Mixed Data and Task Parallelism with HPF and PVM *

Salvatore Orlando [a], Paolo Palmerini [b] and Raffaele Perego [b]

[a] *Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy*
[b] *Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy*

We present a framework to design efficient and portable HPF applications which exploit a mixture of task and data parallelism. According to the framework proposed, data parallelism is restricted within HPF modules, and task parallelism is achieved by the concurrent execution of several data–parallel modules cooperating through $COLT_{\texttt{HPF}}$, a coordination layer implemented on top of PVM. $COLT_{\texttt{HPF}}$ can be used independently of the HPF compilation system exploited, and it allows instances of cooperating HPF tasks to be created either statically or at run-time. We claim that $COLT_{\texttt{HPF}}$ can be exploited by means of a simple skeleton–based coordination language and associated compiler to easily express mixed data and task parallel applications runnable on either multicomputers or cluster of workstations. We used a physics application as a test case of our approach for mixing task and data parallelism, and we present the results of several experiments conducted on a cluster of Linux SMPs.

1. Introduction

Many applications belonging to important application fields exhibit a large amount of potential parallelism that can be exploited at both the *data* and the *task* level. The exploitation of *data parallelism* requires the same computations to be applied to different data, while *task parallelism* entails splitting the application into several parts, which are executed in parallel on different (groups of) processors. These parts then communicate and synchronize with each other by means of message passing or other mechanisms. Data parallelism is generally easier to exploit due to the simpler computational model involved. In addition, data–parallel languages are available which substantially help the programmer to develop data-parallel programs. High Performance Fortran (HPF) is the most notable example of such languages [14]. Unfortunately, data–parallelism alone does not allow many potentially parallel applications to be dealt with efficiently. Task parallelism is in fact often needed to reflect the natural structure of the application algorithm or to efficiently implement applications that exhibit only a limited amount of data parallelism. Many multidisciplinary applications (e.g. global climate modeling), as well as many applications belonging to various fields (e.g computer vision, real–time signal processing) do not allow efficient data–parallel solutions to be devised, but can be efficiently structured as ensembles of sequential and/or data-parallel tasks which cooperate according to a few common patterns [3,7,12]. The capability of integrating task and data parallelism into a single framework is the subject of many proposals since it allows the number of addressable applications to be considerably enlarged [11,22,10,8,6,15]. Also the latest version of the HPF standard, HPF 2.0, provides extensions which allows restricted forms of task parallelism to be exploited [16]. Unfortunately, the extensions are not suitable for expressing complex interactions among asynchronous tasks as required by multidisciplinary applications. Moreover, communication among tasks can occur only implicitly at the subroutine boundaries, and non-deterministic communication patterns cannot be expressed. A different approach regards the design of an HPF binding for a standard message-passing library such as MPI [10]. Low-level message passing primitives are provided which allow the exchange of distributed data among concurrent HPF tasks. The integration of task and data parallelism has also been proposed within an object-oriented coordination language such as Opus [8]. By using a syntax similar to that of HPF, Opus programmers can define classes of objects which encapsulate distributed data and data–parallel methods.

In this paper we present $COLT_{\texttt{HPF}}$ (COordination Layer for Tasks expressed in HPF), a portable coordination/communication layer for HPF tasks. $COLT_{\texttt{HPF}}$ is implemented on top of PVM and provides suitable mechanisms for starting, even at run–time, instances of data-parallel tasks on disjoint groups of (PVM virtual) 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.

There are many possible uses of the $COLT_{\texttt{HPF}}$ layer. Its interface can be used directly by programmers to write their applications as a flat collection of interacting data-parallel tasks. This low–level approach to HPF

* This work has been partially supported by HPPC/SEA contract number EU1063.

task parallelism has already been proposed by Foster et al. for the above mentioned HPF-MPI binding [10]. With respect to this proposal, $COLT_{\mathtt{HPF}}$ introduces new and important features, namely its complete portability among distinct compilation systems, and the run-time management of HPF tasks.

Indeed, in our view the best use of $COLT_{\mathtt{HPF}}$ is through a compiler of a high-level coordination language aimed at facilitating programmers in structuring their data-parallel tasks according to compositions of common forms of *task-parallelism* such as pipelines, processor farms, and master-slave paradigms [13,17]. Such a coordination language and the associated skeleton-based compiler have been proposed elsewhere [20]. The previous version of $COLT_{\mathtt{HPF}}$ used MPI as a message transport layer among HPF tasks. The exploitation of MPI, however, requires slight modifications to the HPF run–time, thus preventing the use of commercial HPF compilers. Here on the other hand, we discuss the use of PVM, whose features allowed us to overcome this limitation of the previous proposal, and also to support a dynamic task model where HPF tasks can be created/managed at run–time. As a consequence, the PVM version of $COLT_{\mathtt{HPF}}$ can be used to program heterogeneous networks of computers/workstations as well, where pure HPF approaches cannot be efficiently used due to the large and often unpredictable variations in the load and power of the computational resources available. HPF compilers in fact currently decide both data distribution and computation scheduling at compile-time, and perform static optimizations based upon the knowledge of the uniform communication and computation costs assumed for the target system [19,18]. Most of these static optimizations fail when processors have different capacities and when communication costs vary. The possibility of exploiting an additional level of parallelism with respect to HPF, should allow this limitation to be partially overcome if we treat a heterogeneous environment as a collection of several homogeneous sub-systems. A profitable "metacomputing" approach [24] can be thus adopted which tries to map the data-parallel parts of a complex application (i.e. the HPF tasks) onto the various homogeneous sub-systems on the basis of "affinity" evaluations. Inter-task communications implemented by means of $COLT_{\mathtt{HPF}}$ mechanisms are, on the other hand, much less sensitive to high and varying latencies, and can take place between these homogeneous clusters.

This paper aims to investigate this solution, by discussing the exploitation of $COLT_{\mathtt{HPF}}$ on a cluster of Symmetric Multi-Processor (SMP) Linux PCs. In particular, we used a physics application as a test case for our approach, while the testbed system was a cluster of Linux 2-way SMPs, interconnected by a 100BaseT switched Ethernet, where each SMP was equipped with two Pentium II - 233 MHz processors and 128 MB of main memory. The HPF compiler used was `pghpf` from the Portland Group Inc., version 3.0-4.

The paper is organized as follows. Section 2 discusses the coordination model behind our proposal for integrating task and data parallelism, and the main implementation issues. Section 3 introduces the design and functionalities of $COLT_{\mathtt{HPF}}$, and describes its implementation on top of the PVM communication layer. In Section 4 a sample application used to validate our approach is presented, and two different $COLT_{\mathtt{HPF}}$ mixed task and data parallel implementations are described. The results of the experiments conducted on our cluster of SMPs are also reported and discussed in depth. Finally, Section 5 draws some conclusions and outlines future work.

2. Integrating task and data parallelism

Experience in applicative fields, above all deriving from the development of *multidisciplinary applications*, seems to suggest that the best way to integrate task and data parallelism is to keep them on two distinct levels [3]:

- an outer *coordination* level for task parallelism. Task parallelism requires languages/programming environments that allow concurrent tasks to *coordinate* their execution and communicate with each other. Programmers are responsible for selecting the code executed by each task, and for expressing the interaction among the tasks. Most task parallel programming environments provide a separate address space for each task, and require programmers to explicitly insert communication statements in their code.
- an inner *computational* level for data parallelism. Data parallelism is considered "easy" to express, at least for "regular" problems. It is efficiently exploited on most parallel architectures by means of SPMD (Single Program Multiple Data) programs. However, the hand-coding of SPMD programs for distributed-memory machines is very tedious and error-prone. Fortunately, high-level languages such as HPF allow programmers to easily express data parallel programs in a single, global address space. The associated compiler, guided by user-provided directives, produces an SPMD program optimized for the target machine, and transparently generates the code for data distribution, communication, and synchronization.

In this paper we show how an existing HPF compiler, in our case a commercial compilation system by the Portland Group Inc. (PGI), can be used to build the inner "computational" level of the above model. To this end we devised $COLT_{\mathtt{HPF}}$, a run-time support that allows HPF tasks to be created either statically or dynamically, and provides primitives for the asynchronous message-passing of scalar and vector data. $COLT_{\mathtt{HPF}}$ is thus the basis to construct the "coordination" outer level required to integrate task and data parallelism.

Note that other coordination mechanisms, besides message-passing, could be adopted for task parallelism integration: for example, either Remote Procedure Calls (or Remote Method Invocations, if object-oriented wrappers were provided for HPF modules), or a virtual shared space, such as a Distributed Shared Memory abstraction or a Linda tuple-space [5]. We chose a simpler, but more efficient, message-passing abstraction, where array data are asynchronously sent over communication channels, but at the same time we recognized that message-passing is error-prone and too low level to express task parallelism in a similar framework. We are thus working to exploit $COLT_{\mathtt{HPF}}$ by means of a high-level *skeleton*-based coordination language [1,20]. The coordination of HPF tasks is expressed by means of specific language constructs, each corresponding to a very common *paradigm* for task parallelism, i.e. pipeline, processor farm, master-slave, acyclic task graph, etc [13,17]. Programmers choose the best skeleton for their purposes, and for each task specify the HPF *computational* code as well as the list of input/output data exchanged between the tasks. The skeleton corresponds to the instantiation of the specific language constructs (i.e. the specific parallel programming paradigm), and is in turn implemented by means of a set of *templates*, a sort of "wrappers" for the user-provided computational codes. The *templates* encapsulate all the control and coordination code needed to implement the associated parallel programming paradigm. By instantiating the *templates* provided for each task making up the skeleton, the compiler produces the HPF programs implementing the required mixed task and data parallel application.

$COLT_{\mathtt{HPF}}$ is currently binded to HPF only, and is specifically designed to consider the issues deriving from the communication of *distributed* array data. Further work consists in extending $COLT_{\mathtt{HPF}}$ to permit tasks written with different, also sequential languages such as Fortran 77, C/C++, and Java, to communicate each other. This feature will allow us to increase the number of addressable application fields, and, in particular, will make the resulting programming environment suitable for solving emerging *multidisciplinary applications*. A first step in this direction has been already made with the integration of $COLT_{\mathtt{HPF}}$ within the run-time of SkIE-CL, a skeleton-based multi-language coordination environment [2,26].

2.1. Implementation issues

The integration of task and HPF data parallelism presents several implementation issues, deriving from the need to coordinate parallel tasks rather than sequential processes. Since most HPF compilers generate an SPMD f90/f77 code containing calls to the HPF run-time system responsible for data distribution and communication, a $COLT_{\mathtt{HPF}}$ mixed task and data parallel program corresponds to an ensemble of distinct SPMD parallel programs (an MPMD program) coordinated by means of $COLT_{\mathtt{HPF}}$. By exploiting the *templates* corresponding to the skeletons specified by programmers, our *skeleton*-based compiler thus generates multiple HPF programs containing calls to the $COLT_{\mathtt{HPF}}$ run-time support. Such multiple HPF programs then have to be compiled with the HPF compiler and linked with the $COLT_{\mathtt{HPF}}$ and HPF libraries. Finally, note that both $COLT_{\mathtt{HPF}}$ and HPF run-time supports use some low-level message-passing middleware. The `pghpf` compiler used exploits RPM, the PGI Real Parallel Machine system. RPM supports process spawning and communication among processes mapped onto a group of homogeneous hosts. In our target architecture, RPM is implemented with UNIX sockets. RPM is very similar to PVM, although it is claimed that it offers greater efficiency and performance. On the other hand, $COLT_{\mathtt{HPF}}$ exploits PVM for inter-task communications, which is in turn implemented on top of UNIX sockets. The PVM and RPM libraries must thus also be linked to the MPMD object files in order to obtain the executables for the target machine.

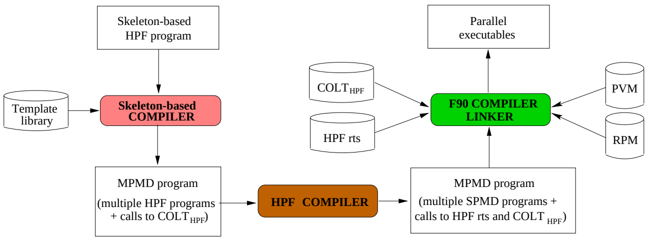


Figure 1. Structures and relationships of the $COLT_{\mathtt{HPF}}$ and HPF source-to-source compilers integrating task and data-parallelism.

The overall structure of the hierarchy of compilers which implement our model for task and data parallel integration is shown in Figure 1.

Inter-task communications. The primitives to exchange array data structures constitute one of the main difficulties in implementing $COLT_{\mathtt{HPF}}$. Consider, in fact, that arrays can be distributed differently on the groups of processors running the various HPF tasks. Moreover, since HPF compilers usually produce executable code in which the degree of parallelism can be established through suitable command-line parameters, the actual boundaries of array sections allocated on each processor executing the HPF task are unknown until run-time. $COLT_{\mathtt{HPF}}$ thus has to inspect at run-time the HPF support to find out on both the sender and receiver sides the actual mapping of any distributed array exchanged. All the processes involved in the communication then have to compute the intersections of their own array partitions with the ones of the processes belonging to

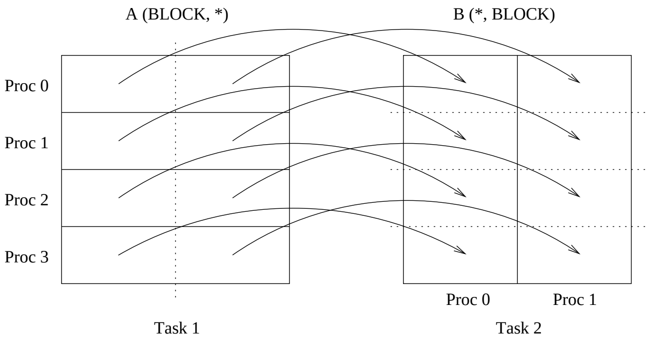


Figure 2. Point-to-point communications to send a bidimensional array from one HPF task, mapped on 4 processes, to another HPF task, mapped on 2 processes. Data distributions on the sender and the receiver tasks are `(BLOCK,*)` and `(*,BLOCK)`, respectively.

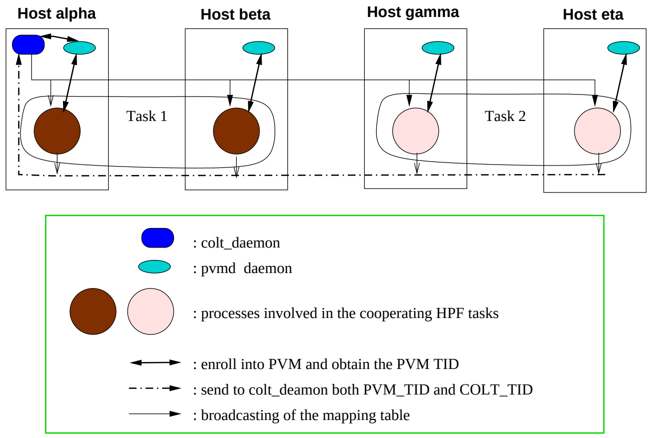


Figure 3. Interactions with *colt_daemon* needed to statically start two HPF tasks, each exploiting two processors.

the partner task. In this way, each sender process will know which portion of its array section must be sent to each process of the receiver task, while each process of the receiver task will know which array portions it has to receive from any of the sender processes. At the end of this preliminary phase, the actual comunication can take place. Note that communicating distributed data between data-parallel tasks thus entails making several point-to-point communications, which, in our case, are PVM communications. A simple example of the point-to-point communications required to communicate a distributed array is illustrated in Figure 2.

Low-level communication layer. Another important implementation choice regards the low-level communication middleware exploited for inter-task communications. As mentioned above, in this paper we discuss the adoption of PVM. On the other hand MPI was adopted in a previous release of $COLT_{\texttt{HPF}}$ [20]. MPI supports a static SPMD model, according to which the same program has to be loaded on all the virtual processors on which the mixed task and data parallel program has to be mapped. However, our MPMD model of execution was achieved by forcing disjoint groups of MPI processes to execute distinct HPF *subroutines* corresponding to the various HPF task involved. As a consequence, we had to modify the HPF run-time system in order to make each SPMD subprogram implementing an HPF task exploit a distinct communication context embracing only the corresponding group of MPI processes. Note that for the MPI-based version of $COLT_{\texttt{HPF}}$ we had to use a public-domain HPF compiler [4], for which the source code of the run-time support was available and modifiable. Moreover, the MPI static task model prevented the exploitation of “adaptive parallelism” approaches which result to be very effective on non-traditional parallel architectures such as clusters of workstations.

On the other hand, the PVM-based version of $COLT_{\texttt{HPF}}$ discussed in this paper supports a dynamic task model, according to which any HPF task (compiled as a separate executable) can be started on demand at run-time and participate in the $COLT_{\texttt{HPF}}$ application. This allows the set of running HPF tasks to be enlarged or restricted dynamically in order to adapt the running application to varying conditions of the computing environment used. Moreover, since we do not need to modify the HPF run-time system, a commercial compilation system like the PGI one can be employed.

3. $COLT_{\texttt{HPF}}$ implementation

In this section we describe the implementation of the $COLT_{\texttt{HPF}}$ layer on top of PVM. First of all we specify the mechanisms which allow multiple HPF tasks to be executed concurrently onto disjoint groups of PVM virtual processors. 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.

To obtain portability among different HPF compilation systems, $COLT_{\texttt{HPF}}$ exploits HPF standard features alone and does not rely on compiler specific features. Its HPF interface consists of a library of `HPF_LOCAL EXTRINSIC` subroutines [14]. 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 a true SPMD style of execution. In other words, through a `LOCAL` routine programmers can get the SPMD program generated by the HPF compiler under their control. Depending on the language definition, `HPF_LOCAL` subroutines have to be written in a restricted HPF language where, for example, data stored on remote processes cannot be accessed transparently, but each process can only access its own section of any distributed array.

3.1. Task loading

In order to exploit task parallelism on top of PVM, our approach entails distinct HPF tasks being run concurrently onto disjoint groups of virtual processors of the same PVM machine. The advantages of exploiting PVM as a communication layer are related to the dynamic process model supported by the PVM abstraction. According to this model, any process can enroll itself in the PVM machine at run-time. We exploited the dynamic features of PVM to support a dynamic HPF task model as well: $COLT_{\mathtt{HPF}}$ tasks, which are built as separate executable programs linked with the $COLT_{\mathtt{HPF}}$ and PVM libraries, enroll themselves in the $COLT_{\mathtt{HPF}}$ application at run-time, and can ask for other HPF tasks to be created dynamically.

The first main issue is thus the mechanism by which the various HPF tasks are loaded and start their execution. HPF executables exploit at this purpose compiler-dependent loading mechanisms, with which decisions about the degree of parallelism and processor mapping can be postponed until the launching time. For example, the `pghpf` compiler from the Portland Group Inc. produces an executable that accepts specific command-line parameters[1].

Hence, while HPF tasks are launched and mapped on the target system by exploiting their proprietary mechanisms, all the processes running each task (i.e., the processes partecipating in the corresponding SPMD subprogram generated by the HPF compiler) have to start executing by enrolling themselves into the PVM machine. To this end, the $COLT_{\mathtt{HPF}}$ initialization primitive, `colt_init()`, invokes the PVM primitive `pvmfmytid()`, through which all these processes not only enroll themselves into the PVM machine, but also receive their PVM identifiers (hereafter `pvm_tids`). Note that in order to exploit the PVM communication layer, the $COLT_{\mathtt{HPF}}$ support needs to know the `pvm_tids` of all the processes created and their relationship with the corresponding HPF tasks, each of which is identified within $COLT_{\mathtt{HPF}}$ by a distinct task identifier (hereafter `colt_tid`). It worth considering, in fact, that $COLT_{\mathtt{HPF}}$ communication primitives use `colt_tids` to name HPF tasks (e.g. the `colt_tid` of the receiver task in a send primitive), while the implementation of the primitives exploits the `pvm_tids` of the processes running the HPF tasks involved. The $COLT_{\mathtt{HPF}}$ support stores all the information about the correspondence between the various `colt_tids` identifying each HPF task, and the set of `pvm_tids` corresponding to the group of processes (PVM virtual processors) running the task, within a replicated data structure, called *mapping table*.

In order to gather information about the `colt_tids` of the other running tasks and the associated sets of `pvm_tids`, the $COLT_{\mathtt{HPF}}$ initialization primitive interacts with a specific $COLT_{\mathtt{HPF}}$ daemon, hereafter *colt_daemon*. We introduced the *colt_daemon* not only to centralize all the information about task registration, but also to manage *static* and, above all, *dynamic* task creation.

Static task creation is performed by the *colt_daemon* on the basis of a user-provided *configuration file*. This file stores, for each task to be started at launching time, **(a)** the name of the associated HPF *executable module*, **(b)** the degree of parallelism for the task, **(c)** the list of PVM hosts onto which the task has to be executed, and **(d)** the `colt_tid` to be assigned to it. As soon as the *colt_daemon* starts, it performs the following steps (see Figure 3):

1. enrolls itself into the PVM virtual machine, and obtains its own PVM TID, hereafter `daemon_pvm_tid`;
2. reads the configuration file (and performs error checking);
3. starts, by means of the mechanisms provided by the OS and the specific HPF compilation system used, the HPF tasks on the specified hosts. Furthermore, it supplies to each launched executable module (associated with an HPF task) the `daemon_pvm_tid` and its unique `colt_tid` as *command-line arguments*. As soon as the spawned processes start executing, they call subroutine `colt_init()`, to enroll themselves into the PVM machine and exchange information with *colt_daemon*.
4. receives from each spawned process the pair (`pvm_tid`, `colt_tid`). This allows *colt_daemon* to know the `pvm_tids` of all the started processes as well as their HPF task ownership.
5. stores the information received in its *mapping table*, and broadcasts the table to all the processes running the spawned HPF tasks.

Once all these steps have been completed, all the HPF tasks specified in the *configuration file* will run concurrently with the desired degree of parallelism. Moreover, each process running the $COLT_{\mathtt{HPF}}$ application is enrolled into the PVM virtual machine, and knows the identifier of the HPF task to which it belongs as well as the `pvm_tids` of all the processes involved in the various HPF tasks that have been statically started. Each process also knows the PVM tid of the *colt_daemon*. Further interactions with the *colt_daemon* are in fact needed to: **(1)** manage task termination by means of a call to subroutine `colt_exit()`[2], and **(2)**, ask the *colt_daemon* for the *dynamic* creation of new HPF tasks. Any HPF task, during its execution and through a specific $COLT_{\mathtt{HPF}}$

[1] We cannot use the PVM spawn primitive for creating sequential tasks, since we have to create data-parallel tasks.

[2] *colt_daemon* terminates after the termination of all the created HPF tasks.

primitive, `colt_spawn()`, can in fact ask *colt_daemon* for dynamically spawning another HPF task. The input parameters of this $COLT_{\texttt{HPF}}$ primitive are the *pathname* of the executable file, the *degree of parallelism* of the task, and the list of *PVM hosts*. The *task identifier* `colt_tid` is an output parameter of the primitive. It is univocally chosen by *colt_daemon* and returned to the requesting HPF task. In more detail, at the reception of a spawn request from a given HPF task identified by a `colt_tid` $\overline{T}$, *colt_daemon* performs the following steps:

1. chooses a unique `colt_tid` $\overline{T}_{new}$, and assigns it to the new task;
2. spawns the task as in the static case;
3. receives from all the processes running the newly created task their `pvm_tids`, updates accordingly its *mapping table*, and broadcasts its contents to all the processes executing task $\overline{T}_{new}$. In this way, the new HPF task is informed about all the HPF tasks participating in the $COLT_{\texttt{HPF}}$ application. A portion of this table, i.e. the row containing information about the new task $\overline{T}_{new}$, is also sent to the requesting task $\overline{T}$.

Other running tasks may clearly need to find out whether a given task is running or not. To this end, through a call to routine `colt_info()` they can ask *colt_daemon* for the status of the running application. In particular, the routine allows the existence of a given HPF task to be probed for. If the task is running, its `pvm_tids` are returned to the $COLT_{\texttt{HPF}}$ run-time system of the caller task. In this way also the HPF tasks which did not create the new task can become aware of its existence and cooperate with it.

Figure 4 shows the overheads incurred by $COLT_{\texttt{HPF}}$ on our testbed system for dynamically spawning an HPF task as a function of the degree of parallelism of both the spawner and the spawned tasks on our testbed system. The times reported also include the time needed to establish a $COLT_{\texttt{HPF}}$ communication channel between the spawner task and the spawned one (see Section 3). The plot on the left-hand reports results referred to experiments where the spawner and the spawned tasks are mapped onto different SMP machines, while the plot on the right-hand refers to experiments conducted on a single SMP machine. In both cases, the *colt_daemon* shares the workstation with the spawner task. Almost independently of the parallelism degree of the spawner task, the time needed for starting at run-time a two processor HPF task is about 0.30 `s` if the host onto which the task is spawned is different from the one running the spawner task, while it is about 0.22 `s` when the same SMP is used for running both the tasks. Only a small fraction of these overheads, however, are due to $COLT_{\texttt{HPF}}$. Most of this time is in fact spent by the operating system for actually launching the executable file (several processes have to be created), and for starting the relative HPF run-time support within each process.

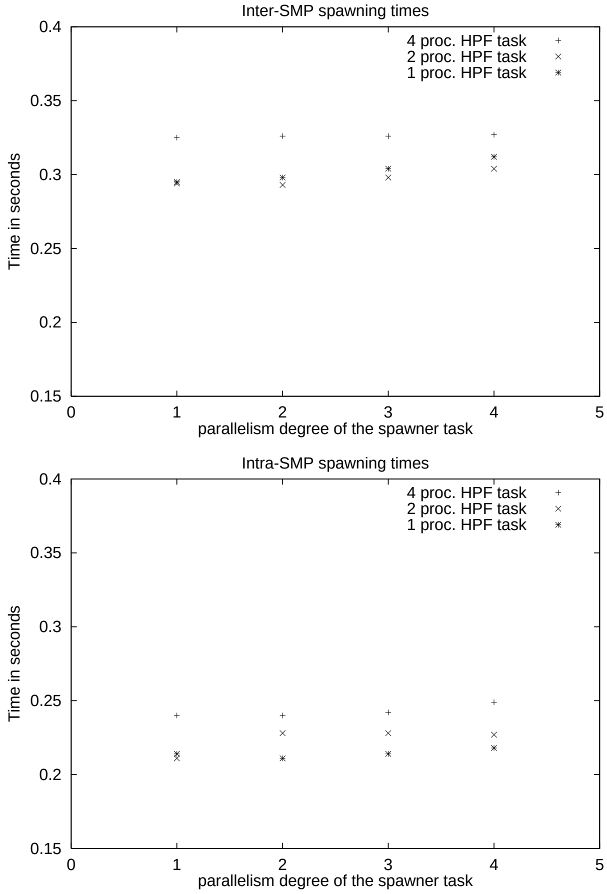


Figure 4. Times required to spawn at run–time a new HPF task.

3.2. Channel initialization and data transmission

$COLT_{\texttt{HPF}}$ supplies primitives to exchange both arrays and scalars between tasks. The exchange of scalar values is straightforward, even if the *non-deterministic* reception of such messages entails some implementation issues: in fact, to ensure correctness, we have to guarantee that the same non-deterministic choice is globally made by all the processes executing the receiving task [20]. These non-deterministic primitives are used to implement forms of parallelism where jobs are scheduled dynamically to reduce load imbalance problems.

On the other hand, communicating distributed data between data-parallel tasks is much more complex, since it 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 exchanged data to be redistributed. We solved the data redistribution problem by adopting Ramaswamy and Banerjee's *pitfalls* algorithm [21]. Since run-time redistribution algorithms are quite expensive and the same communications are usually repeated several times, $COLT_{\texttt{HPF}}$ allows *communication schedules* to be computed only once, and to be reused when possible.

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.

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 PVM send primitives.
When the 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.

Figure 5 shows the times required on our cluster of SMPs to exchange a distributed array between two data-parallel tasks as a function of the size of the array. The plot on the left-hand of the Figure regards ping-pong tests between two HPF tasks allocated onto two distinct SMP machines, while the other regards two HPF tasks mapped onto the same 2-way SMP. Each curve refers to a communication test between tasks with different degrees of parallelism, so that a curve with label `N-N` corresponds to a test regarding two tasks both mapped onto `N` logical HPF processors. The arrays exchanged are square matrixes of `REAL*4`, whose distribution on the two communicating partners is also specified in the labels associated with each curve. Note that if the array exchanged is distributed `(*, BLOCK)` on both tasks, and both tasks are mapped on a pair of processors, the communication is carried out by $COLT_{\texttt{HPF}}$ with only two PVM messages. On the other hand, if one array is distributed `(BLOCK, *)`, while the other is `(*, CYCLIC)`, each processor of the sender task has to communicate with all the processors of the destination task to accomplish the communication. Thus, if both tasks are mapped on two processors, four PVM messages are needed to transfer the array.

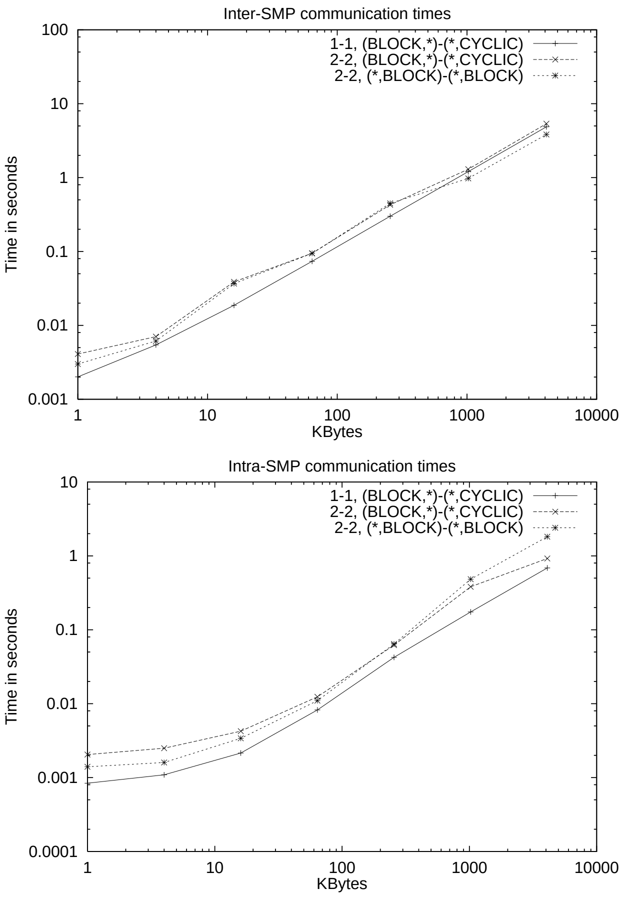


Figure 5. Inter-task communication times as a function of the dimension and distribution of arrays exchanged.

Since *inter-SMP* communications occur over a shared media[3], we cannot profit by the potential parallelism offered by $COLT_{\texttt{HPF}}$ in transmitting distributed arrays between data–parallel tasks, differently from other experiments we conducted on an SGI/Cray T3E [20]. The sustained communication bandwidth measured ranges between 5 and 8 Mb/s. Communication times increase almost linearly with the dimension of the data transferred due to message fragmentation and related overheads incurred by low-level protocols. Communication latencies are slightly smaller when the HPF array distribution is the same on both the sender and receiver sides, and thus the $COLT_{\texttt{HPF}}$ support transfers the array with less PVM messages. Of course, *intra-SMP* communication performances, reported in the plot on the right-hand of Figure 5, are almost an order of magnitude better than the inter-SMP counterparts. Even in this case, however, we have lower latencies when arrays have the same distributions and are thus transmitted with less PVM messages.

[3] Though we employed a switched Ethernet, each SMP was equipped with a single network interface card, so that multiple communications coming from the same SMP are serialized when crossing the network.

```
do NPAR
    call Init()
    do NEXT
        do DELTAT
            call Evolve()
            call Energy()
        enddo DELTAT
        do NINT
            call Evolve()
        enddo NINT
    enddo NEXT
    call Collect_Results()
enddo NPAR
call Average()
```

Figure 6. Pseudo-code of the simulation algorithm

.

4. A case study from physics

The test-case application chosen to show the benefits of our mixed task and data-parallel approach is the simulation of a Fermi-Pasta-Ulam (FPU) chain of N oscillators, coupled with unharmonic terms [9,23]. The characterization of the way such Hamiltonian systems reach their asymptotic equilibrium state has important implications in understanding the dynamic properties of systems with many degrees of freedom. The pseudo-code of the algorithm is shown in Figure 6. The algorithm basically consists of a main loop (on `NEXT`) which simulates the temporal evolution of the system initialized with energy not equally distributed among the normal modes. A symplectic algorithm of integration for the equation of motion is applied to simulate system evolution and is implemented within routine `Evolve`. As the system evolves, energy distribution is measured until equipartition is observed. Energy distribution is measured by routine `Energy` which implements the most computational expensive part of the simulation: it involves two one-dimensional Fourier transforms and a reduction operation. To speed up the simulation, energy is not measured at each time step, but the temporal loop (on `NEXT`) includes two nested loops. The former computes both evolution and energy distribution during a time-window of width `DELTAT`. The latter only simulates the evolution of the system during `NINT` time steps. The use of a `DELTAT`-wide time-window arises from the need of obtaining a measure of energy distribution averaged over more consecutive time-steps. Finally, since the evolution of the system depends on the sequence of random numbers used to generate initial conditions, the global simulation (i.e. the loop on `NEXT`) is repeated for `NPAR` times in order to average over some executions. The results of the `NPAR` simulations are thus collected by routine `Collect_Results`, while routine `Average` performs a statistical analysis.

Scientists are particularly interested in the behavior of such systems for low energies, for which relaxation to equilibrium is extremely slow. A number of time-steps of the order of 10^{10} have to be therefore simulated before reaching energy equipartition. The number N of oscillators simulated is also very important. In principle we would have to let $N \to \infty$. A scaling analysis can however show that $N = 1024$ or $N = 2048$ give a good enough approximation of such a limit.

The HPF implementation of the simulation code was not particularly difficult. The original Fortran 77 sequential code was slightly modified in order to remove the dependencies preventing the exploitation of independent loops, while data structures, above all one-dimensional arrays, were BLOCK distributed. Moreover, some subroutine calls were expanded to reduce overheads.

As already explained, most computation time is spent within routines `Evolve` and `Energy` which simulate the evolution of the system and measure energy distribution, respectively. Figure 7 shows the execution times of these HPF subroutines on our test-bed architecture as a function of the number of processors used. The execution times of routines `Init`, `Collect_Results` and `Average` are not plotted, since they are negligible with respect to the times of the subroutines above. Plots refer to the execution of a few time steps of the simulation of two FPU chains with 1024 and 2048 oscillators, respectively. To take into account possible changes in the system workload, we executed the same tests several times and only report the best results obtained.

As can be seen, the two time–consuming routines, `Energy` and `Evolve`, behave very differently when the number of processors is increased. Execution times of routine `Energy` scale very well and in some cases, due to the exploitation of the memory hierarchy, exhibit superlinear speedups with respect to the execution on a single processor. On the other hand, execution times of routine `Evolve` do not scale at all, since they increase as more processors are used for the execution. This behavior is due to stencil array references, whose implementation is very inefficient on our target platform. Further tests have shown that routine `Evolve` starts scaling only when the size of the problem (i.e. the number of oscillators simulated) is increased by at least one order of magnitude. Unfortunately, computing the energy equilibrium for such large systems is a computationally intractable problem, and would not increase significantly the numerical accuracy of the simulation.

The consequence of this behavior is that to choose the best number of processors to execute the simulation we have to trade off the need to exploit high parallelism to reduce execution time of routine `Energy`, with the inefficiencies introduced by the corresponding increase in the execution time of routine `Evolve`. Although the pure HPF implementation allows the total completion time to be reduced, the benefits of parallelism are not particularly satisfactory. As we will show

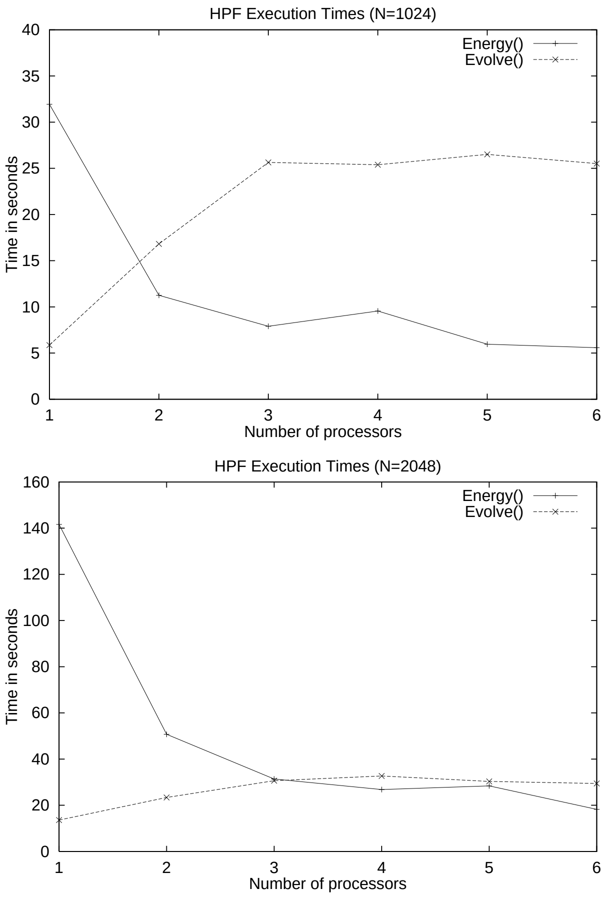


Figure 7. Execution times (in seconds) on a cluster of Linux 2-way SMPs for the HPF subroutines `Evolve` and `Energy` as a function of the number of processors. Plots refer to a few hundred simulation steps of FPU chains with 1024 and 2048 oscillators, respectively.

in the next section, the $COLT_{\mathtt{HPF}}$ approach allowed us to overcome the limits of the pure data–parallel implementation. The introduction of task parallelism, in fact, will allow the number of processors executing routines `Evolve` and `Energy` to be chosen independently, thus allowing the global performance of the whole application to be optimized.

4.1. The COLT$_{\mathtt{HPF}}$ *solutions*

Mixed task and data parallelism has been exploited by structuring the HPF code of the sample application according to two common forms of task parallelism: *pipeline* and *master-slave*. In order to obtain the actual implementations, the templates implementing the pipeline and master-slave skeletons provided by the high-level coordination language were instantiated with the application source code, as well as with the calls to the $COLT_{\mathtt{HPF}}$ primitives which initialize the communication channels and exchange data between the data-parallel tasks. A template can be considered as the "wrapper" for the computational code of an HPF task which cooperates with other tasks according to a fixed interaction pattern [20]. For our purposes we thus

Stage$_1$

```
do NPAR
   call Init()
   do NEXT
      do DELTAT
         call Evolve()
         send to STAGE 2
      enddo DELTAT
      do NINT
         call Evolve()
      enddo NINT
   enddo NEXT
enddo NPAR
```

Stage$_2$

```
do NPAR
   do NEXT
      do DELTAT
         recv from STAGE 1
         call Energy()
      enddo DELTAT
   enddo NEXT
   send to STAGE 3
enddo NPAR
```

Stage$_3$

```
do NPAR
   recv from STAGE 2
   call Collect_Results()
enddo NPAR
call Average()
```

Figure 8. $COLT_{\mathtt{HPF}}$ pseudo-code implementing the case study application as a three stage pipeline.

instantiated the templates which implement the first, middle and last stages of the pipeline skeleton, and the master and the slave task of the master-slave structure. The set of instantiated templates were then compiled and linked with the $COLT_{\mathtt{HPF}}$ and PVM libraries to obtain the application executables (see Figure 1). Below we describe the two different implementations and discuss in detail the results of the experiments conducted on our SMP cluster.

The pipeline implementation The simulation algorithm described in the previous section, can be structured by means of $COLT_{\mathtt{HPF}}$ as a pipeline of three stages, where each stage is implemented by a distinct HPF program. The HPF codes executed by the three stages are illustrated in Figure 8. The first stage (Stage$_1$) takes care of the initialization of the system (routine `Init`) and of its temporal evolution (routine `Evolve`). The second stage (Stage$_2$) computes energy equipartition (routine `Energy`) at given time instants on the basis of data coming from the previous stage. The third stage (Stage$_3$) is responsible for collecting the results (routine `Collect_Results`) and for performing the statistical analysis (routine `Average`). We made this subdivision after a careful analysis of the data-flow dependencies among the various subroutines called from within the external loop over `NPAR` in the original code (see Figure 6). Each call to routine `Energy` has to follow the corresponding call to routine `Evolve` performed within the inner loop over `DELTAT`. Moreover, the computation of `Energy` also depends on the completion of all the calls to `Evolve` carried out during previous iterations of the loop over `NEXT`. On the other hand, `Evolve` does not depend on any data computed by `Energy`. Stage_1 thus sends a copy of the system state to Stage_2 at each iteration of the inner loop over `DELTAT`, and Stage_2 receives the data and evaluates the energy equilibrium on the received data. On the other hand, Stage_2 sends its results to Stage_3 only `NPAR` times exactly at the end of the two nested loops identified by `NEXT` and `DELTAT`. The dimension of the elements of the *data stream* flowing in the pipeline is quite small. The system status exchanged between Stage_1 and Stage_2 consists of two arrays of N `REAL*8` values, where N is the number of oscillators simulated. On the other hand, Stage_2 transmits to Stage_3 only an array of `NEXT` `REAL*8` values at each iteration of the loop over `NPAR`.

The overall throughput of a pipeline can be improved by increasing the bandwidth of each pipeline stage, i.e. by reducing the time required to process each stream element, and at the same time by maintaining the bandwidths of all the stages balanced. The bandwidth of each data-parallel stage can be increased in two ways: either by adding processors for its data-parallel execution, or by replicating the stage into several copies [25]. Replication, however, can only be exploited if the computation performed by the stage does not depend on its internal "state". Moreover, replication may entail modifying the ordering of the stream elements sent from the copies of the replicated stage to the next pipeline stage. This may happen because distinct elements of the stream may have different execution times, or because the capacities of the processors executing the replicas of the stage may be different or may change due to variations in workstation loads.

As regards the templates adopted to implement a pipeline where a stage $\mathcal{S}_i$ is replicated, the preceding stage $\mathcal{S}_{i-1}$ dynamically dispatches its outcoming stream elements to the copies of $\mathcal{S}_i$ to balance the workload, while $\mathcal{S}_{i+1}$ receives the results from all $\mathcal{S}_i$'s non-deterministically. In particular we adopted a *self-scheduling* policy, according to which each copy of $\mathcal{S}_i$ signals its availability to receive further work (i.e., a new stream element) to $\mathcal{S}_{i-1}$, while $\mathcal{S}_{i-1}$ non-deterministically receives these signals from all the copies of $\mathcal{S}_i$. Moreover, $\mathcal{S}_{i-1}$ exploits a *prefetching* strategy to prevent replicas from waiting idly for a new stream item.

Returning to our test-case application, experiments showed that the middle stage, Stage_2, is the slowest one. Fortunately, Stage_2 can be replicated since it is stateless, and the correctness of the computation performed by Stage_3 is preserved independently of the reception order of data from Stage_2. Furthermore, Figure 7 clearly shows that the efficiency of routine `Energy` is at its maximum when it is executed on only two processors, that is within a single 2-way SMP. In particular, the speedup observed in this case is superlinear due to a better exploitation of the memory hierarchies.

From the observations above we deduce that if m 2-way SMPs are available for executing Stage_2, it is more profitable to replicate the stage into m copies, each running on a single SMP, rather than exploiting all the $2 \cdot m$ processors to execute a single instance of Stage_2. Moreover, since we noted that when a single instance of Stage_2 runs on a 2-way dedicated SMP, the computational bandwidth of the SMP is not completely saturated, two replicas of the second stage can be profitably mapped onto each SMP to optimize the overall throughput, i.e. the number of `Energy` computations completed within the unit of time.

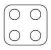
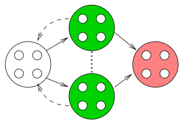
Table 1 compares the execution times obtained with the $COLT_{\mathtt{HPF}}$ pipeline and the pure HPF implementations of our case study application. The results reported refer to a few hundred iterations of the simulation run by fixing the problem size to 1024 and 2048 oscillators, respectively. All the experiments were conducted when no other user was logged in the machines. As regards the $COLT_{\mathtt{HPF}}$ implementation, the tests were performed by varying both the number of processors used, and the "organization" of the three-stage pipeline implementing the application (i.e. the degree of parallelism of each stage and number of replicas of the middle stage). The columns labeled *Structure* in the table indicate the mappings used for the corresponding test. For example, in the rows that refer to the tests conducted with six processors, $[1\ (2,2)\ 1]$ means that one processor was used for both the first and last stages of the pipeline, while each of the two replicas of the middle stage were run on two distinct processors. On the other hand, in the rows that refer to the experiments conducted on three processors, $[\frac{1}{2}\ (2)\ \frac{1}{2}]$ means that the corresponding test was executed by mapping both Stage_1 and Stage_3 on the same processor, and Stage_2 on two other processors. Finally, in the rows referring to experiments conducted on six processors, $[1\ (\frac{2}{2}, \frac{2}{2}, \frac{2}{2}, \frac{2}{2})\ 1]$ means that the corresponding test was executed by exploiting four replicas of Stage_2, each pair of replicas sharing the same 2-way SMP, so that each istance of Stage_2 had 2 half-processors (i.e., $\frac{2}{2}$ processors) available.

The degree of parallelism exploited for Stage_1 and Stage_3 was fixed to one for all the tests conducted, because of the increase in execution times of routine `Evolve` when run on several processors (see Figure 7) and the slight computational cost of the third stage of the pipeline.

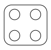
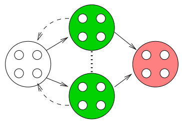
The execution times measured with the $COLT_{\mathtt{HPF}}$ implementations were always better than the HPF ones. The performance improvements obtained are quite im-

Table 1
Completion times (in seconds) obtained by the $COLT_{\mathtt{HPF}}$ pipeline and the pure HPF implementations of the case study application.

Problem Size=1024

	HPF 	$COLT_{\mathtt{HPF}}$ **(pipeline)** 		
Procs	*sec.*	*Structure*	*sec.*	*Ratio*
1	34.6	–	–	–
2	38.2	–	–	–
3	36.4	$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	15.6	2.33
4	41.7	[1 (2) 1]	14.7	2.84
5	35.3	$[\frac{1}{2}\ (2,2)\ \frac{1}{2}]$	7.7	4.57
		$[\frac{1}{2}\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ \frac{1}{2}]$	7.6	4.64
6	34.0	[1 (4) 1]	9.9	3.43
		[1 (2,2) 1]	7.6	4.47
		$[1\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ 1]$	7.5	4.53

Problem Size=2048

	HPF 	$COLT_{\mathtt{HPF}}$ **(pipeline)** 		
Procs	*sec.*	*Structure*	*sec.*	*Ratio*
1	148.0	–	–	–
2	87.2	–	–	–
3	70.4	$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	56.0	1.26
4	69.3	[1 (2) 1]	55.8	1.25
5	55.2	$[\frac{1}{2}\ (2,2)\ \frac{1}{2}]$	31.0	1.78
		$[\frac{1}{2}\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ \frac{1}{2}]$	27.8	1.98
6	52.1	[1 (4) 1]	32.6	1.59
		[1 (2,2) 1]	30.2	1.72
		$[1\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ 1]$	26.8	1.96

pressive and range from 25% to 353%. If we consider the results in terms of absolute speedup over the execution on a single processor[4], the $COLT_{\mathtt{HPF}}$ implementation with the problem size fixed to 1024, obtained speedups of 2.2 and 4.6 on three and six processors, respectively. On the tests conducted with 2048 oscillators, the speedups were 2.6 on three processors and 5.5 on six. As expected, due to the behavior of routine `Energy` discussed above, it turned out to be more profitable to execute Stage_2 on two processors and to replicate the stage, rather than using all the available processors for its data-parallel implementation. There was also a slight performance increase from running two replicas of Stage_2 on each 2-way SMP in order to saturate SMP computational bandwidth.

The master-slave implementation In the *master-slave* implementation of our test-case application, the master takes care of the initialization of the system (routine `Init`) and of its temporal evolution (routine `Evolve`). Moreover, it is responsible for collecting the results (routine `Collect_Results`) and for performing the statistical analysis (routine `Average`). In other words, the master performs the job that was done by Stage_1 and Stage_3 in the pipeline implementation discussed above. Each slave computes energy equipartition (routine `Energy`) on the data coming from the master, and returns the results to the master itself. Similarly to the pipeline implementation, where the first stage dynamically dispatches jobs to the replicas of the second stage, in this case too the master dynamically dispatches jobs to the slave tasks according to the same self-scheduling policy.

In the pipeline implementation, all HPF tasks were created statically. Also the degree of parallellism for the data-parallel stages, and the number of replicas for the second stage were chosen before starting the execution. The only dynamic feature exploited was the scheduling policy, which may be very effective when the execution time of the replicated stages is non-uniform. Unfortunately, it does not eliminate all the load balance problems which may arise due to the SMP time-sharing environment which may introduce unpredictable variations in the load of the machines used.

The $COLT_{\mathtt{HPF}}$ templates implementing the master-slave skeleton use another dynamic strategy in order to improve the handling of unpredictable variations in the processor capacities occurring at run–time. In fact. only the master and one slave are started at the beginning. Further slaves are created dynamically by the master if its bandwidth is higher than the aggregate bandwidth of all the slaves currently running. To this end, the master profiles the execution and measures the aggregate bandwidth of the slaves $\mathcal{B}_{slaves}$ as well as its own bandwidth $\mathcal{B}_{master}$. While disequation $\mathcal{B}_{slaves} > \mathcal{B}_{master} + T$, with T a fixed threshold, is satisfied and other processors are available, the master spawns other slave tasks. The configuration of the master-slave application is thus tuned dynamically, by chosing the best number of slave tasks on the basis of an accurate run-time estimate which considers not only the algorithmic features of the application, but also the actual capacities of the machines involved.

Table 2 compares the execution times obtained with 5 physical processors with the $COLT_{\mathtt{HPF}}$ master-slave and the pure HPF implementations of our case study

[4] The sequential executable was obtained by compiling the HPF program with the PGI `f90` compiler.

Table 2
Completion times (in seconds) obtained by the $COLT_{\mathtt{HPF}}$ master-slave and the HPF implementations of the case study application.

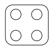
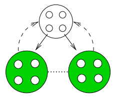
Problem Size=2048				
	HPF 	***COLT*$_{\mathtt{HPF}}$ (master-slave)** 		
Procs	sec.	*Structure*	sec.	*Ratio*
5	55.2	[1 (2,2)]	31.4	1.75
		[1 ($\frac{2}{2}$,$\frac{2}{2}$,$\frac{2}{2}$,$\frac{2}{2}$)]	26.4	2.09

Table 3
Dynamic vs. Static master-slave (**M-S**) by varying the load on the master machine. Master allocated on a dedicated 2-way SMP, and up to 4 slaves mapped on other two dedicated 2-way SMPs.

Problem Size=2048				
	***Static* M-S**		***Dynamic* M-S**	
Load	*no. of slaves*	*sec.*	*no. of slaves*	*sec.*
0	4	26.4	4	27.1
2	4	31.1	3	30.1
3	4	35.6	2	34.5
4	4	46.2	2	43.2

application. The results reported refer to experiments conducted by fixing the problem size to 2048. As can be seen, the completion times measured are very similar to those obtained with the pipeline version of the application (see Table 1). This means that the master bandwidth is high enough to dispatch the work to two slave tasks and to perform all the computations previously done by the first and last stage of the pipeline.

To evaluate the effectiveness of the strategy exploited to dynamically tune the number of slave tasks as a function of the bandwidth of the master and slaves, we introduced varying artificial workloads onto the SMP running the master task. In particular, up to four cpu-bound processes were executed concurrently with the master task. Table 3 reports the results obtained and compares them with those obtained by disabling dynamic task creation and always exploiting four slave tasks (Static master-slave). The Table highlights that in the absence of external workloads the static implementation is slightly more efficient. However, as the load on the SMP running the master task increases, the dynamic master-slave implementation adapts itself by starting less slave tasks and outperforms the static one. For example, when four cpu-bound processes share the same 2-way SMP with the master task, the $COLT_{\mathtt{HPF}}$ dynamic master-slave implementation becomes aware of the master's lower bandwidth and only starts two slave tasks. On the other hand, in the static case the number of slave tasks is fixed to four (the optimum configuration on an unloaded system) and the master pays additional overheads for initializing the $COLT_{\mathtt{HPF}}$ communication channels and for scheduling and managing the slave tasks whose aggregate bandwidth is significantly higher than the master one. Moreover, the dynamic strategy implemented ensures that the minimum number of slave tasks needed to maximize the overall throughput is exploited, thus minimizing resource usage.

We also conducted tests to evaluate the effects of the presence of external loads on the SMPs running the slave tasks. We observed that even in this case, at least as regards the performance of our test-case application, it is always profitable to start up two slaves on each SMP. Of course, the presence of other cpu-bound processes on these machines reduces the SMP's capacities accordingly, and thus causes an increase in the completion time of our application. Thus, we did not gain any advantages from the dynamic creations of slave tasks, provided that in both implementations, i.e. in the static and the dynamic one, the same machines and the same maximum number of slaves were considered. The only advantage of the dynamic master-slave skeleton is its ability to select the most unloaded machine on which to start new slave tasks.

5. Conclusions

We have presented $COLT_{\mathtt{HPF}}$, a coordination layer for HPF tasks, which exploits PVM to implement optimized inter-task communications. $COLT_{\mathtt{HPF}}$ allows a mixed task and data-parallel approach to be adopted, where data-parallel tasks can be started either statically (at loading-time) or dynamically (on demand). It worth remarking that, differently from other related proposals [8,11,6,22], our framework for integrating task and data parallelism is not based on the adoption of an experimental HPF compilation system. An optimized commercial product such as the PGI `pghpf` compiler, now available for different platforms and also for clusters of Linux PCs, can be instead employed.

An important features of the coordination model proposed in this paper is the adoption of a high-level *skeleton*-based coordination language, whose associated compiler generates HPF tasks with calls to the $COLT_{\mathtt{HPF}}$ support on the basis of pre-packaged templates associated with the task-parallel skeletons chosen by programmers, e.g. pipeline, processor farm, and master-slave skeletons.

Mixed task and data parallel approach has several benefits over a pure data-parallel one. Pure HPF only allows to choose whether loops have to be executed in parallel or sequentially. If a loop is parallel, it is scheduled among all the processors involved in the execution. Unfortunately, distinct parts of the application often scale differently, and some of them may also show an increase in the completion time as more processors are used for the parallel execution. This behavior can be frequently observed on our testbed architecture, a cluster of SMP machines connected by a high-latency communication network.

On the other hand, our mixed task and data-parallel approach allows performances to be optimized on a global basis by independently choosing the best degree of parallelism for the various parts of the application, each implemented by a different HPF task. In addition, effective mapping strategies can be exploited, which map tightly-coupled HPF tasks onto single SMPs, so that only intra-task $COLT_{\mathtt{HPF}}$ communications occur over the high-latency network. Although $COLT_{\mathtt{HPF}}$ ensures the minimization of the number of communications needed to transfer distributed data between two data-parallel tasks, these communications are characterized by high latencies, mainly due to the TCP/IP protocol used. However, intra-task communications are generally less critical than inter-task HPF-related ones, because they do not require tight synchronization between the processes.

We applied our parallelizing strategy to a physics application, implemented with a pipeline skeleton composed of three different HPF tasks, where we have discussed in depth the profitability of replicating the middle stage of the pipeline. We also experimented an alternative, more dynamic, master-slave skeleton to implement the same test-case application.

We conducted several experiments on our testbed cluster of SMPs. Static optimizations, such as the choice of the best degree of parallelism for the HPF tasks, were carried out on the basis of the knowledge of task scalability. The $COLT_{\mathtt{HPF}}$ implementation obtained encouraging performances, with improvements of up to 353% in the completion time over the pure HPF implementation. Finally, it is worth noting that the largest improvements with respect to the pure HPF implementation were obtained in the tests involving a smaller dataset. This behavior is particularly interesting because for many important 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 [12].

References

[1] 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.

[2] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 2000. To appear.

[3] H.E. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, pages 74–84, July-Spet. 1998.

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

[5] N. Carriero and D. Gelenter. How to write a parallel program: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, Sept. 1989.

[6] 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.

[7] 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.

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

[9] E. Fermi, J. Pasta, and S. Ulam. Los Alamos Report LA 1940. In E. Segre', editor, *Collected Papers of Enrico Fermi*, volume 2, page 978. University of Chicago Press, 1965.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] High Performance Fortran Forum. *HPF Language Specification*, May 1993. Ver. 1.0.

[15] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, Nov. 1994. Version 0.8.

[16] High Performance Fortran Forum. *HPF Language Specification*, Jan. 1997. Ver. 2.0.

[17] 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.

[18] S. Orlando and R. Perego. A Comparison of Implementation Strategies for Non-Uniform Data Parallel Computations. *Journal of Parallel and Distributed Computing*, 52(2):132–149, 1998.

[19] S. Orlando and R. Perego. Scheduling Data-Parallel Computations on Heterogeneous and Time-Shared Environments. In *Proc. of EUROPAR'98*, pages 356–366, Southampton, UK, Sept. 1998. LNCS 1470, Spinger-Verlag.

[20] S. Orlando and R. Perego. $COLT_{\mathtt{HPF}}$, a Run–Time Support for the High–Level Coordination of HPF Tasks. *Concurrency: Practice and Experience*, 11(8):407–434, 1999.

[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.

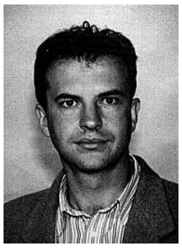
[22] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, Nov. 1997.

[23] Stefano Ruffo. The fput numerical experiment: time scales for the relaxation to thermodynamical equilibrium. In

D. Bambusi and G. Gaeta, editors, *Symmetry and Perturbation Theory*, page 188. Quaderni CNR, 1998.
[24] L. Smar and C.E. Catlett. Metacomputing. *Comm. of the ACM*, 35(6):45–52, June 1992.
[25] 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.
[26] Marco Vanneschi. PQE2000: HPC Tools for industrial applications. *IEEE Concurrency*, 6(4), October-December 1998.

Salvatore Orlando received a Laurea degree cum laude and a Ph.D. degree in Computer Science from the University of Pisa in 1985 and 1991, respectively. He is currently an assistant professor at the Department of Computer Science of the Ca' Foscari University of Venice. His research interests include parallel languages and computational models, optimizing and parallelizing tools, parallel algorithm design and implementation.
E-mail: orlando@unive.it

Paolo Palmerini received a Laurea degree in Physics from the University of Florence in 1996. He is currently a researcher at CNUCE, an Istitute of the Italian National Research Council (CNR). His research interests are high performance computing and parallel data mining.
E-mail: paolo.palmerini@cnuce.cnr.it



Raffaele Perego received his Laurea degree in Computer Science from the University of Pisa in 1985. He is currently a researcher at CNUCE, an Istitute of the Italian National Research Council (CNR), and a contract professor at the Department of Computer Science of the University of Pisa. His research interests include design and analysis of parallel algorithms, parallel languages and tools, high performance computing.
E-mail: raffaele.perego@cnuce.cnr.it