Mixed Data and Task Parallelism with HPF and PVM

Salvatore Orlando

Dip. di Informatica, Università Ca' Foscari di Venezia, Venezia, Italy. E-mail: orlando@unive.it

Paolo Palmerini, Raffaele Perego

Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy. E-mail: r.perego@cnuce.cnr.it

Abstract

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/communication layer implemented on top of PVM. $COLT_{\texttt{HPF}}$ is portable and can be used independently of the HPF compilation system exploited, it allows instances of cooperating HPF tasks to be created either at loading time or at run-time. We claim that $COLT_{\texttt{HPF}}$ can be easily used to implement HPF applications which exploit structured or unstructured forms of task parallelism on either homogeneous or heterogeneous computing environments. To validate our approach we present some experimental results obtained by running a physics application implemented by means of $COLT_{\texttt{HPF}}$ 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 [9]. 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. For example, many applications of computer vision, image and signal processing do not allow efficient data–parallel solutions to be devised, but can be efficiently structured as *pipelines* of data-parallel tasks where low-scalable stages (e.g. the ones which deal with external devices) may be run on a few processors, while most processors are reserved for more scalable parts of the application [2, 7]. 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 [6, 19, 5, 3, 1, 10]. Also the latest version of the HPF standard, HPF 2.0, proposes an unintuitive and inflexible approach to express pipelining among HPF tasks [11]. A different approach regards the design of an HPF binding for a standard message-passing library such as MPI [5]. The integration of task and data parallelism has also been proposed within an object-oriented coordination language such as Opus [3]. 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 task parallelism has already been proposed by Foster et al. for the above mentioned HPF-MPI binding [5]. With respect to this proposal, $COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ is through a compiler of a high-level coordination language aimed at facilitating programmers in structuring their data-parallel tasks according to compo-

sitions of common forms of *task-parallelism* such as pipelines, processor farms, and dynamic master-slave paradigms [8, 12]. Such a coordination language and the associated skeleton-based compiler have been proposed elsewhere [17]. The previous version of $COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ 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 [16, 14]. 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 [21] 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 $COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ on a heterogeneous environments composed by a network of PCs that run Linux. Some of these PCs are Symmetric Multi-Processor machines (SMPs), and can thus be considered as homogeneous sub-systems.

The paper is organized as follows. Section 2 introduces the design and functionalities of $COLT_{\texttt{HPF}}$, and describes its implementation on top of the PVM communication layer. In Section 3 the application used to validate our approach is presented, and the results of the experiments conducted are discussed in depth. Finally, Section 4 draws some conclusions and outlines future work.

2 $COLT_{\texttt{HPF}}$ implementation

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 [9]. A previous version of $COLT_{\texttt{HPF}}$ [17] exploited MPI for both inter and intra–task communications, and required slight modifications to the HPF run–time in order to avoid interferences between the two communication contexts. The need to modify the HPF run–time meant that commercial black-box HPF compilers could not be used. The release of $COLT_{\texttt{HPF}}$ discussed in this paper uses PVM for all the inter-task communications, and relies on other compiler specific mechanisms for HPF intra–task communications.

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. HPF tasks participating in a $COLT_{\texttt{HPF}}$ application can thus be instances of HPF modules compiled separately with any HPF compiler, and linked to the $COLT_{\texttt{HPF}}$ library. We exploited the dynamic features of PVM to support a dynamic HPF task model as well according to which $COLT_{\texttt{HPF}}$ tasks can ask for other tasks to be created at run–time.

In our experiments we tested $COLT_{\texttt{HPF}}$ with the `pghpf` compiler from the Portland Group Inc., and the `gmdhpf` compiler (Adaptor) from GMD-SCAI. The former was configured for using the proprietary RPM layer, while the latter for the MPI communication layer.

2.1 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 first main issue is thus the mechanism by which the various HPF tasks are loaded and start their execution. As mentioned above, the HPF tasks are separate executable programs linked with the $COLT_{\texttt{HPF}}$ and PVM libraries. These HPF executables have their own 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 produces an executable that accepts specific command-line parameters.

Our solution was to include in the $COLT_{\texttt{HPF}}$ initialization primitive, `colt_init()`, the invocation

of the PVM primitive `pvmfmytid()`, through which all the processes involved in the execution of the various HPF tasks enroll themselves into the PVM machine and receive their PVM identifiers (hereafter `pvm_tid`s). In order to exploit the PVM communication layer, the $COLT_{\texttt{HPF}}$ support needs to know the `pvm_tid`s of all the processes created and their relationship with the corresponding HPF tasks, each of which is identified by a distinct task identifier (hereafter `colt_tid`). `colt_tid`s are used to identify HPF tasks involved in the $COLT_{\texttt{HPF}}$ application. $COLT_{\texttt{HPF}}$ communication primitives use, in fact, `colt_tid`s to identify HPF tasks (e.g. the `colt_tid` of the receiver task in a send primitive), while the implementation of the primitive exploits the suitable `pvm_tid`s of the processes running the HPF task involved.

In order to gather information about the `colt_tid`s of the other running tasks and the associated sets of `pvm_tid`s, the $COLT_{\texttt{HPF}}$ initialization primitive interacts with a specific $COLT_{\texttt{HPF}}$ 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. This paper discusses in depth *static* task creation, while issues related to the dynamic creation and management of the HPF tasks are only dealt with briefly.

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:

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_tid`s of all the started processes as well as their HPF task ownership.

5. stores the information received at the previous step in a table, hereinafter called *mapping table*, and broadcasts this 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 $COLT_{\texttt{HPF}}$ 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_tid`s 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()`[1], and **(2)**, ask the *colt_daemon* for the *dynamic* creation of new HPF tasks. HPF tasks through a specific $COLT_{\texttt{HPF}}$ primitive, can in fact require the *colt_daemon* to spawn a given HPF *executable module* on a set of PVM virtual processors, thus creating a new HPF task. The parameters of this $COLT_{\texttt{HPF}}$ primitive are analogous to those specified in the *configuration file*. Other running tasks clearly may need to find out whether a given task has been started. To this end they can query *colt_daemon* on the status of the running application. In particular, a task can probe for the existence of a given HPF task and obtain the `pvm_tid`s of all the processes running the task.

2.2 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. Communicating distributed data between data-parallel tasks instead 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 [18]. 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

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

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. For a more detailed discussion about channel creation and the relative primitives to use, please refer to previous papers on $COLT_{\texttt{HPF}}$ [17, 15].

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.

HPF tasks may need to receive data from a task which is non-deterministically chosen from several possibilities. The problem is that to ensure correctness, the same non-deterministic choice must be globally made by all the processes running the receiving HPF 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 processes of $\overline{T}$ in order to accomplish the overall communication. $COLT_{\texttt{HPF}}$ solves this problem by providing an appropriate receive primitive that causes only the root process of the receiver task to make the non-deterministic choice of the sender, and then to broadcast its choice to all the other processors of the receiver task. Only when this choice has been communicated to the other receiving processes, do they begin to receive the data from the selected source task as discussed above.

Moreover, the non-deterministic primitive may happen to receive a message from an unknown task dynamically created at run–time by some other task. In this case the receiving task asks the *colt_daemon* for the mapping information about the new HPF task before actually reading the message data.

3 A case study from physics

The test-case application chosen to show the benefits of our mixed task and data-parallel approach is a physics simulation, used to study the dynamic properties of a system with many degrees of freedom. The chosen system is a Fermi-Pasta-Ulam (FPU) chain of N oscillators, coupled with unharmonic terms [4, 20]. A characterization of the way such Hamiltonian systems reach their asymptotic equilibrium state has important implications in understanding some macroscopic phenomena (e.g. phase transitions) in terms of the microscopic properties of the system. Due to the complexity of such systems, numerical simulation remains one of the most effective tools of investigation.

In the following we first introduce the equations describing the physics system, then we discuss the simulation algorithm and its HPF implementation. Finally the $COLT_{\texttt{HPF}}$ mixed task- data-parallel implementation is analyzed in depth. The performance results obtained by running both the pure HPF and $COLT_{\texttt{HPF}}$ versions of the application are reported and critically discussed. The parallel architecture exploited for the experiments is a cluster of three SMP Linux PCs, where each PC is equipped with two Pentium II - 233 MHz processors and 128 MB of main memory. The PCs are interconnected by means of a 100 Mb/sec switched Ethernet. The HPF compiler used for the experiments was `pghpf` from the Portrald Group Inc.

3.1 The Fermi-Pasta-Ulam β model

An FPU chain of N oscillators coupled with unharmonic terms is described by the Hamiltonian:

$$H(q,p) = \sum_{i=0}^{N-1} (\frac{p_i^2}{2} + \frac{(q_{i+1} - q_i)^2}{2} + \beta \frac{(q_{i+1} - q_i)^4}{4}) \tag{1}$$

with periodic boundary conditions $q_0 = q_{N-1}, p_0 = p_{N-1}$. For such a system the time τ needed to reach the asymptotic state, where energy is equipartitioned among the normal modes, is:

$$\tau = \tau_0 e^{(\frac{\epsilon_0}{\epsilon})^\delta} \tag{2}$$

where ϵ is the energy density, and the constants ϵ_0 and δ are approximately 1. We want to investigate the energy range within which equation (2) holds.

To measure energy equipartition we use the *spectral entropy* [13] given by:

$$S(t) = -\sum_{k=0}^{N-1} w_k(t) \ln w_k(t) \tag{3}$$

where the probabilities w_k represent the energy in the k-th Fourier mode, i.e.:

$$w_k(t) = E_k(t) / \sum_l E_l \tag{4}$$

$$E_k(t) = \frac{1}{2}(P_k^2 + \omega_k^2 Q_k^2) \tag{5}$$

where P and Q indicate the Fourier transforms of p and q, respectively. To determine whether we have equipartition or not, we use a normalized parameter:

$$\eta(t) = \frac{S(t) - S_{\max}}{S(0) - S_{\max}} \tag{6}$$

which varies between 0 (no equipartition) and 1 (equipartition).

3.2 The simulation algorithm

Once the system is initialized with only a small packet of modes excited, i.e. with energy not equally distributed among the normal modes, we let the system evolve, and measure the energy until equipartition is observed. As explained in the previous section, this is measured by the quantity η. The pseudo-code of the algorithm is shown in Figure 3.2. The algorithm basically consists of a main loop (on `NEXT`) which simulates the temporal evolution of the system. Since we want to average over some realizations of the sequence of random numbers which generated the initial conditions, the global simulation (i.e. the loop on `NEXT`) is repeated for `NPAR` times. During the evolution we apply a symplectic algorithm of integration for the equation of motion implemented by routine `Evolve`, and we measure energy distribution (i.e. η) by calling routine `Energy`. The latter routine is the most computational expensive part of the code, since it involves two one-dimensional Fourier transforms and a reduction operation (see Equation 4). To fasten the simulation, energy is thus 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` in order to obtain an average value for η. The latter only simulates `NINT` steps of system evolution. The results of the `NPAR` simulations are collected by routine `Collect_Results`, while routine `Average` performs a statistical analysis. In Figure 2 $\eta(t)$ is plotted for different values of the energy

```
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 1: Pseudo-code of the simulation algorithm

density ϵ. As we explained above, we are interested in the behavior of the system for low energies. This means $\epsilon < 0.001$. Figure 2 shows that for such energies relaxation to equilibrium is extremely slow and a significant number of iterations are of the order of 10^{10}. We therefore have to choose `NEXT`, `NINT` and `DELTAT` so that `T=NEXT*(NINT+DELTAT)` approximately reaches such a limit.

The number of oscillators N constitutes the size of the problem. In principle we would have to let $N \to \infty$. A scaling analysis is not reported here, however it showed that for our purposes $N = 1024$ or $N = 2048$ give a good enough approximation of such a limit.

3.3 HPF implementation

The HPF implementation of the application was not particularly difficult. It was needed to modify the original sequential code to remove dependencies that were preventing independent loop exploitation, while data structures, above all one-dimensional arrays, were distributed BLOCK. Moreover, some subroutine calls were expanded to reduce overheads.

As already explained, the most intensive computational part is the evaluation of η which is performed by routine `Energy`. Figure 3 shows the execution times on our cluster for the HPF subroutines `Evolve` and `Energy` as a function of the number of processors. The execution times of routines `Init`, `Collect_Results` and `Average` are not plotted, since they are negligible with respect to 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,

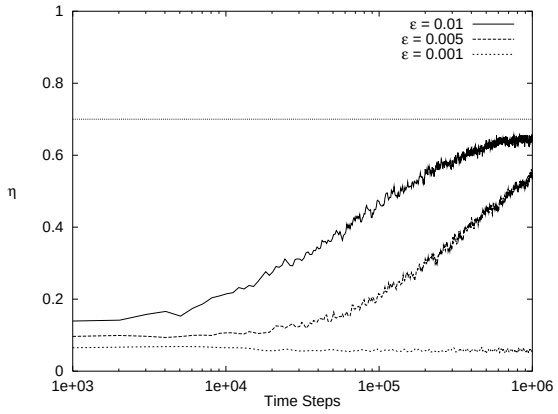

Figure 2: Results of simulations of an FPU chain with 2048 oscillators. The asymptotic state is reached for $\eta = 0.7$ due to fluctuations. Note that in the curve corresponding to $\epsilon = 0.01$, the observed time interval is too short to see any energy exchange among modes.

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 super-linear 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`. From the curves reported in Figure 3, it is clear that on our target system it makes no sense to exploit more than two or three processors for problem sizes of 1024 and 2048 oscillators, respectively.

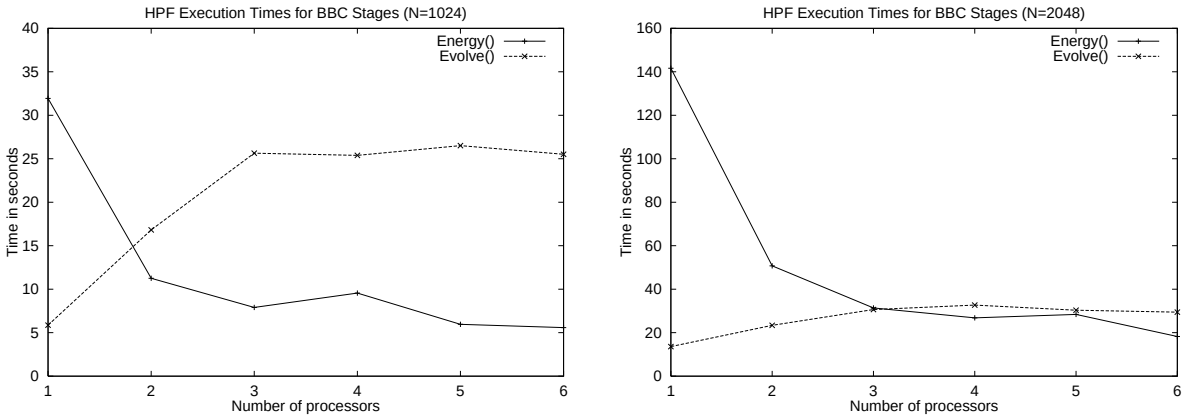

Figure 3: Execution times on a cluster of Linux SMPs for the HPF subroutines `Evolve` and `Energy` as a function of the number of processors. Plots refer to a few hundred iterations of the simulation of two FPU chains with 1024 and 2048 oscillators, respectively.

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 in the next section, the $COLT_{\texttt{HPF}}$ 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.

3.4 The $COLT_{\texttt{HPF}}$ implementation

In order to take advantage of the exploitation of task parallelism, we structured the application as a three stage pipeline of HPF modules. The first data-parallel stage takes care of the initialization of the FPU chain (routine `Init`) and of its temporal evolution (routine `Evolve`). The second stage computes energy equipartition (routine `Energy`) at given time instants on the basis of data coming from the previous stage. The third stage is responsible for collecting the results (routine `Collect_Results`) and for performing the statistical analysis (routine `Average`). The structure of the HPF codes executed by the three stages is illustrated in Figure 4. 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`. Each call to routine `Energy` has to follow each call to routine `Evolve` within the inner loop identified by `DELTAT`. Moreover, the computation of `Energy` also depends on the completions of all the calls to `Evolve` done during previous

$Stage_1$	$Stage_2$	$Stage_3$
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	do NPAR do NEXT do DELTAT recv from STAGE 1 call Energy() enddo DELTAT enddo NEXT send to STAGE 3 enddo NPAR	do NPAR recv from STAGE 2 call Collect_Results() enddo NPAR call Average()

Figure 4: $COLT_{\texttt{HPF}}$ pseudo-code implementing the case study application as a three stage pipeline.

iterations of the main loop over `NPAR`. On the other hand, `Evolve` does not depend on any data computed by either `Energy` or `Collect_Results`.

Figure 4 highlights that the first two stages execute the same temporal loops. $Stage_1$ sends a copy of the system state to $Stage_2$ at each iteration of the loop over `DELTAT`, and $Stage_2$ receives the data and evaluates the energy equilibrium on the received copy of the system state. On the other hand, $Stage_2$ sends its results to $Stage_3$ only `NPAR` times, precisely at the end of the two nested loops identified by `NEXT` and `DELTAT`. Therefore, if we try to model the data stream that flows into the three stages, we can consider it as a stream of `NPAR` elements. However, as far as the interaction between the first two pipeline stages is concerned, each stream element is furtherly split into a sub-stream of `NEXT * DELTAT` sub-elements, so that the transmission of the original stream item is completed only when all the elements of the sub-stream have been transmitted.

It is worth considering that the correctness of the computation performed by $Stage_3$ is preserved independently of the reception order of data from $Stage_2$. This allowed us to evaluate the possibility of replicating $Stage_2$, by inserting a *processor farm* within the original pipeline [17, 22]. Replicating $Stage_2$ entails $Stage_1$ dynamically dispatching its outcoming stream to the replicas of the second stage to balance the load, while $Stage_3$ receives the results from all the replicas non-deterministically. This can clearly result in a change in the reception order on $Stage_3$, which does not however affect the simulation

Table I: Completion times (in seconds) obtained by $COLT_{\texttt{HPF}}$ and pure HPF implementations of the case study application. The columns labeled *Structure* show both the number of physical processors assigned to the stages and the replication degree of the middle stage.

	Problem Size=1024				**Problem Size=2048**			
Procs	**HPF**	$COLT_{\texttt{HPF}}$		**Ratio**	**HPF**	$COLT_{\texttt{HPF}}$		**Ratio**
	sec.	*Structure*	sec.		sec.	*Structure*	sec.	
1	39.64	–	-	-	163.6	–	-	-
2	33.78	–	-	-	82.72	–	-	-
3	37.47	[1 (1) 1]	33.25	1.19	70.18	[1 (1) 1]	61.73	1.14
		$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	16.17	2.32		$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	53.26	1.32
4	34.27	[1 (2) 1]	14.11	2.43	71.84	[1 (2) 1]	53.14	1.35
6	35.78	[1 (4) 1]	9.52	3.76	56.90	[1 (4) 1]	33.81	1.68
		[1 (2,2) 1]	8.69	4.12		[1 (2,2) 1]	29.65	1.92

performed.

The curves in Figure 3 clearly show that, from a performance point of view, the efficiency of routine `Energy` is at its maximum when it is executed on only two processors. The main reason is that HPF communications are much more efficient when they occur within the same SMP machine. Hence, if $2 \cdot P$ processors are available for executing Stage_2, it always seems more profitable to replicate the stage into P copies, each running on two processors, rather than exploiting all the $2 \cdot P$ processors to execute a single instance of Stage_2.

Table I compares the execution times obtained with both $COLT_{\texttt{HPF}}$ and 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. As regards the $COLT_{\texttt{HPF}}$ implementation, the tests were performed by varying both the number of processors used, and the "organization" (i.e. degree of parallelism and replication of the stages) of the three stage pipeline implementing the application. The columns labeled *Structure* in the table indicate the mappings used for the $COLT_{\texttt{HPF}}$ tests where both the number of processors assigned to the stages and the replication degree of the middle stage were varied. 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. 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 it is run on several processors (see Figure 3) and the slight computational cost of the third stage of the pipeline.

The execution times measured with the $COLT_{\texttt{HPF}}$ implementations were always better than the HPF ones. The performance improvements obtained are quite impressive and range from 14% to 312%. If we consider the results in terms of absolute speedup over the execution on a single processor, the $COLT_{\texttt{HPF}}$ implementation with the problem size fixed to 1024, obtained speedups of 2.45 and 4.56 on three and six processors, respectively. On the tests conducted with 2048 oscillators, the speedups were 3.07 on three processors and 5.52 on six.

The largest improvements with respect to the pure HPF implementation were obtained in the tests where an FPU chain with a smaller number of oscillators is simulated. 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 [7]. As expected, the table also shows that due to the above discussed behavior of routine `Energy`, 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.

4 Conclusions

We have presented $COLT_{\texttt{HPF}}$, a coordination layer for HPF tasks, which uses PVM for implementing optimized inter-task communications. $COLT_{\texttt{HPF}}$ allows a mixed task and data-parallel approach to be exploited, where data-parallel tasks can be started either statically (at loading-time) or dynamically (on demand).

$COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ communications occur over the high-latency network. Although $COLT_{\texttt{HPF}}$ 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 as a pipeline of three different HPF tasks. Also, we have discussed in depth the profitability of replicating the middle stage of the pipeline.

We conducted several experiments on a cluster of SMP PCs. 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 $COLT_{\texttt{HPF}}$ implementation obtained encouraging performances, with improvements of up to 312% in the completion time over the pure HPF implementation.

Future work will be to design a visual programming tool to support our skeleton-based parallel programming methodology [17], and new skeletons suitable for time-shared heterogeneous environments, where static optimizations can be invalidated by workload changes occurring at run-time [16]. Finally, we are extending $COLT_{\texttt{HPF}}$ to allow HPF tasks to interoperate with tasks written in other programming languages (e.g., C, Java, Fortran).

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] R. Livi, M. Pettini, S. Ruffo, M. Sparpaglione, and A. Vulpiani. *Physical Review A*, 31:1039, 1985.

[14] 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.

[15] S. Orlando and R. Perego. An MPI-Based Run-Time Support to Coordinate HPF Tasks. In *Proc. of the EuroPVM/MPI98 Conference*, pages 289–296, Liverpool, UK, Sept. 1998. LNCS 1497, Spinger-Verlag.

[16] 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.

[17] S. Orlando and R. Perego. $COLT_{\texttt{HPF}}$, a Run–Time Support for the High–Level Coordination of HPF Tasks. *Concurrency: Practice and Experience*, 1999. To appear.

[18] 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.

[19] 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.

[20] 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.

[21] L. Smar and C.E. Catlett. Metacomputing. *Comm. of the ACM*, 35(6):45–52, June 1992.

[22] 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.