A2,N)              ! print/check the results

!hpf$ end task_region
```

The transpose routine will be entered by all processors, but processors that do not participate will leave the routine immediately.

```
      subroutine TPOSE (A, B, N)
      integer, intent(in) :: N
      real, dimension (2,N,N) :: A, B
!hpf$ distribute (*,*,BLOCK) onto * :: A, B

      integer :: I, J, K

      forall (K=1:N,J=1:N,I=1:2) B(I,J,K) = A(I,K,J)

      end subroutine TPOSE
```

Attention: The `ONTO *` clause is very important to tell the compiler that the subroutine must handle actual arguments that are mapped onto any processor subset. This guarantees that there will be never any redistribution of the arguments.

10.3 Results

	serial	NP = 1	NP = 2	NP = 4	NP = 8
only data	253	296	173	101	56
data+task	286	308	164	95	52

Table 7: Results for two-dimensional FFT on IBM SP2

11 Example of Scheduled HPF Tasks

The example in this section shows how to implement a pipeline of data parallel tasks. The pipeline has 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.

11.1 Task Creation

```
!hpf$ independent task_region
!hpf$ on (PROCS(1)), resident                   ! will be task 1
        call STAGE1 ()
!hpf$ on (PROCS(2:PA+1)), resident              ! will be task 2
        call STAGE2 ()
!hpf$ on (PROCS(PA+2:P-1)), resident            ! will be task 3
        call STAGE2 ()
!hpf$ on (PROCS(P)), resident                   ! will be task 4
        call STAGE3 (ERRORS(1))
!hpf$ end task_region
```

Figure 7 shows the principle of message passing between the different tasks.

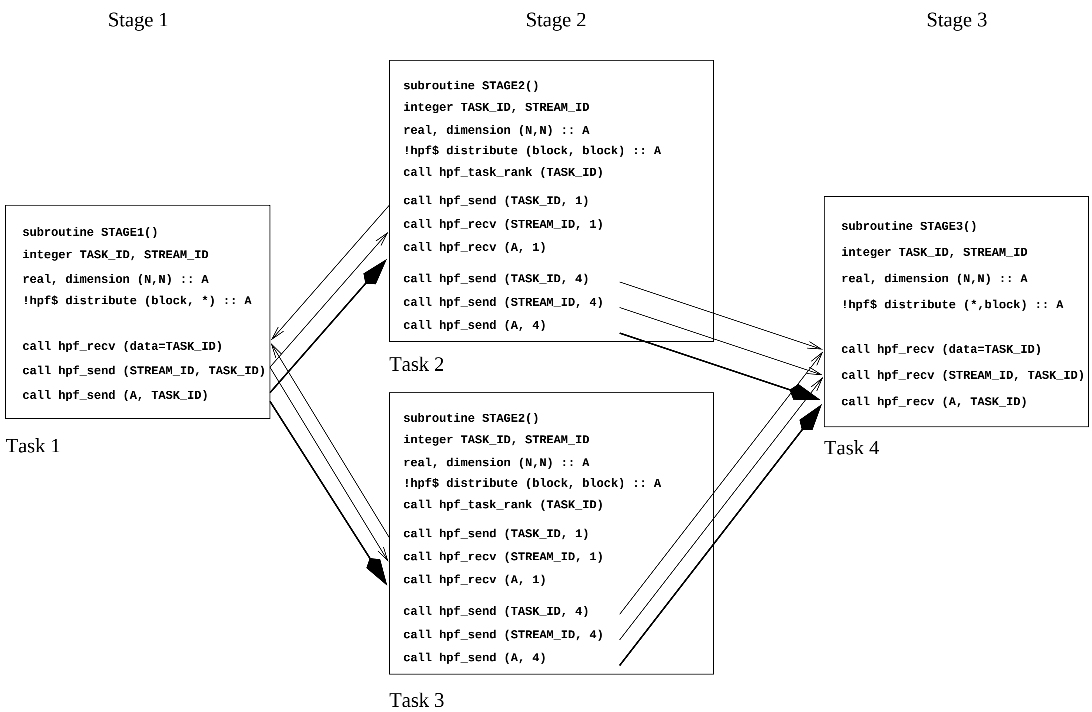

Figure 7: Message passing for data parallel tasks.

11.2 Stage 1

The subroutine `STAGE1` implements the data parallel task for the first stage in the pipeline. It initializes the data and sends it to the next available worker task on stage 2.

```
subroutine STAGE1 ()

use HPF_TASK_LIBRARY

implicit none

integer, parameter :: N = 100
integer, parameter :: ITERS = 9

integer :: WORKERS          ! number of tasks for STAGE2
integer :: STREAM_ID
integer :: TASK_ID
integer :: I, J

integer, dimension (N,N) :: A
!hpf$ distribute (block,*) :: A

call HPF_TASK_INIT ()
call HPF_TASK_SIZE (WORKERS)

WORKERS = WORKERS - 2      ! first and last task are not workers

STREAM_ID = 1

do while (STREAM_ID <= ITERS)

   forall (I=1:N,J=1:N) A(J,I) = STREAM_ID     ! init array A

   call HPF_RECV (TASK_ID)  ! wait for ready task of STAGE 2

   call HPF_SEND (data=STREAM_ID, dest=TASK_ID)  ! send stream id
   call HPF_SEND (data=A, dest=TASK_ID)          ! send data

   STREAM_ID = STREAM_ID + 1

end do

do I = 1, WORKERS                ! send end of stream to all working tasks
   call HPF_RECV (data=TASK_ID)  ! receive any task id
   call HPF_SEND (0, TASK_ID)    ! send task end of stream signal
end do

call HPF_TASK_EXIT ()

end subroutine STAGE1
```

11.3 Stage 2: Working Tasks

The subroutine STAGE2 implements the data parallel task for the second stage in the pipeline. There might be more than one incarnation of this task. Every incarnation will be a worker task. This worker task sends a ready signal to the first stage1 when it is free for doing work. It receives the data, works on it and sends it at the end to the final stage 3 in the pipeline.

```
subroutine STAGE2 ()

use HPF_TASK_LIBRARY

implicit none
```

```
integer, parameter :: N = 100
integer :: TASK_ID
integer :: SIZE
integer :: STREAM_ID

integer, dimension (N,N) :: A
!hpf$ distribute (block,*) :: A

call HPF_TASK_SIZE (size=SIZE)        ! get number of tasks
call HPF_TASK_RANK (rank=TASK_ID)     ! get my TASK_ID

call HPF_SEND (TASK_ID, dest=1)       ! tell task 1 that I am ready
call HPF_RECV (STREAM_ID, source=1)   ! wait for a stream id

do while (STREAM_ID <> 0)             ! stop for stream id 0

   call HPF_RECV (A, source = 1)      ! receive the data

   A = A + 1                          ! data parallel work for A

   call HPF_SEND (TASK_ID, SIZE)      ! tell final task that I am ready
   call HPF_SEND (STREAM_ID, SIZE)    ! send the stream id
   call HPF_SEND (data=A, SIZE)       ! send the data

   call HPF_SEND (TASK_ID, dest=1)        ! tell task 1 that I am ready
   call HPF_RECV (STREAM_ID, source=1)    ! wait for the next stream id

end do while

call HPF_SEND (TASK_ID, SIZE)    ! tell final task that I am finished
call HPF_SEND (0, SIZE)          ! send the end of stream signal

end subroutine STAGE2
```

11.4 Stage 3: Final Task

The subroutine STAGE3 implements the final stage in the pipeline. It collects the data arrays from the worker tasks until all worker tasks have finished.

```
subroutine STAGE3 ()

use HPF_TASK_LIBRARY

implicit none

integer, parameter :: N = 100
integer, parameter :: WORKERS = 2

integer :: STOPS          ! counts end of workers
integer :: TASK_ID        ! identification for communication task
integer :: STREAM_ID      ! identification of data stream

integer, dimension (N, N) :: A
!hpf$ distribute (*, block) :: A

STOPS = 0
ERRORS = 0
```

```
do while (STOPS <> WORKERS)    ! there are still unfinished workers

   call HPF_RECV (data=TASK_ID)                    ! wait for ready task
   call HPF_RECV (source=TASK_ID, data=STREAM_ID)  ! receive the stream id

   if (STREAM_ID == 0) then
       STOPS = STOPS +1
     else
       call HPF_RECV (source=TASK_ID, data=A)
       call output (A)                             ! final work on A
   end if

end do while

end subroutine STAGE3
```

12 Implementation of Task Parallelism within ADAPTOR

12.1 Support of Processor Subsets

The support of processor subsets was one of the first issues when starting with the implementation of task parallelism.

12.2 Support of the `ON` Directive

The `ON` directive was already available for a long time, but with certain restrictions. With the support of processor subsets, now also processors can be specified as the home of computations.

12.3 Compiler Support

Beside the syntactical checks for a `TASK_REGION`, the compiler has now also to generate code that defines task identifiers for all lexical tasks.

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

The definition of the task context must be executed by all processors. The tasks itself are executed on the corresponding processor subsets that become the active processors.

```
     call DALIB_start_tasking ()
     call DALIB_subtop_create (PROCS_1_TOPID,PROCS_DSP,1,1,4,1)
     call DALIB_set_task_procs (PROCS_1_TOPID)
     call DALIB_subtop_create (PROCS_2_TOPID,PROCS_DSP,1,5,10,1)
```

```
call DALIB_set_task_procs (PROCS_2_TOPID)
call DALIB_subtop_create (PROCS_3_TOPID,PROCS_DSP,1,11,16,1)
call DALIB_set_task_procs (PROCS_3_TOPID)
call DALIB_subtop_create (PROCS_4_TOPID,PROCS_DSP,1,17,20,1)
call DALIB_set_task_procs (PROCS_4_TOPID)
if (DALIB_is_in_procs(PROCS_1_TOPID)) then
   call DALIB_push_procs_context (PROCS_1_TOPID)
   call STAGE1 ()
   call DALIB_pop_procs_context ()
end if
if (DALIB_is_in_procs(PROCS_2_TOPID)) then
   call DALIB_push_procs_context (PROCS_2_TOPID)
   call STAGE2 ()
   call DALIB_pop_procs_context ()
end if
if (DALIB_is_in_procs(PROCS_3_TOPID)) then
   call DALIB_push_procs_context (PROCS_3_TOPID)
   call STAGE2 ()
   call DALIB_pop_procs_context ()
end if
if (DALIB_is_in_procs(PROCS_4_TOPID)) then
   call DALIB_push_procs_context (PROCS_4_TOPID)
   call STAGE3 ()
   call DALIB_pop_procs_context ()
end if
call DALIB_stop_tasking ()
```

Furthermore, the compiler will check the correct typing of the routines of the HPF_TASK_LIBRARY. It translates them to corresponding calls of the DALIB runtime system.

```
call HPF_RECV (TASK_ID)  ! receive any taskid from other task
call HPF_SEND (data=STREAM_ID, dest=TASK_ID)
call HPF_SEND (data=A, dest=TASK_ID)
```

In case of a scalar argument, the compiler generates an additional parameter for the number of bytes. In case of an array argument, it passes the array descriptor.

```
call DALIB_HPF_RECV_SCALAR (TASK_ID,4,DALIB_0)
call DALIB_HPF_SEND_SCALAR (STREAM_ID,4,TASK_ID)
call DALIB_HPF_SEND_ARRAY (A_DSP,TASK_ID)
```

12.4 Runtime Support

- Management of a task context, allowing nested task parallelism.
- Implementation of the routines of the HPF_TASK_LIBRARY.

The highest functionality is given by the possibility of sending and receiving distributed arrays or sections of distributed arrays between data parallel tasks. Most of this functionality was already available in the DALIB runtime system. Only the exchange of descriptors for arrays, sections and mappings had to be added as a new functionality.

In case of task interaction with global arrays, the descriptors have not to be exchanged between the tasks as every task has already the necessary information.

12.5 Outstanding ADAPTOR Problems

- In the following example the actual argument will be redistributed. Though there is no communication involved, it requires copy-in and copy-out data transfer.

```
      real, dimension (N) :: A
!hpf$ processors P(1:NP)
!hpf$ distribute A(block) onto  P
      ...
!hpf$ on (P(1:NP2), resident
         call SUB (A(0:N2-1))
```

  Runtime support is required as the compiler is not able to recognize that the array section as an actual argument has still a block distribution.

- No mapping of scalar variables to processor subsets.
- Sending and receiving of replicated arrays causes serious problems as compiler assumes own copies when calling the corresponding assignment problem
- TAG argument for send and receive not available
- ORDER argument not supported yet, problem is that receiving process has also to know the permutation
- No `DO` loop within an independent task region to create a variable number of tasks.

13 Conclusions and Future Work

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`. Coupling of data parallel subprograms is possible via the argument lists, there is no interaction during the execution of the subprogram. 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. By this way, task parallelism becomes available for separately compiled HPF programs. Many well established parallel programming styles like farming can be used in a data parallel framework if the receiving of arbitrary tasks is supported.

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, datatype, context, etc.).

We considered other solutions than message passing, but due to the similarity with MPI it seems to be very convenient to follow this library approach. It also combines 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 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.

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] E. Brakkee, K. Wolf, D.-P. Ho, and A. Schüller. The COupled COmmunications LIBrary. In University of Westminster, editor, *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Computing, London, UK*, pages 155–162, London, UK, January 1997. IEEE Computer Society Press.

[3] T. Brandes. ADAPTOR Programmer's Guide (Version 6.0). Technical documentation, GMD, June 1998. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/pguide.ps`.

[4] T. Brandes and R. Höver-Klier. ADAPTOR Installation Guide (Version 6.0). Technical documentation, GMD, June 1998. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/iguide.ps`.

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

[6] 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.

[7] B. M. Chapman, P. Mehrotra, J. Van Rosendale, and H. P. Zima. A software architecture of multidisciplinary applications: Integrating task and data parallelism. In *Proceedings of CONPAR94-VAPP VI Third International Conference on Vector and Parallel Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 664–676. Springer Verlag, September 1994.

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

[9] 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, March 1994.

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

[11] 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.

[12] 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.

[13] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.1, Department of Computer Science, Rice University, November 1994.

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

[15] 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, March 1998. Paper submitted to Concurrency: Practice and Experience.

[16] 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, February 1995.

[17] 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, November 1997.

[18] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, 21:150–159, April 1994.

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