$COLT_{\texttt{HPF}}$, a Run–Time Support for the High–Level Coordination of HPF Tasks

Salvatore Orlando*, Raffaele Perego†

February 22, 1999

Abstract

This paper describes $COLT_{\texttt{HPF}}$, a run–time support specifically designed for the coordination of concurrent and communicating HPF tasks. $COLT_{\texttt{HPF}}$ is implemented on top of MPI and requires only small changes to the run–time support of the HPF compiler used. Although the $COLT_{\texttt{HPF}}$ API can be used directly by programmers to write applications as a flat collection of interacting data-parallel tasks, we believe that it can be used more productively through a compiler of a simple *high-level coordination language* which facilitates programmers in structuring a set of data-parallel HPF tasks according to common forms of *task-parallelism*. The paper outlines design and implementation issues, and discusses the main differences from other approaches to exploiting task parallelism in the HPF framework. We show how $COLT_{\texttt{HPF}}$ can be used to implement common forms of parallelism, e.g. pipeline and processor farms, and we present experimental results regarding both synthetic micro-benchmarks and sample applications. The experiments were conducted on an SGI/Cray T3E using Adaptor, a public domain HPF compiler.

Keywords: *Task parallelism, Data Parallelism, Coordination Languages, HPF.*

*S. Orlando is with the "Dipartimento di Informatica" of the "Università Ca' Foscari di Venezia", via Torino 155, Venezia Mestre, 30173 Italy. E-mail: orlando@unive.it

†R. Perego is with the "Istituto CNUCE" of the "Consiglio Nazionale delle Ricerche (CNR)", via S. Maria 36, Pisa, 56126 Italy. E-mail: r.perego@cnuce.cnr.it

1 Introduction

Although HPF-1 allows programmers to express data-parallel computations in a portable, high-level way [18], it is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data-parallel paradigm [10]. For these applications, rather than having a single data-parallel program, it is more profitable to subdivide the whole computation into several data-parallel pieces, where the various pieces run concurrently and cooperate, thus exploiting *task parallelism*. The advantage of exploiting both forms of parallelism is twofold. On the one hand, the exploitation of parallelism at different levels may significantly increase the scalability of applications which may exploit only a limited amount of data parallelism [14]. On the other hand, the capability of integrating task and data parallelism into a single framework allows the number of addressable applications to be enlarged. Task parallelism is in fact often needed to reflect the natural structure of an application. For example, applications of computer vision, image and signal processing, can be naturally structured as *pipelines* of data-parallel tasks where stages which deal with external devices may be run on a few processors to better match the available I/O bandwidth, while the remaining processors can be more efficiently used to run computing-intensive parts of the application.

Depending on the applications, HPF tasks can be organized according to patterns which are structured to varying degrees. For example, applications may be modeled by a fixed but unstructured task dependency graph, where edges correspond to data-flow dependencies. However, it is more common for parallel applications to process streams of input data, so that a more regular *pipeline* task structure can be used. Several optimizations can be exploited in this case to increase the efficiency of pipeline structures [23]. For example, the degree of parallelism of some data-parallel stages can be reduced or increased in order to balance the pipeline and optimize its bandwidth. In some cases, to enhance performance, it may be better to replicate a "slow" pipeline stage rather than using several processors for its data-parallel implementation. Replication entails using a *processor farm* [2] structure, where incoming jobs are dispatched on one of the replicas of the stage by adopting either a simple round-robin or a dynamic self-scheduling policy. Note that when the per-job execution times of a replicated stage are not uniform, a dynamic self-scheduling policy may be mandatory in order to balance the workload assigned to the various replicas [3].

In this paper we present $COLT_{\texttt{HPF}}$ (COordination Layer for Tasks expressed in HPF), a portable coordination/communication layer for HPF tasks implemented on top of the MPI communication layer. $COLT_{\texttt{HPF}}$ provides suitable mechanisms for starting distinct HPF data-parallel tasks on disjoint groups of processors, along with optimized primitives for inter-task communication where data to be exchanged may be distributed among the processors according to user-specified HPF directives. The $COLT_{\texttt{HPF}}$ interface can be used directly by programmers to write their applications as a flat collection of interacting data-parallel tasks, but we believe that $COLT_{\texttt{HPF}}$ may be more effectively used through a compiler of a simple high-level coordination language which facilitates programmers in structuring a set of data-parallel HPF tasks according to common forms of *task-parallelism*, such as pipelines and processor farms [15, 19]. We present *templates* which implement these forms of task parallelism, and we discuss the exploitation of these templates by means of a structured, high-level, template-based coordination language that facilitates programmers in organizing HPF tasks according to these forms of parallelism. We claim that the use of such a coordination language simplifies program development and restructuring, while effective automatic optimizations (mapping, choice of the degree of parallelism for each task, program transformations) can be easily devised because the structure of the parallel program is statically known. Unfortunately, this approach requires a new compiler in addition to HPF, though the templates proposed can also be exploited to design libraries of *skeletons* [6, 9, 3]. However, the compiler is very simple, though its complexity may increase depending on the level of optimization supported.

The paper is organized as follows. Section 2 introduces $COLT_{\texttt{HPF}}$ design and functionalities, and describes its implementation on top of the MPI communication layer and the Adaptor compilation system [4]. Section 3 shows how a coordination layer like $COLT_{\texttt{HPF}}$ can be used to design templates exploited by a compiler of a simple, high-level, template–based coordination language. In Section 4 the synthetic micro-benchmarks and the applications used to validate our approach are presented, and the results of the experiments conducted on an SGI/Cray T3E are discussed in depth. Section 5 surveys works dealing with the introduction of task parallelism in the HPF framework, and finally, Section 6 draws some conclusions.

2 Implementation of $COLT_{\texttt{HPF}}$

In this section we describe in detail the implementation of the $COLT_{\texttt{HPF}}$ layer on top of the Adaptor HPF compilation system and the MPI communication layer. First of all we specify the mechanisms which allow multiple HPF tasks to be executed concurrently onto disjoint groups of MPI processors. Although it refers to the Adaptor compilation system, our approach is general, thus any consideration made is valid irrespective of the particular HPF compiler exploited.

We then discuss the techniques used to establish communication channels between data-parallel tasks, and the primitives devised to exchange simple and structured data through these channels. Communicating distributed data between data-parallel tasks entails making several point-to-point communications and, when the data and processor layouts of the sender and receiver tasks differ, it also requires the data exchanged to be redistributed. We solved the data redistribution problem adopting the *pitfalls* algorithm by Ramaswamy and Banerjee's [21]. Since run-time redistribution algorithms are quite expensive and the same communications are usually repeated many times, $COLT_{\texttt{HPF}}$ allows *communication schedules* to be computed only once, and to be reused when possible. Finally, we discuss some $COLT_{\texttt{HPF}}$ primitives which are useful for signaling simple events between tasks, where the reception of messages may be carried out in a non-deterministic way. These primitives are needed to implement many forms of task parallelism which will be discussed in Section 3.

$COLT_{\texttt{HPF}}$ primitives are implemented as `HPF_LOCAL EXTRINSIC` subroutines[16]. This means that when a $COLT_{\texttt{HPF}}$ primitive is invoked by an HPF task, all the processors executing the task switch from the single thread of the control model supplied by HPF to an SPMD style of execution. Depending on the language definition, `HPF_LOCAL` subroutines have to be written in a restricted HPF language where, for example, data stored on remote processors cannot be accessed transparently, but each processor can only access its own section of any distributed array.

2.1 Group definition and task loading

HPF compilers that use MPI as an underlying communication layer, exploit one or more MPI *communicators*, a powerful MPI abstraction that allows communications to occur only within a given *context*,

i.e. within a group of specific processors. A predefined global communicator (`MPI_COMM_WORLD`) is available by default which comprises all the (virtual) processors running the MPI program. Adaptor uses `MPI_COMM_WORLD` to perform communications and also to find out the processor ranks and the number of processors involved in the execution. Note that similar run-time queries are present in the code produced by any HPF compiler in order to arrange the logical processor layout onto the actual physical processor grid and, consequently, distribute data and computations among the processors involved in the execution.

In order to exploit task parallelism, our approach requires that distinct HPF tasks run concurrently on disjoint groups of processors of the same MPI virtual machine. A natural solution is thus to associate a distinct *local* MPI communicator with each HPF task. A self-contained local context is thus supplied to each HPF task, while $COLT_{\texttt{HPF}}$ inter-task communications can use the global communicator `MPI_COMM_WORLD`. An HPF run-time like the one provided by Adaptor, which refers to the global context, is clearly not usable for our purposes. However only small changes are needed to make the run-time context–safe. In fact the changes regards only MPI calls, which have to use the local context, i.e. the MPI communicator local to the task, rather than `MPI_COMM_WORLD`.

***** Place here Figure 1 *****

In order to create the various contexts, initialize the HPF run-times on these contexts, and, finally, load the various HPF tasks, a $COLT_{\texttt{HPF}}$ application is structured as an SPMD program, whose main entry point is the Fortran 77 program illustrated in Figure 1, hereafter called $COLT_{\texttt{HPF}}$ *loader*. Specifically, the *loader* performs the following steps:

1. it initializes MPI and obtains the number of available processors and the relative ranks;

2. it determines group membership information by calling `COLT_MAPPING()`, a subroutine which reads a *configuration file* that defines the various HPF tasks. In particular, the configuration file specifies the number of HPF tasks (`colt_num_tasks`) that make up the parallel application, and, for each task $t_i, i = \{1, \ldots, $`colt_num_tasks`$\}$, the number N_i of processors that have to be reserved for its execution. On the basis of this information, a simple mapping function is established at run–time which allows each HPF task t_i to be associated with logical processors whose global MPI ranks are in the range $[K_{i-1}, K_i)$, where $K_0 = 0$ and $K_i = K_{i-1} + N_i, \forall\ 1 \leq i \leq$`colt_num_tasks`. The subroutine

COLT_MAPPING() returns to the various calling processors colt_num_tasks, i.e. the number of tasks involved, and colt_my_group (colt_my_group= {1,...,colt_num_tasks}), i.e. the identifier of the task to which each processor belongs.

Note that the use of a configuration file allows us to define at run-time the number of processors devoted to the execution of each task, so that the user can change the degree of parallelism of each task without recompiling the application;

3. it creates colt_num_tasks different communicators, one for each group of processors that executes a single HPF task. To this end, MPI provides a powerful function, MPI_COMM_SPLIT, which creates all the communicators that are needed simultaneously, all of them with the same name (COLT_LOCAL_COMM);

4. it calls the HPF run-time initialization routine by passing the disjoint communicators COLT_LOCAL_COMM as an argument. Recall that, for this purpose, the Adaptor HPF run-time was modified to refer to this new local context rather than the global one;

5. it calls the corresponding HPF subroutine on each disjoint group of processors. Note that HPF distributed data structures are declared within the various HPF tasks and are not thus visible at the level of the $COLT_{\texttt{HPF}}$ loader.

2.2 Channel initialization

As mentioned above, $COLT_{\texttt{HPF}}$ optimizes the inter-task communication of distributed data by allowing a pre-computed *communication schedule* to be reused when a given communication has to be repeated several times. Note that this pre-computed information can only be reused if the relative distributions of the data structure to be transmitted do not change either for the sender or for the receiver. If HPF dynamic redistribution directives are exploited, the communication schedule must be recomputed by invoking the appropriate $COLT_{\texttt{HPF}}$ primitives.

To store the communication schedule $COLT_{\texttt{HPF}}$ associates *channel descriptors* with both the ends of a communication channel used to transmit a given distributed array. The descriptors store information on the size and distribution of the array transmitted over a given channel, as well as an optimized

communication schedule which is used to transfer the array contents from the various processors of the source HPF task to the processors of the destination task. To fill the descriptors, $COLT_{\texttt{HPF}}$ provides suitable primitives to be invoked both by the sender and receiver tasks. Below, we illustrate the steps that have to be performed by two interacting HPF tasks in order to prepare a channel for the transmission of an array A:

1. query the HPF run-time support to find out the layout of A on the processor grid associated with their own processor group. To this end, HPF provides an appropriate intrinsic subroutine (`HPF_DISTRIBUTION`). Returned information is stored in the associated descriptor and depends on the number of processors on which the task actually runs, on their layout, on the distribution directives provided by programmers, and also on decisions made by the compiler.

2. exchange the information retrieved at the previous step so that each processor involved knows the layout of A at both the sender and receiver ends*.

3. on the basis of the data layout information, compute the *intersections* between the distributions of array A at the sender and receiver ends by means of the *pitfalls* algorithm [21]. Build, on the basis of the result of the *pitfalls* algorithm, the *Communication Schedule* which is used whenever a communication actually occurs to pack (unpack) the elements of A and send (receive) them to (from) each destination (source) processor. The *Communication Schedule* is stored in the channel descriptor associated with A and clearly differs for each processor. For example, consider a processor p_k which has to send (receive) its local section of A to (from) processors belonging to the destination (source) task. For each receiver (sender) processor p'_j, p_k will store the index ranges (for all the array dimensions) corresponding to local elements of A owned by p'_j as well.

Figure 2 shows the HPF code of two tasks loaded onto two distinct processors groups by the $COLT_{\texttt{HPF}}$ loader illustrated in the previous section (see Figure 1). The two tasks prepare a channel for the transmission of A, a 100×100 array of `REAL`s, which is distributed in different ways on the two groups of

*Note that a deadlock situation may arise when tasks have to prepare several channels, and the invocation order of sends/receives needed to exchange array layout information is not chosen carefully. A simple way to avoid the deadlock is to use an order in which all the (asynchronous) send calls precede all the receive calls on each processor.

processors. The array is then transmitted 50 times from `HPF_Task1` to `HPF_Task2`.

Looking at the code of the two tasks one can note the calls to the $COLT_{\texttt{HPF}}$ primitives which define a communication channel between two HPF tasks. The primitives `COLT_FILL_DESCR` and `COLT_INTERSECT` correspond to steps 1 and 3 above, respectively. The data layout information exchange performed in the step 2 corresponds, on the other hand, to the pair of primitives `COLT_SEND_DESCR` and `COLT_RCV_DESCR`. Finally, the array data exchange is performed by the primitives `COLT_SEND` and `COLT_RCV`.

***** Place here Figure 2 *****

2.3 Data transmission

Our layer supplies "deterministic" primitives to exchange simple and structured data between tasks. Here the term deterministic applies to the receiver partner: in these primitives, in fact, a channel identifier has to be specified, which univocally determines the sender task (see Figure 2).

When a $COLT_{\texttt{HPF}}$ send primitive is invoked to transmit a distributed structure, array data are packed by each processor of the sender group on the basis of the information stored in the channel descriptor, and sent to the processors of the receiver task. In the worst case each processor of the sender task may need to communicate with all the processors of the receiver group. However, the channel descriptor contains all the information needed, so that the processors involved carry out the "minimum" number of point-to-point communications needed to complete the task-to-task communication. Data are sent by means of asynchronous MPI send primitives which use the global communicator and suitable tags. The scheduling ordering of sends tries to prevent several processors of the sender group from simultaneously sending messages to the same processor of the receiver task.

When a corresponding $COLT_{\texttt{HPF}}$ receive primitive is invoked on the receiver task, all the processors of the corresponding group wait for messages from processors of the sender task and read them FIFO. They use the information stored in the descriptor to find out both the number of messages to be received and the relative sources, and to unpack received data.

The exchange of scalar values between tasks is simpler. In fact, no channel setting is needed in this case. Since HPF scalars are replicated on all the processors of a given task, the send primitive is started on

all the processors of the sender task, but only the root processor of the source group actually broadcasts the scalar value to all the processors of the receiver group, while, on the receiver side, all the processors wait to read the message.

2.4 Special messages and non-determinism

Messages often need to be received in a *non-deterministic* way. As we will show in Section 3, an HPF task may need to receive data from a task which is non-deterministically chosen from several possibilities. The problem is that to ensure correctness the same non-deterministic choice must be globally made by all the processors that executes the receiving task. In other words, if a task $\overline{t}$ non-deterministically decides to receive first from task t_i, and then from t_j, this order must be maintained in the point-to-point communications performed by all the processors of $\overline{t}$ in order to accomplish the overall communication.

Our layer thus provides an appropriate receive primitive that causes only the root processor of the receiver group to make the non-deterministic choice of the sender, and then to broadcast its choice to all the other processors of the receiver group. Only when this choice has been communicated to the other receiving processors, can they invoke a *deterministic* primitive (by providing the identifier of the chosen sender) to receive the data from the selected source task as discussed above.

3 Exploiting $COLT_{\texttt{HPF}}$ for structured task parallelism

A programmer can write by hand different task–parallel structures by using $COLT_{\texttt{HPF}}$ primitives to coordinate HPF data-parallel tasks. In other words, $COLT_{\texttt{HPF}}$ can be used as a low-level message–passing API to structure HPF task–parallel programs. However, we do not believe that this is the best way to exploit $COLT_{\texttt{HPF}}$ features. In fact, this programming methodology is too low level, requires a tedious and error prone message–passing programming, and needs programs to be deeply restructured if programmers want to modify the task–parallel structure. We believe that novel programming tools should be provided to allow programmers to express task parallelism in a more portable and high-level way.

In our view programmers should only provide the HPF user-code of the various tasks along with their coordination structure, while a compiling tool should produce the actual code containing the explicit calls

to the $COLT_{\texttt{HPF}}$ layer to express inter-task communications, and the code for task definition and loading (see Section 2.1). Many choices regarding resource allocation, such as the mapping of the various tasks on the machine and the degree of parallelism of each task, might be decided by the compiler tool on the basis of either suitable performance models or directives supplied by the programmer.

As an example of this high–level approach, below we show how $COLT_{\texttt{HPF}}$ can be exploited by a high-level, template-based coordination language that facilitates programmers in organizing HPF tasks according to pipelines and processor farms forms of task parallelism [15, 19]. We present *templates* which implement these forms of task parallelism and we discuss their instantiation by means of a simple compiler.

3.1 Structured parallel programming

A simple high–level language to express the structured parallel programming strategy discussed above, P^3L, has already been proposed elsewhere [3, 8]. P^3L supplies a set of constructs, each corresponding to a specific form of parallelism. There are constructs for control and data parallelism, and they can be composed hierarchically. This means that although the final structure of an application can be seen as a flattened coordination of sequential tasks, it has been obtained by *hierarchically* composing several P^3L constructs. In this paper we propose the adoption of a P^3L-like coordination language, where tasks to be coordinated are data-parallel HPF tasks instead of sequential processes.

***** Place here Figure 3 *****

As an example of the use of a P^3L-like language to express the coordination among HPF tasks, Figure 3.(a) shows the structure of an application obtained by composing five data-parallel tasks according to a *pipeline* structure, where the first and the last tasks of the pipeline only produce and consume, respectively, a data stream. The data type of the input/output channels connecting each pair of interacting tasks is also shown. For example, Task 2 receives an input stream, whose elements are pairs composed of an `INTEGER` and an $N \times N$ matrix of `REAL`s. Of course, the same data types are associated with the output stream elements of Task 1. Figure 3.(b) shows the same application where Task 3 has been replicated to enhance performance, thus exploiting a *processor farm* structure within the original pipeline. In this

case, besides computing their own job by transforming their input data stream into the output one, Tasks 2 and 4 also have to carry out other work related to the presence of a farm structure. In particular, Task 2 has to dispatch the various elements of the output stream to the three replicas of Task 3, while Task 4 has to collect the elements received from the replicas of Task 3.

The P^3L-like code to express Task 3 in Figure 3 would be:

```
task_3 in(INTEGER a, REAL b) out(REAL c(N))
       hpf_distribution(DISTRIBUTE C(BLOCK, *))
       hpf_code_init( <init of the task status> )
       hpf_code( <code that uses a and b, and produce c> )
end
```

Note the input and the output lists of the task, the specification of the layout for distributed parameters, and the HPF user code which initializes the task status and transforms an item of the input stream into an element of the output one. In Figure 3.(b) Task 3 is replicated. This can be expressed by means of a farm construct, whose identical workers are replicas of Task 3:

```
farm foo in(INTEGER a, REAL b) out(REAL c(N))
     task_3 in(a, b) out(c)
end farm
```

Finally, the farm must be composed with the other tasks to obtain the task structure illustrated in Figure 3.(b)[†]:

```
main
   pipe in() out()
        task_1 in()    out(INTEGER a, REAL b(N,N))
        task_2 in(a,b) out(INTEGER c, REAL d)
        foo    in(c,d) out(REAL e(N))
        task_4 in(e)   out(INTEGER f(M))
        task_5 in(f)   out()
   end pipe
end main
```

Note the hierarchical composition of the task-parallel constructs: there is a `pipe`, which invokes a `farm`, which, in turn, invokes a simple HPF data-parallel task. The specification of the structure of the application is concise, simple, and high-level. Moreover, by only modifying this high-level description, a programmer can radically change the parallel structure of the application to test alternative implementa-

[†]For the sake of brevity, the definition of the other tasks of the `pipe` is not reported.

tions. The compiler, on the basis of the input/output lists of the various constructs, can check whether, between consecutive stages in the pipe, data types match.

Note that the high-level code above does not specify the number of processors to be exploited by each task, nor the number of workers of the farm (e.g. the number of replicas of Task 3). Suitable directives could be provided, so that a programmer could tune these parameters to optimize performance and resource allocation. However, since we are concentrating on a set of restricted and structured forms of parallelism, many optimizations based on ad-hoc performance models can be devised [23]. To exploit these performance models for automatic code restructuring and optimization, some information about the costs of the specific application often has to be known [3]. More specifically, the optimization techniques should be based on:

- an analytic performance model of the structures used to exploit task parallelism;
- profiling information about execution times of data-parallel tasks, as well as about data transmission times. This information depends on the particular HPF user code, and on the features of the target machine;
- the number of processors actually available on the target machine.

3.2 Implementation templates

To implement each form of parallelism provided by the proposed high–level coordination language, we have to devise a set of distinct *implementation templates* whose composition realizes the desired task structure. A template can be considered as the code skeleton of an HPF task which cooperates with other tasks according to a fixed interaction pattern. We have thus various different templates which implement, as an example, the first, middle and last stages of a pipeline, or the generic worker of a farm structure. In order to obtain the actual implementation of a user application, the templates corresponding to the chosen parallel structure must be instantiated by inserting the user-provided code, as well as the correct calls to the $COLT_{\texttt{HPF}}$ primitives which initialize the communication channels and exchange data between tasks. The set of instantiated templates thus entirely define the HPF tasks of the user application, and only have to be compiled and linked with the $COLT_{\texttt{HPF}}$ loader (see Section 2.1) to obtain the application

executable code. Note, in fact, that in our approach the instantiated templates are exactly the HPF subroutines called by the $COLT_{\texttt{HPF}}$ loader on the various groups of MPI processors.

Below we present some templates which implement pipeline and processor farm forms of parallelism, and we briefly describe the instantiation of the template of a generic pipeline stage starting from the high-level specification of the stage itself.

3.2.1 A pipeline template and its instantiation

A *pipeline* structure is a chain of data-flow stages, which consumes and produces an input and an output data stream, respectively. Thus the code scheme of a generic pipeline stage, i.e. its implementation template, has to be organized as a loop that receives an element of the input stream, executes some user code, and, finally, sends the corresponding element of the output stream.

Moreover, if the length of the processed stream is unknown until run–time, a distributed termination protocol must be implemented within the templates to propagate termination information along the pipeline stages. To this end we associate an incremental mark with each element of the stream. The transmission of this mark precedes the communication of the associated stream element between any pair of HPF stages. The stream termination is thus signaled by the reception of a particular `END_OF_STREAM` mark. On the other hand, if the length of the stream is statically known, a template that does not adopt the above distributed termination protocol can be exploited instead.

***** Place here Figure 4 *****

Figure 4 shows the $P^3L - like$ specification of a generic pipeline stage and the instantiation of the corresponding template. We are thus assuming that the instantiation is carried out by a compiler associated with the $P^3L - like$ coordination language outlined in Section 3. Note that the input/output lists of data, along with their distribution directives, are used by the compiler to generate an include file `typedef_distribution.inc`. Moreover, the declaration of task variables along with the relative code that must be executed for their initialization, is included in another file, `init.inc`. Finally, the code to be executed to consume/produce the data streams is contained in the include file `body.inc`. These files are directly included in the source code of the template which is also shown in the figure. To complete the

instantiation of the template, the appropriate calls to the $COLT_{\texttt{HPF}}$ layer which initialize the input/output channels (see Section 2.2) and send/receive the elements of the input/output stream (see Section 2.3), also have to be generated and included. The correct generation of these calls relies on knowledge of the task input/output lists, as well as the mapping of the tasks on the disjoint groups of processors.

The template shown in Figure 4 regards a task that is in the middle of a pipeline chain. The templates for the first and the last tasks of the pipeline are slightly different. In fact, the first one has to communicate with the following task in the chain alone, while the last one has to receive data solely from the previous task in the chain. Moreover, the first task has to generate the incremental marks associated with stream elements and the `END_OF_STREAM` mark which propagates termination.

3.2.2 Nesting processor farm structures within a pipeline

The *processor farm* form of parallelism ensures that data coming from an input stream are dispatched to a set of workers, which produce the corresponding output stream. The processor farm structure is conceived as being used to increase the bandwidth of a "slow" pipeline stage when the alternative method of increasing the number of processors exploited by its data-parallel implementation is not suitable.

Inserting a farm into a pipeline structure entails changing the templates used to implement the tasks executed by the stages that precede and follow the farm structure. We call these two tasks, i.e. the one preceding and the one following the farm structure, *emitter* and *collector*, respectively.

***** Place here Figure 5 *****

As an example of hierarchical compositions of pipelines and farms structures, consider Figure 5. Figure 5.(1) shows the structure of a simple pipeline application composed of five data-parallel stages, namely Tasks 1-5. Note the numbers close to each circle representing a given task: these numbers are the identifiers of the processor groups, i.e. the set of processors, onto which each task has been mapped. Figure 5.(2) shows the same pipeline where the bandwidth of the second stage has been increased by replicating the stage in three copies, according to a farm structure. Here, in terms of composition of diverse forms of parallelism, the outermost pipeline has been hierarchically composed with a farm, which, in turn, exploits a single data-parallel task (Task 2). Finally, Figures 5.(3) and 5.(4) show the same

pipeline where the third stage has been replicated as well. In the former case, the outermost pipeline is hierarchically composed with two farms, which exploit, respectively, replicas of Task 2 and Task 3. In the latter case (Figures 5.(4)), the outermost pipeline is hierarchically composed with a single farm, whose workers are in turn structured as a pipeline made up of two stages (Task 2 and Task 3).

This example clearly shows the advantage of exploiting a high-level coordination language like the one proposed in Section 3. In this case in fact, all the transformations outlined in the figure only entail changing the nesting of the high-level language constructs and recompiling the program. On the other hand, if template instantiations and code restructuring is done manually, the programmer's job becomes harder and error-prone.

We now outline the organization of the templates that implement the structures illustrated in Figure 5. Consider first the case of a farm inserted in a pipeline (see Figure 5.(2)). Note that the various workers are mapped on distinct processor groups, each composed of the same number of processors. The templates used to implement the emitter (Task 1) and the collector (Task 3) are slightly different from the generic template of a pipeline stage, which was discussed in Section 3.2.1. First of all, note that the emitter and collector have to establish a channel with all the workers (each being a replication of Task 2) rather than with a single preceding/following task. However, since the processor and data layouts of all the workers are identical, both the emitter and the collector only need to build a single *Communication Schedule* which can be reused for all the communications with the various workers. Another difference regards the management of distributed termination when the stream's length is not known a priori. In this case the emitter has to communicate the `END_OF_STREAM` mark to all the workers of the farm in order to signal that they can terminate. Moreover, since all the workers communicate this mark to the collector before their termination, the collector, before propagating termination, has to wait for the reception of all these `END_OF_STREAM` marks.

In the implementation of the template of a generic collector we need to use the special primitives illustrated in Section 2.4. In fact, the collector has to wait for a new stream element from any of the farm workers, so that it has to receive the mark non-deterministically, thus choosing the processor group from which the corresponding stream element must be received. Once the sender group has been chosen, the collector

can use deterministic primitives (see Section 2.3) to complete the reception of the stream data element. There are also differences in the cooperation between the emitter and the workers. The emitter has to make scheduling decisions to choose the worker to which its current element of the output stream has to be sent. We implemented two distinct scheduling policies. The simplest one is *Round Robin*, in which the emitter always sends the element marked by j to the worker identified by $(j \bmod n_w)+1$, where n_w is the number of workers. The other policy is *Self Scheduling*, where a new element of the stream is sent to a given worker only when it is ready to receive it. As discussed in Section 4, this kind of dynamic scheduling is very important when jobs executed by the workers are characterized by non-uniform execution times. Figure 5.(2) illustrates a farm that exploits self scheduling. Note the dashed arrows from the workers to the emitter: they represent *request signals* sent by the workers to the emitter to *self schedule* a new job[‡]. Also in this case, the signal messages have to be received non-deterministically. At the beginning of the execution, however, the emitter sends a distinct element of the data stream to each worker without waiting for a request signal. In this way, the overhead to start workers is reduced, and the workers may soon begin computing. The emitter then enters a loop in which it sends a new stream element to a worker only if a corresponding request signal has been received, while the workers exploit job prefetching by sending a request signal as soon as they receive a new stream element from the emitter. Note that, due to the uneven finishing times of the workers, the ordering of the stream elements arriving at the collector may be different from the original one. The marks associated with each stream element can be used to restore the original ordering if necessary.

From the discussion above, one can easily imagine the implementation of the structure shown in Figure 5.(4). On the other hand, the case illustrated in Figure 5.(3), where two farms are composed one after another in a pipeline chain, requires further comments. This structure is useful when both execution times of Tasks 2 and 3 are not uniform, so that we need to schedule incoming input stream elements dynamically. In particular, while Task 1 has to dynamically schedule stream items to the workers implementing Task 2, these in turn have to schedule their output stream items to the various replicas of Task 3. To solve this scheduling problem we need to introduce a *broker process*, which is shown in

[‡]Request signals are not needed in the template which exploits Round Robin.

Figure 5.(3) as a small circle mapped on Group 10 (a singleton group). The broker exchanges information with the various replicas of Task 2 and Task 3 in order to dynamically route stream elements produced by each replica of Task 2 to one of the replicas of Task 3. Note that, to avoid making Figure 5.(3) confusing, the arcs corresponding to information exchanges of the broker with the various replicas of Tasks 2 and 3 are not shown.

The template to adopt to implement Task 2 is, in this case, slightly different from the one exploited for a generic worker. In fact, each replica of Task 2 asks the broker for a group identifier to which its next stream item has to be transmitted. Conversely, each replica of Task 3 has to send its request signal to the broker (rather than to the farm emitter). It then waits for a new stream item but, since it can arrive from any of the replicas of Task 2, it must be received non-deterministically.

4 Experimental results

To validate our approach and quantify the costs associated with the implementation of $COLT_{\texttt{HPF}}$, we used both synthetic micro-benchmarks and sample applications. The synthetic micro-benchmarks were used to characterize the performance of the implementation by measuring the costs associated with the communication of distributed data among concurrent data-parallel tasks, and to demonstrate the effectiveness of the scheduling strategies exploited by the farm templates. The applications were used to show the usefulness of our approach and the performance improvement resulting from the exploitation of a mixture of both task and data parallelism with respect only to exploit data parallelism with a pure HPF implementation. To compare the results, the same HPF compiler (Adaptor), the same data layouts and the same parallelization strategy (except the task parallelism, of course) were used for both the pure HPF and $COLT_{\texttt{HPF}}$ implementations of the sample applications.

***** Place here Figure 6 *****

4.1 Synthetic micro-benchmarks

The first micro-benchmark implemented measures the time required to exchange a distributed array between two data-parallel tasks. We executed this sort of "ping-pong" program with 1-D arrays distributed

(BLOCK) in the source task and (CYCLIC) in the receiver task, and 2-D arrays distributed (*,BLOCK) and (BLOCK,*) in the source and destination tasks, respectively. Experiments were executed on an SGI/CRAY T3E by varying both the size of the exchanged arrays and the number of processors within each data-parallel task. We measured the time of the slowest processor and divided this time by the number of communications accomplished to obtain the average time per each communication. The results are reported in Figure 6. The plots reported in Figure 6.(a) show the time required to communicate 1-D arrays between two tasks, where the arrays are block partitioned among the sender processors and cyclically distributed within the destination task. As can be seen, there is a small increase in the communication times measured when two processors are exploited within each task with respect to the case with one processor. This is due to the different data layout exploited which, if several processors are used for each task, entails packing non-contiguous data before sending them. Moreover, communication latencies tend to increase with the number of per-task processors for small array sizes, while the opposite effect was measured in the tests involving large volumes of data. This behavior can also be noted in the plots shown in Figures 6.(b) and 6.(c), which report the results obtained with 2-D arrays. In all these plots communication latency decreases up to a minimum and then tends to increase slightly. For example, for 2-D arrays of 512 KB (see Figure 6.(b)) note that communication latency decreases up to the 16 per-task processors case, and then increases when 32 processors are exploited. With very large arrays (e.g. 8, 32 MB) the decrease is constant up to the maximum number of per-task processors tested. The curves thus behave as expected: for small data volumes the communication startup time dominates the overall latency, while for larger arrays the main contribution to communication latency is given by the message transfer time. Note that the transfer time is directly proportional to the length of messages transmitted and thus indirectly proportional to the number of processors onto which the exchanged array is distributed.

The other micro-benchmark implemented measures the effectiveness of our processor farm implementation template. To this end we built a synthetic pipeline application composed of three stages. The first stage produces a stream of arrays and sends one array to the next stage every T_1 seconds. The second stage is replicated in five copies according to a processor farm structure. It performs on each

array received a dummy computation $\mathcal{C}$, before forwarding it to the third stage. Finally, the third stage simply consumes an element of the stream received from the workers of the farm every T_3 seconds. This application was implemented with different templates thus producing two different versions: the first version exploits a *Round Robin* (RR) technique to dispatch the jobs to the five replicas of the second stage, while the second version exploits the *Self Scheduling* (SS) technique described in Section 3.2.2.

Moreover, with both the versions we conducted three series of experiments by changing the cost T_2^i of the dummy computation $\mathcal{C}$ performed in the second stage on the $i-th$ stream element. In the first series of experiments the costs T_2^i were determined according to an exponential distribution with average μ, in the second series we used a uniform distribution with the same average, and, finally, the costs T_2^i used in the third series of experiments were exactly μ for all the stream elements. Other assumptions regard the value of μ which was forced to be equal to 0.2, 0.4 and 0.8 seconds and the parameters which were fixed for all the experiments. In particular we used four processors within each data-parallel task, we forced $T_1 = T_3 = \mu/5$ (μ divided by the number of farm workers) to balance the pipeline stages, and we fixed to 400 the number of stream elements processed where each element is a 256×256 array of 8 byte integers.

***** Place here Table I *****

Table I reports the results obtained in these tests as a ratio between the overall execution times obtained with the RR version w.r.t. the SS version. The SS version gave performances from 13% to 14% better than the RR one for exponentially distributed values of T_2^i. The improvements ranged instead from 11% to 12% in the case of uniformly distributed costs, while in the balanced case (i.e. $T_2^i = \mu, \forall i$), the difference between the results of the two implementations is negligible with a slight performance loss measured for the SS version (see the fourth column of Table I).

These results demonstrate the utility of employing dynamic scheduling strategies when the computational costs are non-uniform and unknown until run-time. On the other hand, when execution times are uniform, and thus no dynamic scheduling should be needed, the overheads introduced by our implementation of self scheduling are negligible (less than 1%). In the next section we will show the effectiveness of the farm template by exploiting self scheduling for the implementation of a real application as well.

4.2 Sample applications

Two sample applications were implemented by instantiating the templates described in Section 3.2. The first one is a classical 2-D Fast Fourier Transform (FFT) which is probably the application most widely used to demonstrate the usefulness of exploiting a mixture of both task and data parallelism [10, 12]. The second sample application considered is a complete high-level computer vision application which detects in each input image the straight lines that best fit the edges of the objects represented in the image itself.

2-D Fast Fourier Transform. FFT transformations are commonly used in the field of signal and image processing applications which generally require the FFT to be applied in real–time to a stream of frames acquired from an external device. Given an $N \times N$ array of complex values, a 2-D FFT entails performing N independent 1-D FFTs on the columns of the input array, followed by N independent 1-D FFTs on its rows. An HPF code implementing a 2-D FFT thus has the following structure:

```
        complex A(N,N)
!HPF$   distribute (*,BLOCK):: A
        ......
        do while <End of STREAM>
C          read a new input STREAM elem
           call read (A)
!HPF$   INDEPENDENT
           do icol=1,N
              call fft_slice(A(:,icol))
           end do
           A = transpose(A)
!HPF$   INDEPENDENT
           do icol=1,N
              call fft_slice(A(:,icol))
           end do
C          write a new output STREAM elem
           call write (A)
        end do
        ......
```

Note that the 2-D array `A` is distributed over the second dimension while its first dimension is collapsed. This allows the first 1-D FFT to be applied in parallel to each memory-contiguous column of `A`. After the first `independent` loop, matrix `A` is transposed and the 1-D FFT is performed again in parallel on the columns of `A`. No communications are generated by the HPF compiler within the two parallel loops,

while matrix transposition involves *all-to-all* communications.

By exploiting the coordination language proposed in Section 3, we can easily structure the 2-D FFT application as a two-stage pipeline in the following way:

```
pipe in() out()
     stage1   in()  out(COMPLEX A(N,N))
     stage2   in(A) out()
end pipe
```

where the two stages are defined as follows:

```
stage1 in() out(COMPLEX A(N,N))
    hpf_distribution(DISTRIBUTE A(*,BLOCK))
    hpf_code_init(.....)
    hpf_code(
C       read a new input STREAM elem
        call read (A)
!HPF$   INDEPENDENT
        do icol=1,N
           call fft_slice(A(:,icol))
        end do
    )
end
```

```
stage2 in(COMPLEX B(N,N)) out()
    hpf_distribution(DISTRIBUTE B(BLOCK,*))
    hpf_code_init(.....)
    hpf_code(
!HPF$   INDEPENDENT
        do irow=1,N
           call fft_slice(B(irow,:))
        end do
C       write a new output STREAM elem
        call write (A)
    )
end
```

Matrix transposition is not necessary in this case. In fact, due to the `(BLOCK,*)` distribution, the second stage can perform the N independent 1-D FFTs on the matrix rows without communications.

***** Place here Figure 7 *****

Figure 7 shows the per input array execution times for different problem sizes obtained on an SGI/Cray T3E with HPF and $COLT_{\texttt{HPF}}$ implementations of the 2-D FFT. The results are plotted as a function of the number of processors used, where if P is the total number of processors, the $COLT_{\texttt{HPF}}$ implementation exploits $P/2$ processors for both the first and second stages of the pipeline. As can be seen, the $COLT_{\texttt{HPF}}$ implementation considerably outperforms the HPF one in all the tests conducted. The better performance of the mixed task and data-parallel implementation is highlighted by Table II which reports the ratio between the HPF and $COLT_{\texttt{HPF}}$ 2-D FFT execution times for different sizes of the problem and numbers of processors exploited. The performance improvement obtained is significant, and ranges from 11% to 134%. The largest improvements were obtained when 32 or 64 processors were used on small/medium

sized problems. This behavior is particularly interesting because many image and signal processing applications require the 2-D FFT to be executed in real-time on data sets whose size is limited by physical constraints (e.g. the circuitry of the video camera or of other input devices) [14].

***** Place here Table II *****

***** Place here Figure 8 *****

High-level computer vision. The second sample application used to validate our approach is a high-level computer vision application which detects in each input image the straight lines that best fit the edges of the objects represented in the image itself. For each grey-scale image received in input (for example, see Figure 8.(a)), the application enhances the edges of the objects contained in the image, detects the straight lines lying on these edges, and finally builds a new image containing only the most evident lines identified at the previous step. The application can be naturally expressed according to a pipeline structure. The first stage reads from the file system each image, and applies a low-level Sobel filter to enhance the image edges. Since the produced image (see Figure 8.(b)) is still a grey-scale one, it has to be transformed into a black-and-white bitmap (see Figure 8.(c)) to be processed by the following stage. Thus an inexpensive thresholding filter is also applied by the first stage before sending the resulting bitmap to the next stage. The second stage performs a Hough transform, a high-level vision algorithm which tries to identify in the image specific patterns (in this case straight lines) from their analytical representation (in this case the equations of the straight lines). The output of the Hough transformation is a matrix of accumulators $H(\rho, \theta)$, each element of which represents the number of black pixels whose spatial coordinates (x, y) satisfy the equation $\rho = x \cos\theta + y \sin\theta$. Matrix H can be interpreted as a grey-scale image (see Figure 8.(d)), where lighter pixels correspond to the most "voted for" straight lines. Finally, the third stage chooses the most voted for lines, and produces an image where only these lines are displayed. The resulting image (see Figure 8.(e)) is then written in an output file.

***** Place here Table III *****

Table III illustrates some results of experiments conducted on an SGI/Cray T3E. It shows the completion times of each of the three stages, where the input stream is composed of 60 256×256 images. Note

that the I/O times of the first and the third stage do not scale with the number of processors used. If the total completion times reported in the table are considered, it is clear that there is no point exploiting more than 4/8 processors for these stages. On the other hand, the Hough transform stage scales better. We can thus assign enough processors to the second stage so that its bandwidth becomes equal to that of the other stages. For example, if we use 2 processors for the first stage, we should use 4 processors for the third stage, and 16 for the second one to optimize the throughput of the pipeline. Alternatively, since the cost of the Hough transform algorithm very much depends on the input data [7], we may decide to exploit a processor farm for the implementation of the second stage. For example, a farm with two replicated workers, where the bandwidth of each worker is half the bandwidth of the first and the last stages, allows the overall pipeline throughput to be optimized, provided that a dynamic self scheduling policy is implemented to balance the workers' workloads.

******* Place here Table IV *******

Table IV shows the execution times and the speedups measured on a Cray T3E executing our computer vision application, where we adopted a processor farm and self-scheduling for the second stage of the pipeline. The column labeled *Structure* in the table, indicates the mapping used for the $COLT_{\texttt{HPF}}$ implementations. For example, [4 (8, 8) 4] means that 4 processors were used for both the first and last stages of the pipeline, while each one of the two farm workers was run on 8 processors. The table also compares the results obtained by the $COLT_{\texttt{HPF}}$ implementations with those obtained by pure HPF implementations exploiting the same number of processors. The execution times measured with the $COLT_{\texttt{HPF}}$ implementations were always better than the HPF ones. The performance improvements obtained are quite impressive and range from 60% to 160%.

5 Related work

There has been increasing interest in the promising possibility of exploiting a mixture of task and data parallelism, where data parallelism is restricted within HPF tasks and task parallelism is achieved by their concurrent execution. Extensions for the exploitation of task parallelism have also been introduced in the specifications of the new HPF 2.0 standard [17]. These extensions were inspired by a proposal

of the research group involved in the Fx project at CMU [24]. According to the HPF 2.0 standard, task parallelism is introduced by allowing `TASK_REGION` blocks to be specified. Within a `TASK_REGION`, we may specify through `ON` blocks that a set of HPF tasks has to be executed concurrently on disjoint subsets of processors, where each task only accesses data distributed on the associated processor subset. Communications between tasks are accomplished by simple assignments outside `ON` blocks, but inside the outer `TASK_REGION` block. The main advantage of this approach, besides being proposed by the HPF Forum as a standard to be hopefully adopted by the industry, is its adherence to the general philosophy followed by HPF. In this approach, in fact, any HPF source program can also produce a correct sequential executable if a Fortran 90 compiler is exploited. Moreover, since tasks are started at run-time by `ON` directives inside a `TASK_REGION`, the assignment of the various processors subsets can be changed dynamically. For example, at the beginning all processors can be used to carry out a single data-parallel computation, and after, i.e. when they enter a `TASK_REGION` block, they may selectively execute one of the concurrent tasks specified. On the other hand, its main disadvantage is, in our opinion, the excessive deterministic behavior of task communications. In fact, the use of explicit assignments for inter-task communications does not allow programmers to express non-deterministic behaviors. For example, a programmer cannot specify an HPF task which, non-deterministically, waits for input data from several other HPF tasks, even though this communication pattern may occur very often in task parallel programs. Finally, the low-level mapping of tasks is worth noting. Programmers have to explicitly specify processor sub-grids on which tasks must be mapped, even though, as usual, the processor number can be queried at run-time, and processor sub-grids can be expressed in terms of this number.

Another interesting proposal has appeared in the literature [22], which, like HPF 2.0, requires changes to the original HPF 1.0 language and the associated compiler. The proposal adopts language directives similar to the ones previously proposed by the Fx research group [13], and requires programmers to specify the input/output list for all the tasks involved rather than explicitly providing assignments to express communications. Here tasks can either be `PURE` HPF-like subroutines or simple statements. Since programmers don't specify either the allocation of tasks on specific subsets of processors, or explicit communications/assignments between tasks, the compiler has to (1) extract a data dependencies graph of the tasks, (2) decide the best allocation of these tasks, and (3) insert the right communications between

tasks. This approach is much more high-level than the one proposed for HPF 2.0, where a programmer has to explicitly specify processor sub-grids on which tasks are mapped, but it does require sophisticated compiling techniques for task allocation. Nevertheless, the type of task interaction that can be specified is still deterministic, as in HPF 2.0. A disadvantage with respect to HPF 2.0 is that the resulting source code is no longer a syntactically correct Fortran 90 program.

Opus [11] is an object-oriented language that allows task and data-parallelism to be integrated. By adopting Opus one can define classes of objects, called *ShareD Abstractions* (SDAs), using a syntax similar to that of HPF. Each instance of an SDA encapsulates distributed data and methods, where methods have exclusive access to encapsulated data. Data parallel tasks are thus started by creating instances of specific SDAs, while the inter-task cooperation takes place by means of remote synchronous (or asynchronous) method invocations. Note that SDA instances are started dynamically by a so called *coordination* task, so that the run-time that implements inter-task communication has to control passing distributed data structures from one task to another, including any possible remapping that might be needed. The run-time accomplishes this through a handshaking protocol, which exchanges the distribution information about the actual argument (on the caller SDA) and the formal one (on the callee SDA) of a given method. Note that this handshaking protocol is very similar to the $COLT_{\texttt{HPF}}$ protocol to create a channel between two tasks. Finally, even though in this paper we envision the use of $COLT_{\texttt{HPF}}$ to start and coordinate a statically fixed set of HPF tasks, if more dynamic behaviors are needed to implement other forms of parallelism, the $COLT_{\texttt{HPF}}$ loader could be modified to support dynamic task creation. Unfortunately, the version of $COLT_{\texttt{HPF}}$ discussed in this paper exploits MPI which does not allow virtual processors to be dynamically added/deleted. Thus dynamically created data parallel tasks can only exploit subsets of the processors statically included in the MPI virtual machine. We are porting $COLT_{\texttt{HPF}}$ to exploit PVM, which, due to its powerful dynamic features, will permit us to implement much more dynamic behaviors.

Another notable example of a language attempting to integrate task and data parallelism is Fortran M, a message–passing language, which has been proposed as a coordination language for Fortran and HPF tasks [5]. Such an integration of Fortran M with HPF is still only a proposal. The authors noted that a "clean" integration poses several difficulties. For example, the necessity of extending HPF with

SEND and RECEIVE operations over Fortran M channels, though they propose an alternative approach based on the use of a message–passing library, i.e. the use of HPF *extrinsic procedures* invoking Fortran M mechanisms. Note that the last approach departs from the original proposal of having a language where the structures and the mechanisms for supporting message–passing are all defined within the language, thus making accurate compile-time analysis possible.

In our opinion, the adoption of a message–passing paradigm to express HPF task parallelism is too "low-level". It requires large and error-prone code restructuring when a program has to be tuned to better harness a given target architecture. Consider, for example, the code re-writing needed to introduce a processor farm structure within a pipeline, where the code to implement dynamic scheduling of incoming jobs is in charge of programmers. On the other hand, although our proposal employs a message–passing layer such as $COLT_{\texttt{HPF}}$, it requires very little programming to structure parallel applications whose structure matches the parallel forms provided by the high-level coordination language. All the communication and scheduling code needed to implement task cooperation is in fact automatically generated by the compiler on the basis of the corresponding implementation templates.

Foster et al. [12] proposed another message–passing approach to exploit task parallelism within HPF. Differently from the Fortran M approach, this proposal regards the binding of a *standard message–passing library* with HPF. More precisely, they propose a framework in which concurrent HPF tasks communicate by means of a restricted subset of MPI communication primitives. Below we will discuss this approach in some depth, above all to point out usage complexities and semantics flaws of the proposed HPF-MPI binding.

The proposed implementation, like the $COLT_{\texttt{HPF}}$ library, uses the `EXTRINSIC` subroutine mechanism and aims to be portable between different HPF compilers. To this end, their HPF-MPI does not access HPF system's internal data structures, but, like $COLT_{\texttt{HPF}}$, uses standard functionalities, available in any HPF-compliant implementation, to find out array data distribution and to start SPMD computations within data-parallel HPF tasks. However, their primitives are coded in C and they use the `pghpf` compiler, which adopts a non-contiguous internal representation for the local portion of distributed arrays. This causes a loss in performance, since arrays must be copied into contiguous temporary C-style arrays before entering the extrinsic subroutine, and copied back upon return. In our case there is no such cost because

$COLT_{\texttt{HPF}}$ primitives are `EXTRINSIC HPF_LOCAL` subroutines which exploit for the data the same internal representation used by the Adaptor run–time.

All the steps illustrated in Section 2.2 to set up communication channels are accomplished by their implementation as well, which, like $COLT_{\texttt{HPF}}$, uses Ramaswamy and Banerjee's algorithm [21] to build the *Communication Schedule*. However, they state that these steps should be repeated for all calls of their HPF-MPI communication primitives. Of course, when an array has to be repeatedly exchanged between two tasks, and its distribution does not change from one communication to another, it would be better to reuse the same *Communication Schedule* to carry out several communications. For this reason Foster et al. also propose an HPF binding for persistent MPI communications. They propose the use of `MPI_Send_init` and `MPI_Recv_init` to set up the channel and build the *Communication Schedule*, and then of `MPI_Start` to actually perform the communication.

Consider first the case of standard MPI communications which require information on array distribution to be exchanged at each communication. This implies that the implementation of `MPI_Send` also receives the mapping information from the destination task. In other words, if the matching `MPI_Recv` is not executed by the partner task, the sender task will block itself on the `MPI_Send`. In practice, this HPF binding of `MPI_Send` has a synchronous semantic, although most MPI implementations provide buffered and asynchronous communication primitives because asynchronous communications make it easier to write efficient and deadlock-free programs.

As regards persistent communications, we have noted a semantics flaw with respect to the MPI standard, according to which a persistent request (i.e. `MPI_Send_init` or `MPI_Recv_init`) can be thought of as a communication port or a "half-channel" [20]. In fact, it does not provide the full functionalities of a conventional channel, since there is no binding of the send port to the receive port. For example, the sender may invoke `MPI_Send_init` and `MPI_Start`, while the receiver continues to use `MPI_Recv` to read the messages. Conversely, Foster et al. propose to use these persistent subroutines to prepare the complete channel, and thus for each `MPI_Send_init` there must be a matching `MPI_Recv_init`. Moreover, the primitives used to create persistent communication requests have a synchronous semantics, so that they must be invoked in a given order to avoid deadlocks, thus making it difficult for programmers to use them.

6 Conclusions

In this paper we have discussed $COLT_{\texttt{HPF}}$, a run-time support to coordinate HPF tasks, and its implementation on top of a public domain HPF compiler, Adaptor. Since standard HPF mechanisms to query data distribution and to start SPMD computations have been used, we claim that $COLT_{\texttt{HPF}}$ can be easily ported in order to exploit other HPF compilers.

More importantly, we have shown how $COLT_{\texttt{HPF}}$ can be exploited to design implementation templates for common forms of parallelism, and how these templates can be used by a compiler of a structured, high-level, template-based, coordination language. We have discussed the benefits for programmers in using such a coordination language to organize the cooperation among HPF tasks according to pipeline and processor farms structures, which are recurrent forms of parallelism in many application fields. Knowledge of the specific forms of parallelism employed to structure a given application, where the forms of parallelism allowed belong to a restricted set, can also be exploited by the compiler to statically optimize resource allocation on the basis of suitable performance models and application costs.

Finally, we have presented some encouraging performance studies, conducted on an SGI/Cray T3E. For some experiments we used synthetic benchmarks in order to show the communication performance of $COLT_{\texttt{HPF}}$, as well as the effectiveness of the various scheduling policies that were adopted to implement processor farms. For other experiments we used real sample applications. We structured these applications in order to exploit a mixture of task and data parallelism, and we compared these implementations with pure data parallel ones. We have presented a classical 2-D Fast Fourier Transform, which was structured as a two-stage pipeline, and a complete computer vision application where a combination of pipeline and processor farm structures was adopted. We observed that the mixed task/data-parallel versions of such applications always achieved performance improvements over the pure data parallel counterparts. These improvements ranged from a few per cent up to 160%.

In the experiments in which we varied data set sizes (e.g. 2-D Fast Fourier Transform), we noted that the largest improvements were obtained when many processors were used to execute experiments on small/medium sized problems. This behavior is particularly interesting because for many interesting applications, e.g. in image and signal processing applications, the size of the data sets is limited by

physical constraints which cannot be easily overcome [14].

Moreover, when the application is not CPU-bound but performs many I/O operations (e.g. our computer vision application), the best organization was to run the I/O-bound parts of the application on a few processors, and to reserve most processors for computing intensive parts which normally scale better with the number of processors.

Future work regards the implementation of the compiler for the proposed high-level coordination language. This work will not begin from scratch. An optimizing compiler for SKIE (SKeleton Integrated Environment), a P^3L-like coordination language [3], has already been implemented within the PQE2000 national project [25]. We only have to extend the compiler to support HPF tasks and provide suitable performance models which will allow the compiler to perform HPF program transformations and optimizations. As a result of this integration, programmers will be able to easily structure their parallel applications by hierarchically composing simpler tasks, where each atomic task can be a sequential process (specified in C, C++, Fortran, or Java), as well as a data-parallel HPF task. A graphic tool aimed to help programmers in structuring their $COLT_{\texttt{HPF}}$ applications by hierarchically composing the provided primitive forms of parallelism is also under development. The tool can be easily extended to also support unstructured forms of task interactions, in a similar way as the AVS [1] visual programming environment does. The AVS system, however, does not support HPF modules and related optimized inter-module communications, and does not allow modules to be structured according to high level forms of task parallelism as pipelines and processor farms.

Acknowledgments

Our greatest thanks are for Thomas Brandes, for many valuable discussions about task parallelism and Adaptor. We also wish to thank the support of PQE2000 project, Ovidio Salvetti for his suggestions about the computer vision application and the CINECA Consortium of Bologna for the use of the SGI/Cray T3E.

References

[1] AVS: Advanced Visual Systems Inc. URL: `http://www.avs.com`.

[2] S. T. Chanson A. S. Wagner, H. V. Sreekantaswamy. Performance Models for the Processor Farm Paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), May 1997.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[4] T. Brandes. ADAPTOR Programmer's Guide Version 5.0. Internal Report Adaptor 3, GMD-SCAI, Sankt Augustin, Germany, April 97.

[5] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C-W. Tseng. Integrated Support for Task and Data Parallelism. *The Int. Journal of Supercomputer Applications*, 8(2):80–98, 1994.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitmann/MIT Press, 1989.

[7] S. C. Orphanoudakis D. Gerogiannis. Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), Sept. 1993.

[8] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A Methodology for the Development and Support of Massively Parallel Programs. In D. B. Skilliconr and D. Talia, editors, *Programming Languages for Parallel Processing*, chapter 7, pages 319–334. IEEE Computer Society Press, 1994.

[9] J. Darlington et al. Parallel Programming Using Skeleton Functions. In *Proc. 5th Int. PARLE Conf*, pages 146–160, Munich, Germany, June 1993. LNCS 694, Spinger-Verlag.

[10] P. Dinda, T. Gross, D. O'Halloron, 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.

[11] B.Chapman et al. Opus: a Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(2), April 1997.

[12] Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 45(2):148–158, Sept. 1997.

[13] 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.

[14] T. Gross, D. O'Halloron, E. Stichnoth, and J. Subhlok. Exploiting task and data parallelism on a multicomputer. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[15] A.J.G. Hey. Experiments in MIMD Parallelism. In *Proc. Int. Conf. PARLE '89*, pages 28–42, Eindhoven, The Netherlands, June 1989. LNCS 366 Spinger-Verlag.

[16] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[17] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Jan. 1997. Version 2.0.

[18] C.H. Koebel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[19] H.T. Kung. Computational Models for Parallel Computers. In C.A.R. Hoare Series, editor, *Scientific applications of multiprocessors*, pages 1–17. Prentice-Hall Int., 1988.

[20] Message–Passing Interface Forum. *MPI: A Message–Passing Interface Standard*, June 1996. version 1.1.

[21] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proc. Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, Feb. 1995.

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

[23] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, June 1996.

[24] J. Subhlok and B. Yang. A New Model for Integrating Nested Task and Data Parallelism. In *Proc. ACM Symp. Principles and Practice of Parallel Programming*, pages 1–12. ACM Press, New York, 1997.

[25] M. Vanneschi. Heterogeneous HPC Environments. In *Proc. 4th Int. Conf. Euro-Par'98*, pages 21–34, Southampton, UK, Sept. 1998. LNCS 1470 Spinger-Verlag.

```
      program COLT_LOADER
      .....
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, colt_my_proc, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, colt_num_proc, ierr)
C     determine group membership info on the basis of mapping information
      call COLT_MAPPING(colt_my_proc, colt_num_proc, colt_my_task, colt_num_tasks)
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, colt_my_task, colt_my_proc, COLT_LOCAL_COMM, ierr)
      call COLT_HPF_INIT(COLT_LOCAL_COMM)
      if (colt_my_group .eq. 1) then call HPF_task1()
      if (colt_my_group .eq. 2) then call HPF_task2()
      .....
      call COLT_HPF_EXIT ()
      END
```

Figure 1: Pseudo-code of the $COLT_{\texttt{HPF}}$ loader.

```
       SUBROUTINE HPF_Task1
       real A(100,100)
!HPF$  distribute A(BLOCK,*)
       ......
       call COLT_INIT()

C      set up channel CH_OUT from this
C      task to task identified by DEST
C      to transmit array A
       call COLT_FILL_DESCR(A, CH_OUT)
       call COLT_SEND_DESCR(DEST, CH_OUT)
       call COLT_RCV_DESCR(DEST, CH_OUT)
       call COLT_INTERSECT(CH_OUT)
       ......

       do i=1,50
          <produce A>
C         send A
          call COLT_SEND(A, CH_OUT)
       end do
```

```
       SUBROUTINE HPF_Task2
       real A(100,100)
!HPF$  distribute A(*,BLOCK)
       ......
       call COLT_INIT()

C      set up channel CH_IN from the task
C      identified by SRC to this task
C      to transmit array A
       call COLT_FILL_DESCR(A, CH_IN)
       call COLT_SEND_DESCR(SRC, CH_IN)
       call COLT_RCV_DESCR(SRC, CH_IN)
       call COLT_INTERSECT(CH_IN)
       ......

       do i=1,50
C         receive A
          call COLT_RCV(A, CH_IN)
          <consume A>
       end do
```

Figure 2: Example of the definition and use of a $COLT_{\texttt{HPF}}$ communication channel to transmit 50 times a 100×100 array of REALs between two HPF tasks.

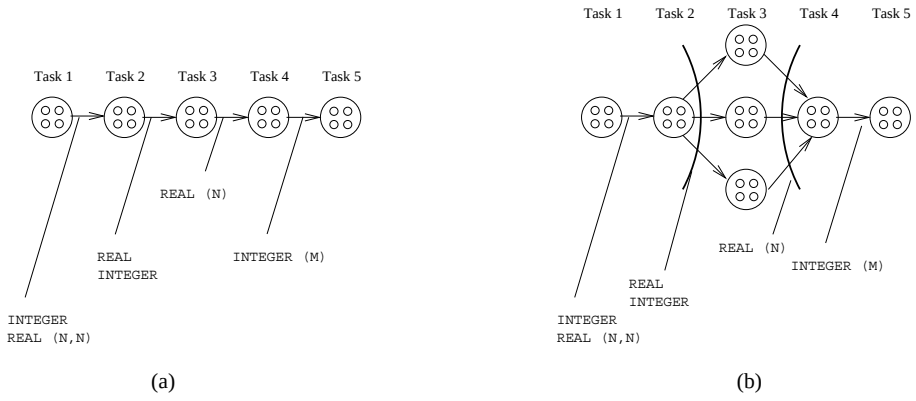


Figure 3: Two examples of the same application implemented (a) by means of a pipeline structure of data-parallel task, and (b) by hierarchically composing the same pipeline with a processor farm structure.

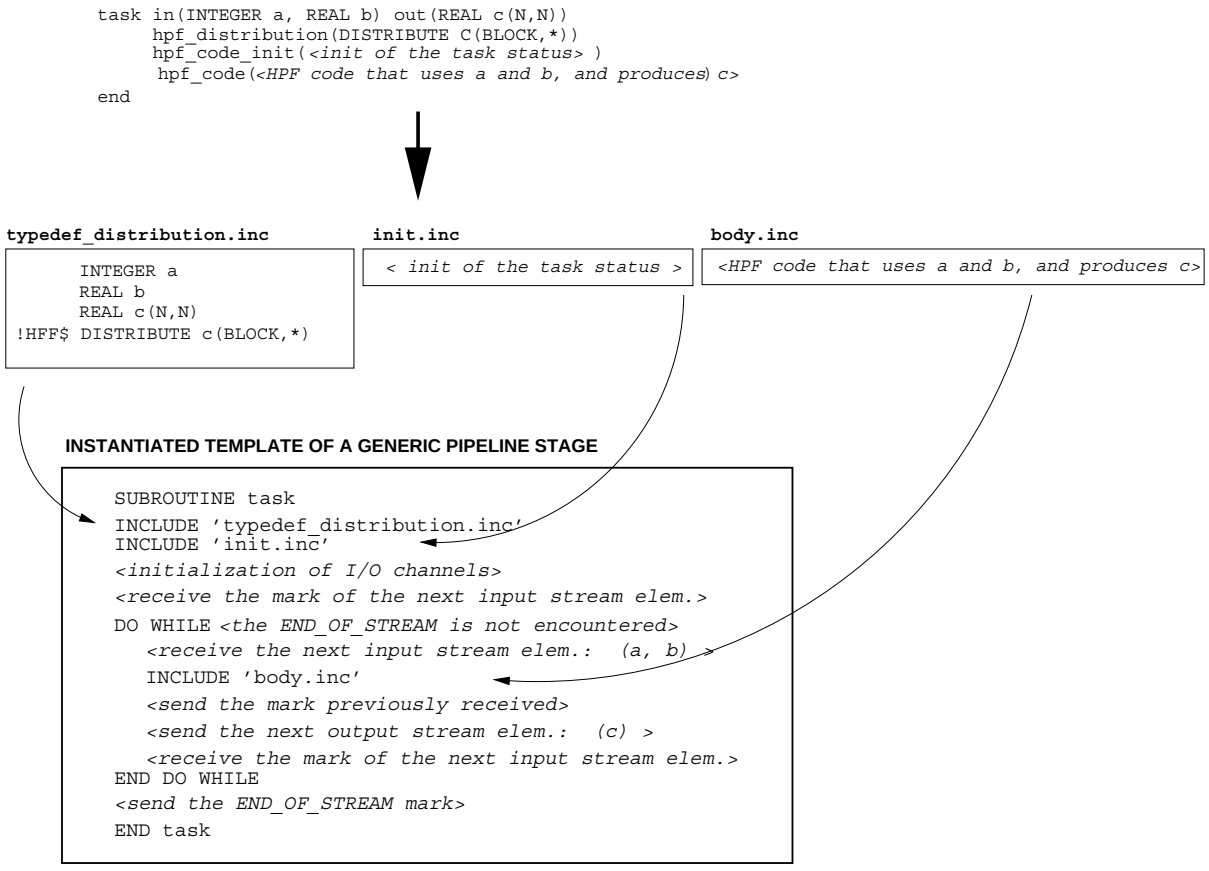


Figure 4: A template of a pipeline stage, where its instantiation is shown starting from a specific construct of a high-level coordination language.

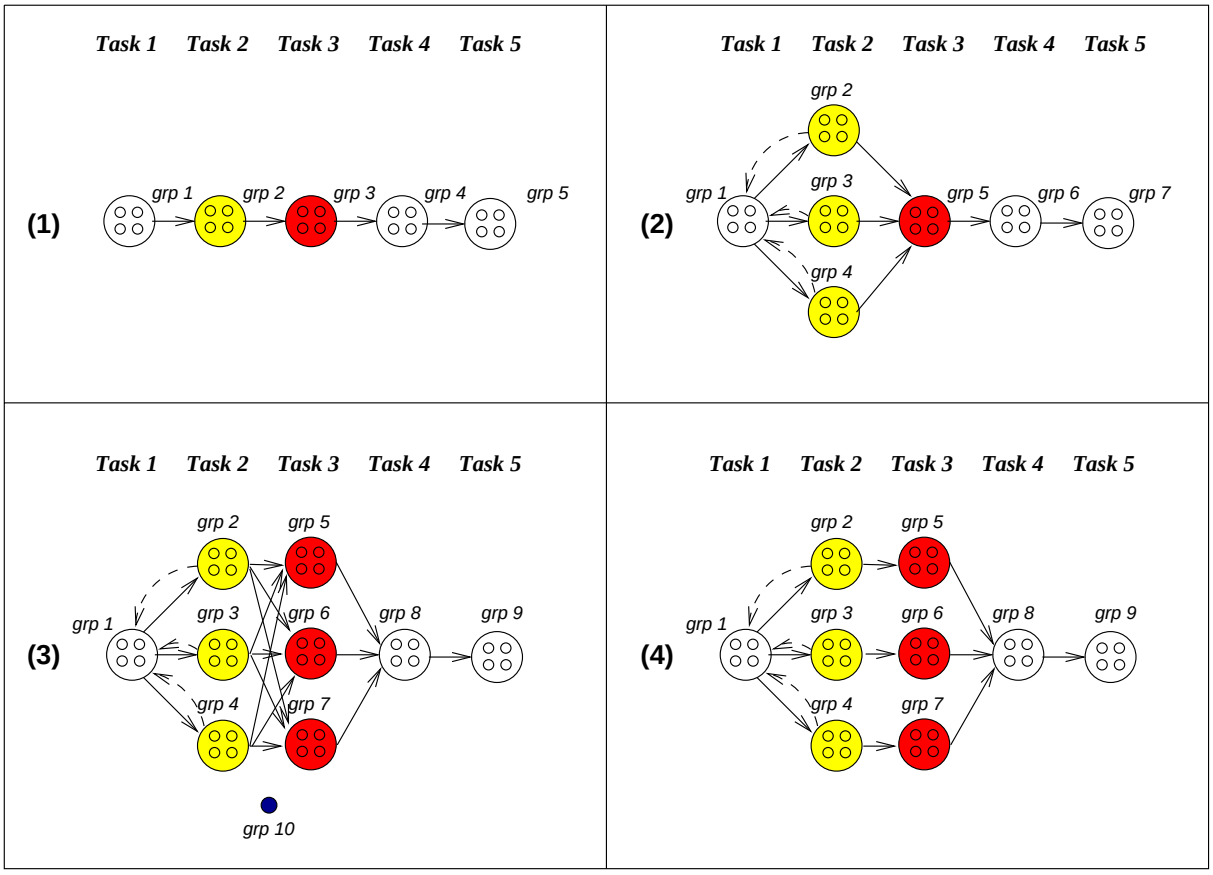
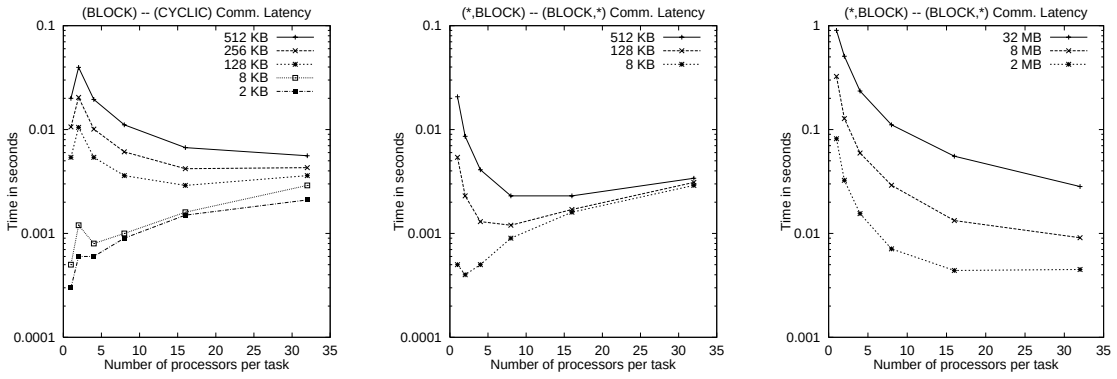
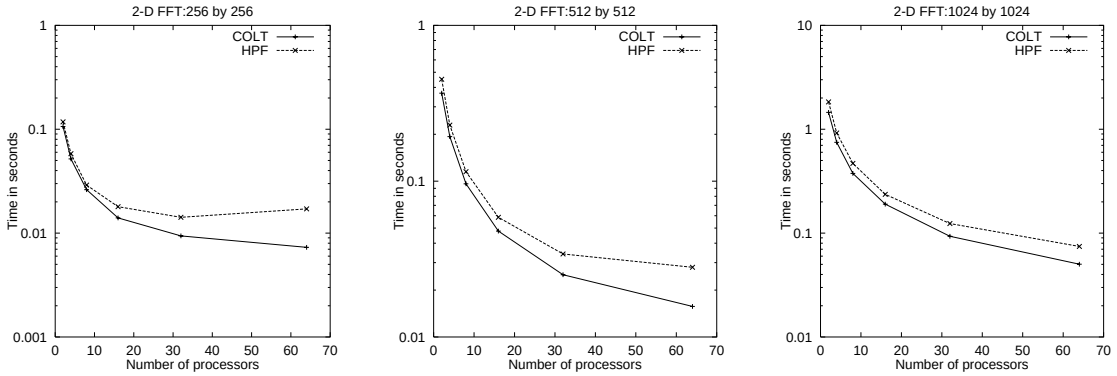


Figure 5: A pipeline composed of 5 stages (1), and the same pipeline where the bandwidth of the second stage has been increased by replicating the stage in three copies according to a farm structure (2), and where both the second and third stages are replicated (3),(4).



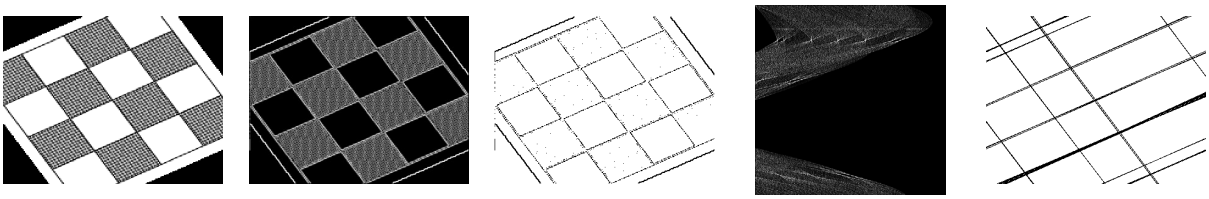
(a) (b) (c)

Figure 6: Average task-to-task communication latencies as a function of the size of the data exchanged and the number of per-task processors.



(a) (b) (c)

Figure 7: Execution time per input array for HPF and $COLT_{\texttt{HPF}}$ implementations of the 2-D FFT as a function of the number of processors.



(a) (b) (c) (d) (e)

Figure 8: Example of the input/output images produced by the various stages of the computer vision application: (a)⇒(b): Sobel filter stage for edge enhancement – (b)⇒(c): Thresholding stage to produce a bit map – (c)⇒(d): Hough transform stage to detect straight lines – (d)⇒(e): de-Hough transform stage to plot the most voted for straight lines.

Table I: Ratio of execution times of processor farm implementations exploiting Round Robin and Self Scheduling techniques.

	RR/SS ratio		
μ	**Exp. distr.**	**Unif. distr.**	**Balanced jobs**
0.2	1.13	1.12	0.999
0.4	1.13	1.12	0.997
0.8	1.14	1.11	0.999

Table II: Ratio of execution times obtained with the HPF and $COLT_{\texttt{HPF}}$ implementations of the 2-D FFT.

	HPF/$COLT_{\texttt{HPF}}$ ratio		
Procs	256 × 256	512 × 512	1024 × 1024
2	1.11	1.23	1.26
4	1.12	1.19	1.23
8	1.11	1.20	1.25
16	1.28	1.23	1.24
32	1.51	1.36	1.32
64	2.34	1.78	1.48

Table III: Computation and I/O times (in seconds) for the HPF implementation of the three stages of the computer vision application as a function of the number of processors used.

	Sobel&Thresh			**Hough**	**de-Hough**		
Procs	I/O	Comp	Total	Comp	I/O	Comp	Total
1	9.6	11.9	21.5	148.3	1.0	21.4	22.4
2	10.0	5.8	15.8	78.0	1.2	17.3	18.5
4	10.2	2.4	12.6	43.5	1.3	13.7	15.0
8	10.4	0.9	11.3	24.7	1.3	12.3	13.6
16	10.5	0.7	11.2	15.9	1.3	11.8	13.1
32	11.6	0.7	12.3	12.3	1.4	11.6	13.0

Table IV: Comparison of execution times (in seconds) obtained with the HPF and $COLT_{\texttt{HPF}}$ implementations of the computer vision application.

	$COLT_{\texttt{HPF}}$			**HPF**		**HPF/$COLT_{\texttt{HPF}}$ ratio**
Procs	*Structure*	Exec. Time	Speedup	Exec. Time	Speedup	
8	[1 (3,3) 1]	30.7	6.26	49.7	3.87	1.6
16	[2 (4,4,4) 2]	19.9	9.66	40.3	4.77	2.0
24	[4 (8,8) 4]	15.8	12.14	38.2	5.03	2.4
32	[8 (8,8) 8]	14.4	13.34	37.6	5.11	2.6