UNIVERSITÀ CA' FOSCARI DI VENEZIA
Dip. di Matematica Applicata ed Informatica

Technical Report Series on Computer Science

Technical Report CS-98-4

March 6, 1998

Salvatore Orlando and Raffaele Perego

$COLT_{\mathrm{HPF}}$, a Coordination Layer for HPF Tasks

Corso di Laurea in Informatica
Via Torino, 155 – 30173 – Venezia Mestre

$COLT_{\mathtt{HPF}}$, a Coordination Layer for HPF Tasks

Salvatore Orlando*, Raffaele Perego†

March 6, 1998

Abstract

This paper describes $COLT_{\mathtt{HPF}}$, a run–time support for the coordination of concurrent and communicating HPF tasks. $COLT_{\mathtt{HPF}}$ is conceived for use by a compiler of a high-level coordination language to structure a set of data-parallel HPF tasks according to popular forms of *task-parallelism*, but it can also be exploited to implement libraries of *skeletons*. Since it requires only small changes to the run-time support of the HPF compiler used, $COLT_{\mathtt{HPF}}$ is easily portable among different compilation systems. The paper outlines design and implementation issues, and discusses the main differences from other approaches proposed to exploit task parallelism in the HPF framework. We show how $COLT_{\mathtt{HPF}}$ 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.*

1 Introduction

Although HPF-1 permits programmers to express data-parallel computations in a portable, high-level way [15], it is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data-parallel paradigm [8]. From the point of view of the overall performance of these applications, rather than having a single data-parallel program, it might make more sense to subdivide the whole computation into several data-parallel pieces, where the various pieces run concurrently and cooperate thus exploiting *task parallelism*. In many cases, in fact, the performance of pure data-parallel programs does not increase proportionally to the number of processors used, and the exploitation of parallelism at a different level may significantly increase performance. Moreover, task parallelism is 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 [20, 21].

*S. Orlando is with the “Dipartimento di Matematica Applicata ed 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

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* [1] structure such as the one in Figure 1 for the third stage of the pipeline, where incoming jobs are dispatched on one of the three replicas 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 become mandatory in order to balance the workload assigned to the various replicas [2].

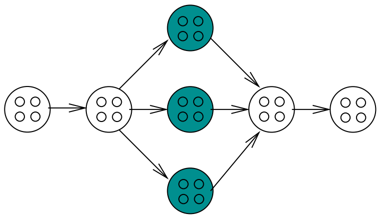


Figure 1: Structure of a pipeline composed of 5 data-parallel stages. Note the third stage replicated according to a farm structure.

In this paper we present $COLT_{\mathtt{HPF}}$ (COordination Layer for Tasks expressed in HPF), a coordination/communication layer for HPF tasks. $COLT_{\mathtt{HPF}}$ 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. We discuss in detail implementation issues, and show how $COLT_{\mathtt{HPF}}$ can be used to structure data-parallel computations that cooperate according to popular forms of task parallelism such as pipelines and processor farms [12, 16]. We present *templates* which implement these forms of task parallelism and we discuss the exploitation of these templates to design a structured, high-level, template-based coordination language that lets programmers easily organize 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* [4, 7, 2]. 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 the forms of parallelism and shows how a coordination layer like $COLT_{\mathtt{HPF}}$ can be exploited by a compiler of a simple, high-level, coordination language. Section 3 describes the functionalities and the design of the proposed layer, and its implementation on top of the Adaptor compilation system [3]. Section 4 presents the templates which implement pipeline and processor farm forms of task parallelism. In Section 5 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 6 surveys works dealing with the introduction of task parallelism in the HPF framework, and finally, Section 7 draws the conclusions.

2 Parallel structures to coordinate HPF tasks

To describe the features of $COLT_{\mathtt{HPF}}$we refer to a particular class of applications which exploit a mixture of task and data parallelism. More specifically, we focus on HPF applications $\mathcal{F}$ which transform a stream of data received in input into an output stream of the same length. $\mathcal{F}$, in turn, can be structured as a composition of simpler HPF tasks, namely $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$, which consume/produce streams of data and communicate these streams by means of *typed channels*. Data parallelism is exploited within the HPF tasks, on the other hand task parallelism is exploited by the concurrent execution of $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$.

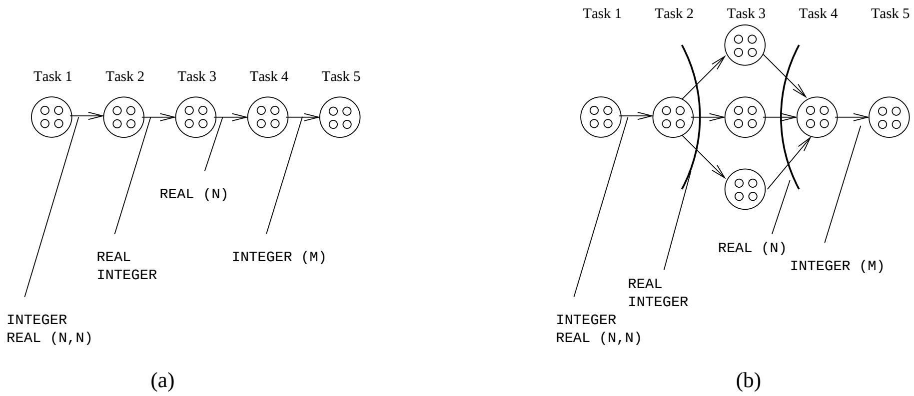


Figure 2: Two examples of the same application implemented (a) by means of a pipeline structure of data-parallel task, and (b) by hierarchically composing the same pipeline with a processor farm structure.

As regards the task parallelism structures used to coordinate these HPF tasks, we focus on a few forms of parallelism, namely *pipelines* and *processor farms*, which have been proved to be very common in many parallel applications [12, 16]. In particular, Figure 2.(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, the 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 2.(b) shows the same application where Task 3 has been replicated, 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, Task 2 and Task 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.

$COLT_{\texttt{HPF}}$ can be used to implement these forms of task parallelism, as well as less structured ones. It allows distinct HPF tasks to be loaded onto distinct processor groups, and to exchange data stream elements. Since a pair of HPF tasks may exchange arrays which are distributed on the various processors of the two groups, $COLT_{\texttt{HPF}}$ implements array communications through multiple point-to-point communications between the processors of the two groups. Moreover, since these communications, due to the presence of a data stream, have to be repeated several times, $COLT_{\texttt{HPF}}$ furnishes *persistent channels* which allow *Communication Schedules* and the relative packing/unpacking information to be reused for each communication. $COLT_{\texttt{HPF}}$ also provides primitives to signal events between HPF tasks. These signals are useful, for example, to implement processor farms that adopt a dynamic self-scheduling policy to dispatch jobs (see Section 4).

To implement each form of parallelism, we can devise a distinct *implementation template*. A template can be considered as the code scheme of a set of communicating HPF tasks which cooperate according to a fixed interaction pattern. Thus, in order to obtain an actual implementation of a user application, a template must be instantiated by inserting the user-provided HPF code, as well as all the correct calls to the $COLT_{\texttt{HPF}}$ layer.

Most of the code production needed to instantiate a template can be automated. The best usage of $COLT_{\texttt{HPF}}$ is thus through a simple high-level, template-based coordination language. Roughly speaking, the associated compiler should in this case be responsible for instantiating templates for users. To use such a coordination language a programmer should only be required to provide the HPF code of each task, its parameter lists to specify the types of the elements composing the input and the output streams, and finally the language constructs to express how these tasks must be coordinated, i.e. according to a pipeline or a processor farm form of parallelism.

A simple language to express this kind of structured parallel programming strategy, P^3L, has already been proposed elsewhere [2, 6]. 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 hierarchically composed. 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. No data-parallel construct is thus needed, since data-parallelism is naturally expressed at the task level by means of HPF. For example, a P^3L-like code to express Task 3 in Figure 2 would be:

```
task_3 in(INTEGER a, REAL b) out(REAL c(N,N))
       hpf_distribution(DISTRIBUTE C(BLOCK, *))
       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. The data layout for HPF distributed arrays is also specified at this level to permit the compiler to optimize intertask communications by exploiting the suitable primitives supplied by $COLT_{\texttt{HPF}}$.

In Figure 2.(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,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 2.(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,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 implementations. 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. The actual code production is carried out by choosing and instantiating the suitable templates associated with the various form of parallelism exploited.

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 [20, 21]. To exploit

*For the sake of brevity, the definition of the other tasks of the `pipe` is not reported.

these performance models for automatic code restructuring and optimization, some information about the costs of the specific application often has to be known [2]. More specifically, the optimization techniques should be based on:

- an analytic performance model of the templates 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 Implementation of $COLT_{\texttt{HPF}}$

In this section we describe in detail the implementation of our coordination/communication layer for HPF tasks. First of all we specify the binding of $COLT_{\texttt{HPF}}$ with an HPF compiler, Adaptor, which has been configured to exploit MPI as a message passing library. We believe that our technique is very general, so that a similar binding might easily be made with other HPF compilation systems that use MPI too.
Then we 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. As mentioned above, communicating distributed data between data-parallel tasks entails several point-to-point communications and, when the data and processor layouts of the sender and receiver tasks differ, it also requires the redistribution of the data exchanged. Figure 3 shows a simple example illustrating the point-to-point communications required to exchange a bidimensional array distributed differently on the sender and the receiver tasks. The data redistribution problem was solved adopting Ramaswamy and Banerjee's *pitfalls* redistribution algorithm [18]. Since run-time redistribution algorithms are quite expensive and, due to the presence of a stream of data, the same communications are repeated several times, $COLT_{\texttt{HPF}}$ computes redistribution information only once, and reuses it when necessary. Finally, we discuss some 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 were used to implement forms of parallelism where jobs are scheduled dynamically to reduce load imbalance problems.

Most $COLT_{\texttt{HPF}}$primitives are implemented as `HPF_LOCAL EXTRINSIC` subroutines[13]. This means that when one of our primitives is invoked by an HPF task, all the processors executing the task switch from the single thread of control provided by HPF to an SPMD style of execution. According to the language definition, `HPF_LOCAL` subroutines have to be written in a restricted HPF language where, for example, it is not possible to transparently access data stored on remote processors, but each processor can only access its own section of any distributed array.

3.1 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 permits one to specify communications to occur only within groups of specific processors. A predefined global communicator (`MPI_COMM_WORLD`) is available by default which comprises all the (virtual) processors of the MPI program.
Adaptor uses the predefined communicator to perform communications and also to ask for the ranks identifying each processor and the number of processors actually exploited. Similar run-time inquiries are present in the code produced by each HPF compiler in order to arrange the logical processor layout onto the physical processor grid and consequently distribute data and computations among the physical processors actually involved.

In order to permit Adaptor to support $COLT_{\texttt{HPF}}$, we had to slightly modify its run-time support. In particular our modified version of Adaptor produces an executable MPI code which:

1. reads during initialization a *configuration file* that defines the various HPF tasks. In particular, the configuration file specifies the number M of HPF tasks constituting the parallel application as well as the name of the associated HPF parallel subroutine, and for each task $t_i, i = 1, \cdots, M$, the number N_i of processors reserved for its execution. On the basis of this information, a simple mapping function is established which allows HPF task t_i to be associated with logical processors whose MPI ranks are in the range $[K_i, K_{i+1})$, where $K_0 = 0$ and $K_i = K_{i-1} + N_i, \forall i \geq 1$;

2. creates, M different communicators, one for each group of processors executing a single HPF task. Fortunately, MPI provides a powerful function, `MPI_Comm_split`, which creates all the communicators we need simultaneously, all of them with the same name. One of its parameters, *color*, is an integer that has to be the same for all the processors belonging to the same group. By calling `MPI_Comm_split` on each processor with a color equal to the task identifier t_i, each HPF task creates a new communicator, named `MPI_LOCAL_COMM_WORLD`, which includes all the processors of its own group;

3. uses `MPI_LOCAL_COMM_WORLD` as a communicator in all Adaptor-related MPI calls. In this way each HPF task still exploits a self-contained environment and any HPF communication or collective operation only affect the appropriate processors[†]. On the other hand, inter-task communications provided by our coordination layer will exploit the default global communicator `MPI_COMM_WORLD`.

The mechanism described above allows one to start multiple HPF tasks onto disjoint MPI groups, but it does not suffice for running distinct codes on each HPF task. To this end, since according to the static SPMD model provided by MPI the same program has to be loaded on each node, the main entry point of the task-parallel HPF program is a "loader", which invokes the right HPF code on each group/task. We implemented this general loader program as a simple sequential Fortran program. At the beginning the loader invokes the initialization routine of the Adaptor run-time support which performs the above steps to create the new communicators according to the group definition provided. At the end, it invokes the corresponding Adaptor finalization routine. Between these two special routines, the correct calls to execute distinct tasks onto the disjoint processor groups are inserted. More specifically, still on the basis of the group definition and instantiation described above, all the processors belonging to task t_i call the HPF routine associated with t_i in the configuration file. HPF tasks thus have to be written as subroutines without parameters, where all the data that are local to each task are defined within the subroutine itself.

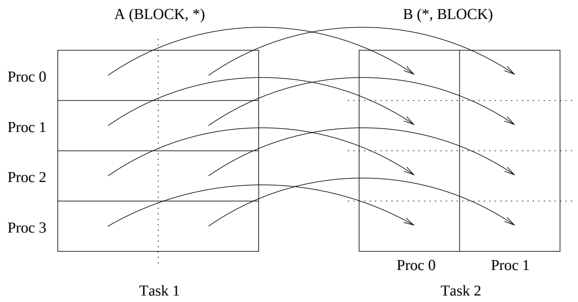


Figure 3: Point-to-point communications required to transfer a bidimensional array distributed per rows on four processors to a task with two processors and (*,BLOCK) distribution.

3.2 Channel initialization

As mentioned above, $COLT_{\texttt{HPF}}$ associates a *descriptor* with each typed channel used to transmit distributed arrays. Such descriptors are used during initialization to store information which is subsequently

[†]Note that, if the *configuration file* is not found, then above steps 1 and 2 are skipped and `MPI_LOCAL_COMM_WORLD` will be a copy of `MPI_COMM_WORLD`. Thus a pure data-parallel HPF program will be correctly compiled even with this modified version of Adaptor.

used to optimize communications. To fill the descriptors all the processors of both the sender and receiver groups, by invoking suitable primitives supplied by our coordination layer, have to carry out the following steps for each distributed array D_i transmitted over the channel:

1. inquire the HPF run-time support to find out the layout of D_i on the processor grid associated with their own processor group. To this end, HPF provides an appropriate intrinsic subroutine. Returned information is stored in the part of the channel descriptor associated with D_i 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 mapping information so that each processor involved knows the layout of D_i at both the sender and receiver sides[‡].

3. on the basis of the information exchanged at the previous step, compute the intersections between D_i distributions at the sender and receiver sides by means of the *pitfalls* algorithm [18].

4. build, on the basis of the result of the *pitfalls* algorithm, the *Communication Schedule* that is used each time a communication actually occurs to pack (unpack) the elements of D_i and send (receive) them to (from) each destination (source) processor. The *Communication Schedule* is stored in the part of the channel descriptor associated with D_i and clearly differs for each processor. For example, consider a processor p_k which has to send (receive) its local section of D_i 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 D_i owned by p'_j too.

3.3 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, the identifier of the sender task must be specified when the receive primitive is invoked. Considering that the primitives to transport distributed structures use information stored in the channel descriptors, to invoke these primitives we also have to supply the corresponding channel identifier.
When a 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 that 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 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 in FIFO order. 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. No channel setting is in fact 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.

[‡]Note that since each data-parallel task may have several arrays to be sent/received to/from other tasks, these communications may result in a deadlock if the invocation order of send/receive calls is not correct. A way to avoid the deadlock is to use an order in which all the (asynchronous) send calls precede all the receive ones on each processor.

3.4 Special messages and non-determinism

Messages often need to be received in a non-deterministic way. For example, an HPF task may need to receive data from a task non-deterministically chosen among several possibilities. The problem is that to ensure correctness the same non-deterministic choice must be globally made by all the processors executing 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.

To solve this problem, our layer 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 the right point-to-point receive primitives and actually receive the data from the selected source task as discussed above.

4 Template examples

In this section we present the templates which implement pipeline and processor farm forms of parallelism by exploiting the functionalities provided by $COLT_{\mathtt{HPF}}$. As discussed in Section 2, these templates can be exploited by a compiler of a structured, template-based, coordination language, or may be used to design skeletons (libraries) directly furnished to programmers.

4.1 Pipeline

A *pipeline* structure is a chain of data-flow stages where each stage is an HPF task which cyclically reads an element of the input stream and produces a corresponding output element. Pipelines may be composed with other forms of parallelism, and in this case they consume/produce streams from/to the external environment. However, since streams need to be generated and consumed, we consider a whole parallel application to be structured as a special pipeline, where the first stage is a task that only produces an output stream, while the last stage only consumes its input stream.

Moreover, if the length of the processed stream is unknown till run–time, a distributed termination protocol must be implemented to propagate termination information along the pipeline stages. To this end we associated 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 tasks. 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.

A process template and its instantiation We now discuss the template of a generic stage of a pipeline. In particular, we illustrate its instantiation starting from a high-level specification of the pipeline stage, thus assuming that the instantiation is carried out by the compiler associated with a template-based coordination language like the one outlined in Section 2.

Figure 4 shows the $P^3L - like$ specification of a generic pipeline stage. 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 $COLT_{\mathtt{HPF}}$ layer which initialize the input/output channels (see Section 3.2) and send/receive the elements of the input/output stream (see Section 3.3), 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.

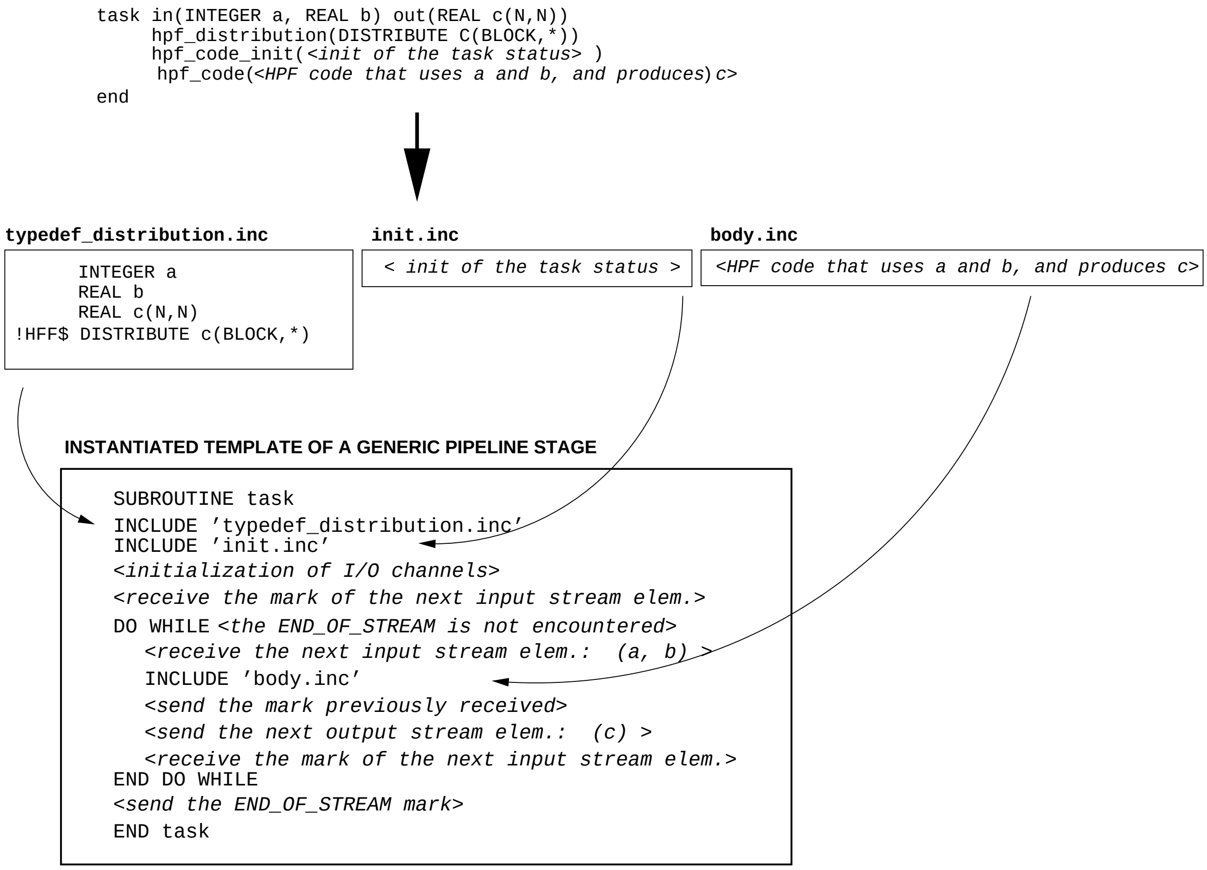


Figure 4: A template of a pipeline stage, where its instantiation is shown starting from a specific construct of a high-level coordination language.

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 only the following task in the chain, while the last one has to receive data from the previous task in the chain only. Moreover, the first task has to generate the incremental marks associated with stream elements and the `END_OF_STREAM` mark which propagates termination.

4.2 Processor farm

The *processor farm* form of parallelism ensures that data coming from an input stream are dispatched to a set of task replicas (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.

The insertion of a farm into a pipeline structure entails changing the templates of 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. Moreover, using the same terminology, we call the various instances of the replicated task *workers* of the farm.

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 5 data-parallel stages. Note the numbers close to each circle which represents a given task: they are the identifiers of the processor

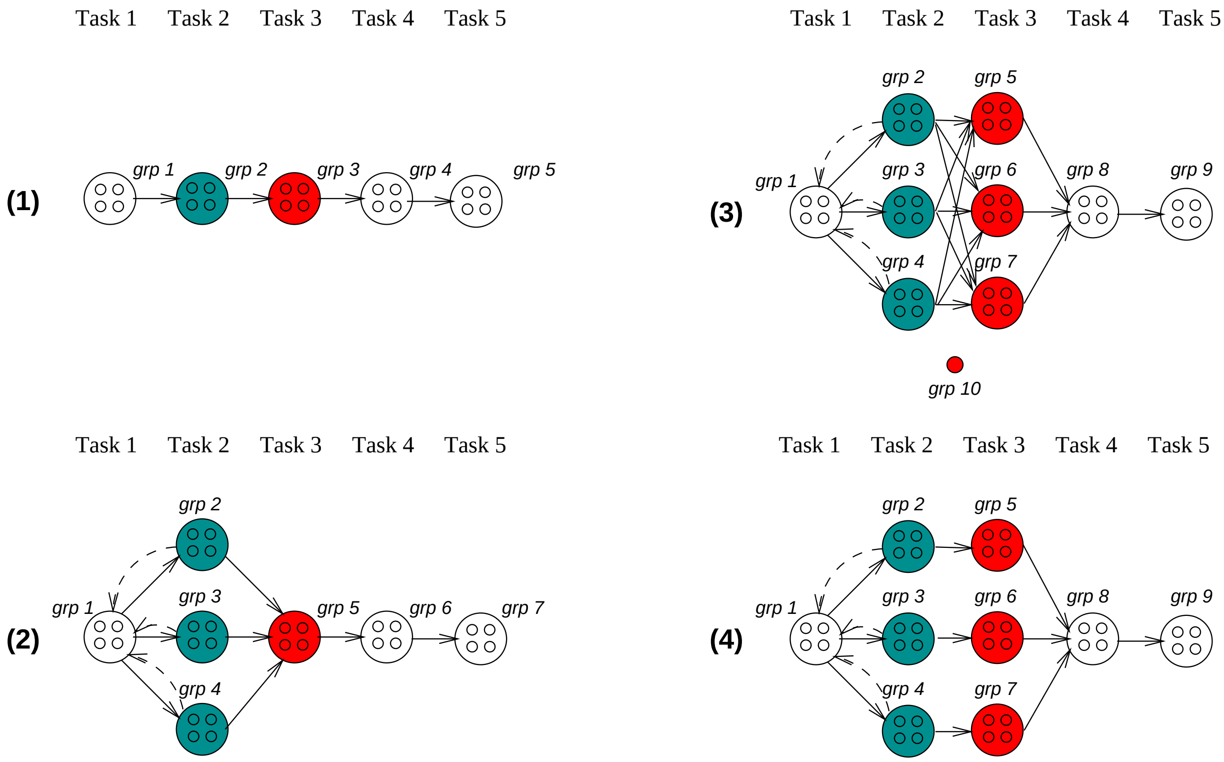


Figure 5: A pipeline composed of 5 stages (1), and the same pipeline where the bandwidth of the second stage has been increased by replicating the stage in three copies according to a farm structure (2), and where both the second and third stages are replicated (3),(4).

.

groups onto which the various data-parallel tasks have 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 the 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 2. 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.

In the following we discuss, without going into too much detail, the organization of the templates that implement the structures illustrated in Figure 5. Consider first the case of a farm inserted into 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 template 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 4.1. First of all, note that the emitter and collector have to establish a channel with all the workers (each one 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 3.4. 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, it can use deterministic primitives (see Section 3.3) 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 5, 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[§]. The signal messages have to be received non-deterministically by the emitter, which means that in this case too, we used the primitives illustrated in Section 3.4. At the beginning of 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. Then the emitter 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 Task 2 and Task 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 problem, we introduced a sequential scheduler shown in Figure 5.(3): note the presence of a small circle mapped on Group 10. This scheduler 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. The template to adopt for implementing 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 scheduler for the 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 sequential scheduler (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.

5 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

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

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.

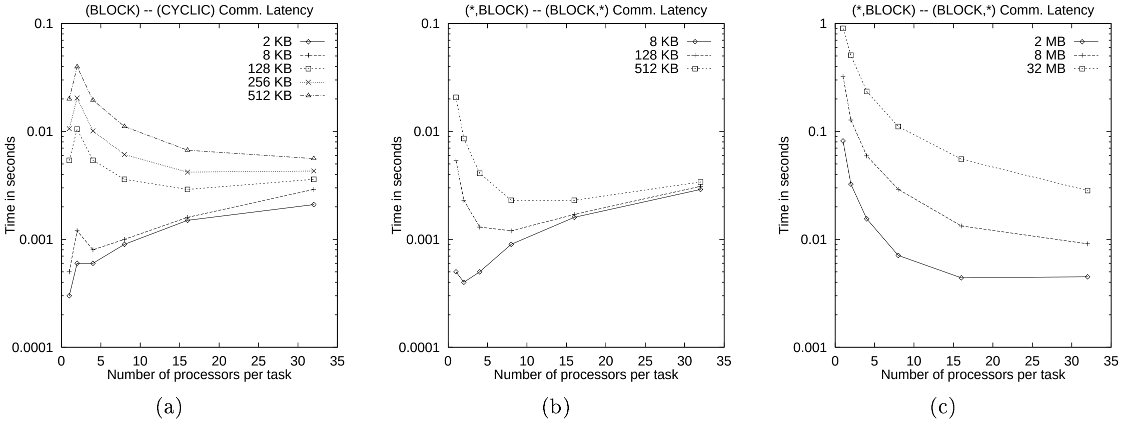


Figure 6: Average task-to-task communication latencies as a function of the size of the data exchanged and the number of per-task processors.

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

5.1 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 the processor farm implementation templates. 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 4.2.

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.

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.

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

	HPF/$COLT_{\text{HPF}}$ 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

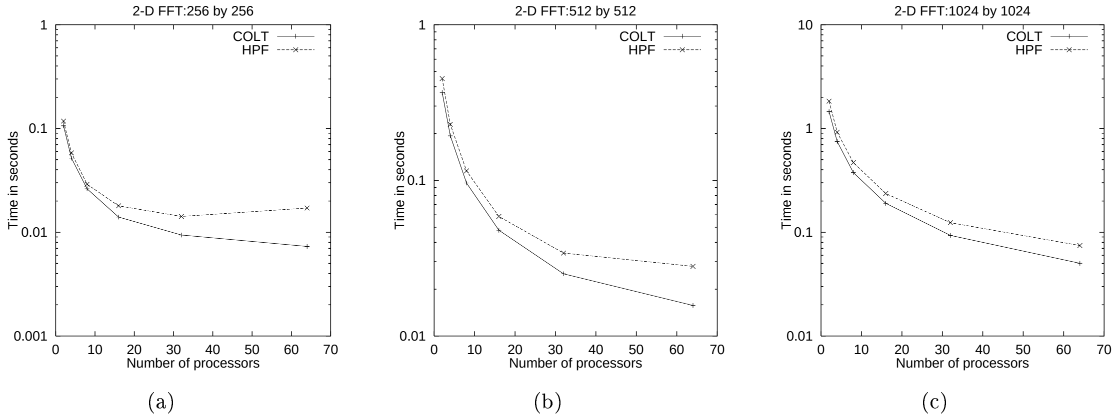


Figure 7: Execution time per input array for HPF and $COLT_{\texttt{HPF}}$ implementations of the 2-D FFT as a function of the number of processors.

5.2 Sample applications

Two sample applications were implemented by instantiating the templates described in Section 4. 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 [8, 9]. 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 $N1 \times N2$ array of complex values, a 2-D FFT entails performing $N2$ independent 1-D FFTs on the columns of the input array, followed by $N1$ independent 1-D FFTs on its rows. An HPF code implementing a 2-D FFT thus has the following structure:

```
        complex A(N1,N2)
!HPF$   distribute (*,BLOCK):: A
        ......
!HPF$   independent
        do icol=1,N2
           call fft_slice(a(:,icol))
        end do
        A = transpose(A)
!HPF$   independent
        do icol=1,N2
           call fft_slice(a(:,icol))
        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 $COLT_{\texttt{HPF}}$ templates, we can easily structure the 2-D FFT application as a two-stage pipeline in the following way:

```
C       STAGE 1
        complex A(N1,N2)
!HPF$   distribute (*,BLOCK):: A
        ......
```

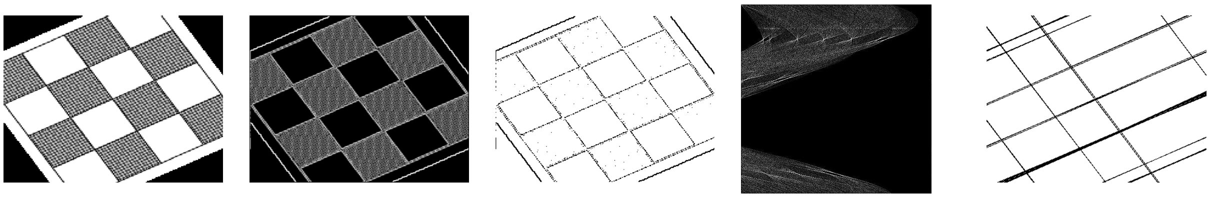
```
C       STAGE 2
        complex A(N1,N2)
!HPF$   distribute (BLOCK,*):: A
        ......
```

```
!HPF$   independent
        do icol=1,N2
           call fft_slice(a(:,icol))
        end do
        <send A to stage 2>
        ......
```

```
        <receive A from stage 1>
!HPF$   independent
        do irow=1,N1
           call fft_slice(a(irow,:))
        end do
        ......
```

Matrix transposition is not necessary in this case. In fact, due to the (BLOCK,*) distribution, the second stage can perform the $N1$ independent 1-D FFTs on the matrix rows without communications.

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 convenient 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) [11].



(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.

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. 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.

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 $COLT_{\mathtt{HPF}}$ implementations of the computer vision application.

	$COLT_{\mathtt{HPF}}$			**HPF**		**HPF/$COLT_{\mathtt{HPF}}$ 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) 8	14.4	13.34	37.6	5.11	2.6

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 [5], 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.

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 $COLT_{\mathtt{HPF}}$ 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 $COLT_{\mathtt{HPF}}$ implementations with those obtained by pure HPF implementations exploiting the same number of processors. The execution times measured with the $COLT_{\mathtt{HPF}}$ implementations were always better than the HPF ones. The performance improvements obtained are quite impressive and range from 60% to 160%.

6 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. Recently a new standard was proposed, HPF 2.0, which introduces task parallelism by allowing `TASK_REGION` blocks to be specified [14]. 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. According to this approach, in fact, any HPF source program can also produce a correct sequential executable if a Fortran 90 compiler is exploited. 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. On the other hand, 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 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. Finally, it is worth noting the low-level mapping of tasks. The programmers have to specify processor sub-grids on which tasks must be mapped, even if, as usual, the processor number can be queried at run-time, and processor sub-grids can be expressed in terms of this number.

Another similar approach has been presented in the literature [10, 19]. Due to the introduction of new directives, it also requires, like HPF 2.0, changes to the original HPF 1.0 language and the associated compiler. In this approach programmers have to specify an input/output list for all the tasks involved rather than explicitly providing assignments to express communications. Here tasks can be either `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 proper 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 HPF code is no longer a syntactically correct Fortran 90 program.

A different approach regards the design of an HPF binding for a standard message-passing library (MPI), proposed by Foster et al. [9]. They propose a framework in which concurrent HPF tasks communicate by means of a restricted subset of MPI communication primitives. We believe that the exploitation of a message-passing interface is too much of a "low-level" approach for introducing task parallelism into HPF. For example, even if primitives to optimize inter-task communication are supplied, these optimizations may be tedious and error-prone for the user. More importantly, the job of restructuring a program to take advantage, for example, of different architectures may become very complex. We will now discuss in some depth their approach, above all to point out usage complexities and semantics flaws of the proposed HPF-MPI binding. The proposed implementation, like ours, 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 multiple-threaded 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 Adaptor.

All four steps illustrated in Section 3.2 to set up communication channels are accomplished by their implementation as well, which, like $COLT_{\texttt{HPF}}$, uses Ramaswamy and Banerjee's algorithm [18] 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 performs the reception of 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 on the `MPI_Send`. In practice, this HPF binding of `MPI_Send` has a synchronous semantics 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 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" [17]. 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 deadlock, thus making it difficult for programmers to use them.

7 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 convenient because for many interesting applications, particularly in image and signal processing, the size of the data sets is limited by physical constraints which cannot be easily overcome [11].

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 [2], has already been implemented within the PQE2000 national project [22]. 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 SKIE applications, by hierarchically composing the provided primitive forms of parallelism, is also under development.

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

[2] 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.

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

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

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

[6] 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.

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

[8] 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.

[9] I. Foster, D. Kohr, R. Krishnaiyer, and A. Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. submitted to the Journal of Parallel and Distributed Computing, 1997.

[10] 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.

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

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

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

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

[15] 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.

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

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

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

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

[20] J. Subhlok and G. Vondran. Optimal Mapping of Sequences of Data Parallel Tasks. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1995.

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

[22] M. Vanneschi. Variable Grain Architectures for MPP Computation and Structured Parallel Programming. In *Programming Models for Massively Parallel Computers*. IEEE press, 1998. in printing.