Coordinating HPF programs to mix task and data parallelism*

Salvatore Orlando
Dipartimento di Informatica
Univ. Ca' Foscari di Venezia
Via Torino, 155
30173 Mestre - Italy
orlando@unive.it

Paolo Palmerini
Istituto CNUCE
CNR
Via S. Maria, 26
56126 Pisa - Italy

Raffaele Perego
Istituto CNUCE
CNR
Via S. Maria, 26
56126 Pisa - Italy
r.perego@cnuce.cnr.it

Keywords

Task parallelism, Data Parallelism, Coordination Languages, HPF.

ABSTRACT

Experience in applicative fields, above all deriving from the development of multidisciplinary parallel applications, seems to suggest a model where an outer *coordination* level is provided to allow data parallel tasks to run concurrently and to cooperate each other. The inner computational level of this coordination model can easily be expressed with HPF, a high-level data-parallel language. According to this model, we devised $COLT_{\mathtt{HPF}}$, a coordination architectural layer that supports dynamic creation and concurrent execution of HPF tasks, and permits these tasks to cooperate though message passing. This paper proposes the exploitation of $COLT_{\mathtt{HPF}}$ by means of a simple skeleton–based coordination language and the associated source-to-source compiler. Differently from other related proposals, $COLT_{\mathtt{HPF}}$ is portable and can exploit commercial, standard-compliant, HPF compilation systems. We used a physics application as a test-case for our approach, 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 [13]. 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; 16; 10; 8; 6; 14]. Also the latest version of the HPF standard, HPF 2.0, provides extensions which allows restricted forms of task parallelism to be exploited [13]. 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 $COLT_{\mathtt{HPF}}$ (COordination Layer for Tasks expressed in HPF), a portable coordination layer for HPF tasks. $COLT_{\mathtt{HPF}}$ 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

*This work has been partially founded by HPCC/SEA contract number EU1063.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright ACM 0-89791-88-6/97/05 ..$5.00

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 [10]. 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 compositions of common *skeletons* for *task-parallelism* such as pipelines, processor farms, master-slaves, etc. [1; 14]. A previous version of $COLT_{\texttt{HPF}}$ used MPI as a message transport layer among HPF tasks. The exploitation of MPI, however, required 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 profitably be used to program heterogeneous networks of computers/workstations as well, where the adoption of a purely static task model may fail due to the large and often unpredictable variations in the load and power of the computational resources available.

We used a physics application as a test-case for validating 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. Moreover, it introduces the design and functionalities of $COLT_{\texttt{HPF}}$, and describes its implementation on top of PVM. In Section 3 the test-case application is presented, and two different $COLT_{\texttt{HPF}}$ 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 4 draws some conclusions.

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, and an inner *computational* level for data 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. On the other hand, 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 $COLT_{\texttt{HPF}}$, 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. $COLT_{\texttt{HPF}}$ 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. We are thus working to exploit $COLT_{\texttt{HPF}}$ by means of a high-level *skeleton*-based coordination language [1; 14]. The coordination of HPF tasks is expressed by means of specific language constructs, each corresponding to a very common skeleton for task parallelism, i.e. pipeline, processor farm, master-slaves, etc. Programmers choose the best skeleton for their purposes, and for each task in the skeleton specify the HPF *computational* code as well as the list of input/output data exchanged between the tasks. Skeleton are 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 skeleton. By instantiating the *templates*, the compiler produces the ensemble of HPF programs implementing the required mixed task and data parallel application.

$COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ 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 $COLT_{\texttt{HPF}}$ within the run-time of SkIE-CL, a skeleton-based multi-language coor-

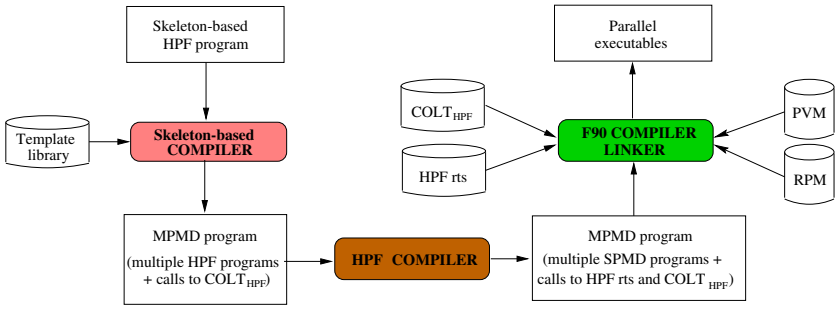

Figure 1: Structure of the compilation system implementing the proposed model for task and data-parallelism integration.

dination environment [2; 18].

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 $COLT_{\texttt{HPF}}$ mixed task and data parallel program corresponds to an ensemble of distinct SPMD parallel programs (an MPMD program) coordinated by means of $COLT_{\texttt{HPF}}$. By exploiting the *templates* corresponding to the skeletons specified by programmers, our *skeleton*-based compiler has thus to generate multiple HPF programs containing calls to the $COLT_{\texttt{HPF}}$ run-time support. Such multiple HPF programs then have to be compiled with the HPF compiler and linked with the $COLT_{\texttt{HPF}}$ and HPF libraries. Finally, note that both $COLT_{\texttt{HPF}}$ and HPF run-time supports use some low-level message-passing middleware. The `pghpf` compiler used exploits RPM, the PGI Real Parallel Machine system. On the other hand, $COLT_{\texttt{HPF}}$ exploits PVM for inter-task communications. 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. The overall structure of the hierarchy of compilers which implement our model for task and data parallelism integration is shown in Figure 1.

Inter-task communications.. The primitives to exchange array data structures constitute one of the main difficulties in implementing $COLT_{\texttt{HPF}}$. 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 at running time, the actual boundaries of array sections allocated on each processor executing the HPF task are not statically known. $COLT_{\texttt{HPF}}$ thus inspects at run-time the HPF support to find out on both the sender and receiver sides the actual mapping of any distributed array exchanged. By means of the Ramaswamy and Banerjee's *pitfalls* algorithm [15], all the processes involved in the communication then compute the intersections of their own array partitions with the ones of the processes belonging to the partner task. A global *communication schedule* is then derived. It establishes, for each sender process, the portion of array section which must be sent to each process of the receiver task. On the other side, each process of the receiver task computes which array portions

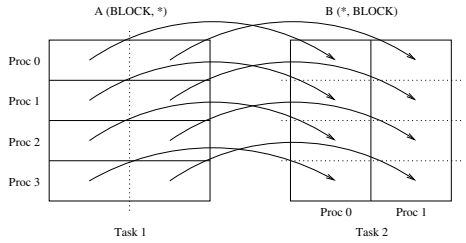

Figure 2: Point-to-point communications to send a bidimensional array from one HPF task, mapped on 4 processes, to another HPF task, mapped on 2 processes. Data distributions on the sender and the receiver tasks are `(BLOCK,*)` and `(*,BLOCK)`, respectively.

it has to receive from any of the sender processes. 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.
Since computing array intersections and communication schedules is quite expensive, and the same array transmission is usually repeated several times, $COLT_{\texttt{HPF}}$ reuses them when possible by storing this information into appropriate *channel descriptors*.
Finally, it is worth point out that, in order to obtain portability among different HPF compilation systems, $COLT_{\texttt{HPF}}$ exploits HPF standard features to interact with the HPF run-time system, and is implemented as a library of standard `HPF_LOCAL EXTRINSIC` subroutines [13]. This means that when a $COLT_{\texttt{HPF}}$ primitive is invoked, all the processes executing the HPF task switch from the single thread of the control model supplied by HPF to a true SPMD style of execution. In other words, through an `HPF_LOCAL` routine programmers can get the SPMD program generated by the HPF compiler under their control.

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 $COLT_{\texttt{HPF}}$ [14]. 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.

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. Since processes enroll themselves in the PVM machine at run-time, we exploited PVM dynamic process model to support a dynamic HPF task model as well: HPF tasks, which are compiled as separate executable SPMD programs linked with the $COLT_{\texttt{HPF}}$ and PVM libraries, enroll themselves in the $COLT_{\texttt{HPF}}$ application at run-time, and can ask for other HPF tasks to be created dynamically.

Usually HPF executables exploit compiler-dependent mechanisms for loading and starting on the parallel system. For example, the `pghpf` compiler produces an executable that accepts specific command-line parameters. Hence, while HPF tasks have to be launched by exploiting their proprietary mechanisms, $COLT_{\texttt{HPF}}$ provides an initialization primitive to enroll them in PVM. Through this primitive, all the processes running a task (i.e., the processes participating in the corresponding SPMD subprogram generated by the HPF compiler) enroll in PVM and receive their own PVM identifiers (hereafter `pvm_tid`s). Since these `pvm_tid`s must actually be used within $COLT_{\texttt{HPF}}$ to exchange PVM point-to-point messages among the processes running the various tasks, the $COLT_{\texttt{HPF}}$ implementation has to know the correspondence between the various HPF tasks, each one identified by a distinct `colt_tid`, and the sets of `pvm_tid`s corresponding to the groups of processes running the tasks themselves. $COLT_{\texttt{HPF}}$ thus maintains this knowledge within a replicated data structure, hereafter *mapping table*. In order to build the *mapping table*, the $COLT_{\texttt{HPF}}$ initialization primitive interacts with a specific $COLT_{\texttt{HPF}}$ daemon, hereafter *colt_daemon*. We introduced the *colt_daemon*, however, not only to centralize task registration, but also to handle *static* and, above all, *dynamic* task creation. Static task creation exploits a user-provided *configuration file* that 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. By means of the mechanisms provided by the OS and the specific HPF compilation system used, *colt_daemon* starts the various HPF tasks specified in the configuration file[1]. As soon as a spawned HPF task starts executing, it calls the $COLT_{\texttt{HPF}}$ initialization primitive to make all the processes running the task enroll in the PVM machine and exchange information with *colt_daemon*. In this way, *colt_daemon* receives from each spawned process the pair (`pvm_tid`, `colt_tid`), and builds the *mapping table* accordingly. Finally, the complete *mapping table* is broadcast to to all the processes running the spawned HPF tasks. Figure 3 illustrates the steps discussed above, by showing the interactions between the *colt_daemon* and two HPF tasks, each one running on a couple of workstations.

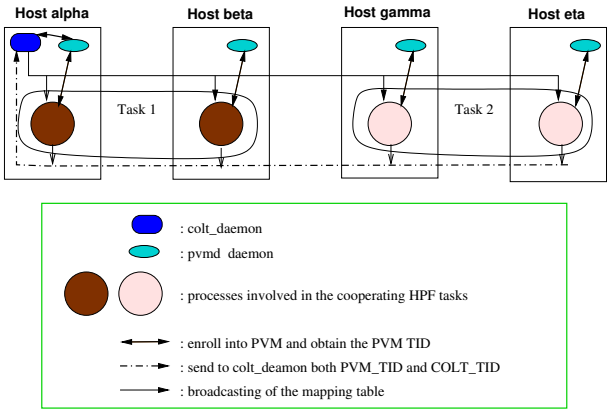

Figure 3: Interactions between *colt_daemon* and two HPF tasks, each exploiting two processors.

HPF tasks can furtherly interact with the *colt_daemon* to ask for the *dynamic* creation of new HPF tasks. The input parameters of the $COLT_{\texttt{HPF}}$ primitive for the dynamic spawning of an HPF task are the *pathname* of the executable file, the *degree of parallelism* of the task, and the list of *PVM hosts*. The unique `colt_tid` associated with the HPF task to be created is in this case chosen by *colt_daemon* and returned to the requesting HPF task. The interactions between the *dynamically* started processes and the *colt_daemon* are similar to those illustrated in Figure 3: *colt_daemon* updates its copy of the *mapping table* by adding a row concerning the newly created task, broadcasts the updated table to all the processes running the new task, and sends the new row added to the table to the task which asked for the dynamic creation.

3. THE TEST-CASE APPLICATION

The test-case application chosen to show the benefits of our mixed task and data-parallel approach simulates the dynamics of a chain of N oscillators, coupled with unharmonic terms. Such system is known as the Fermi Pasta Ulam (FPU) model [9]. The characterization of the way such Hamiltonian systems reach their asymptotic equilibrium state has important implications in understanding the

[1] *colt_daemon* supplies to the launched HPF tasks the *colt_daemon*'s `pvm_tid` and their unique `colt_tid`s.

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

dynamic properties of systems with many degrees of freedom. The pseudo-code of the algorithm is shown in Figure 4. The algorithm basically consists of a main loop (on `NEXT`) which simulates the temporal evolution of the system (routine `Evolve`), through the integration of the equations of motion. As the system evolves, energy distribution is measured until equipartition is observed. Energy distribution is measured by routine `Energy` which implements the most computationally 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. Note, in fact, that the temporal loop (on `NEXT`) includes two further nested loops. 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 particularly slow. In the energy range of interest, a number of time-steps of the order of 10^{10} have to be simulated before equipartition can be observed in the system.

Starting from a Fortran 77 code, the HPF implementation of our test-case was not particularly difficult. 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 5 shows the execution times of these HPF subroutines on our testbed 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 $N = 1024$ and $N = 2048$ 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, `Energy` and `Evolve` routines behave very differently. While `Energy` scale very well, and in some cases, due to the exploitation of the memory hierarchy, exhibit super-linear speedups, `Evolve` do not scale at all. This behavior of routine `Evolve` is due to stencil array references, whose implementation is very inefficient on our target platform. Further tests have shown that routine `Evolve` starts

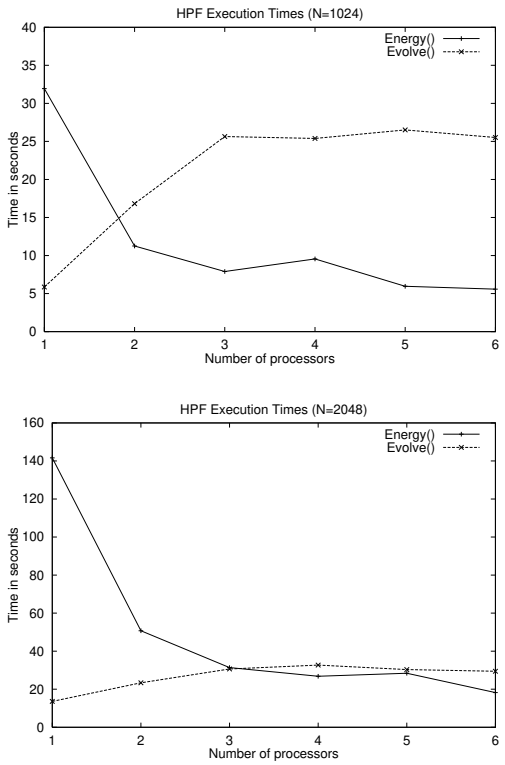

Figure 5: Execution times on a cluster of Linux 2-way SMPs for the HPF subroutines `Evolve` and `Energy` as a function of the number of processors.

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`. While this is not possible by using a pure data-parallel model, we can trade off between these two contrasting requirements by adopting a mixed task and data parallel approach.

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 $COLT_{\texttt{HPF}}$ primitives which initialize the communication channels and exchange data between the data-parallel tasks. For our purposes we thus 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 $COLT_{\texttt{HPF}}$ and

PVM libraries to obtain the application executables. 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 $COLT_{\texttt{HPF}}$ as a pipeline of three stages, where each stage is implemented by a distinct HPF program. As shown in in Figure 6, 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 4).

The overall throughput of such 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 [17]. 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. To avoid load imbalance problems, the templates adopted to support stage replication exploit dynamic policies, according to which replicated stages *self-schedule* jobs by signaling to the preceding stage their availability to receive a further stream element, while the following stage *non-deterministically* receives the results from all the copies of the replicated stage.

Returning to our test-case application, experiments showed that the middle stage, $Stage_2$, is the slowest one. Moreover, $Stage_2$ 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 $Stage_2$. Figure 5 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. Therefore, 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 $COLT_{\texttt{HPF}}$ pipeline and the pure HPF implementations of our test-case 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 $COLT_{\texttt{HPF}}$ 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, 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 25% to 353%. If we consider the results in terms of absolute speedup over the execution on a single processor[2], the $COLT_{\texttt{HPF}}$ implementation with the problem size N 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.

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`). 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.

The $COLT_{\texttt{HPF}}$ templates implementing the master-slave skeleton adopt another dynamic strategy besides the self-scheduling policy used to dynamically dispatch jobs onto the created slave tasks. This policy enhances the handling of unpre-

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

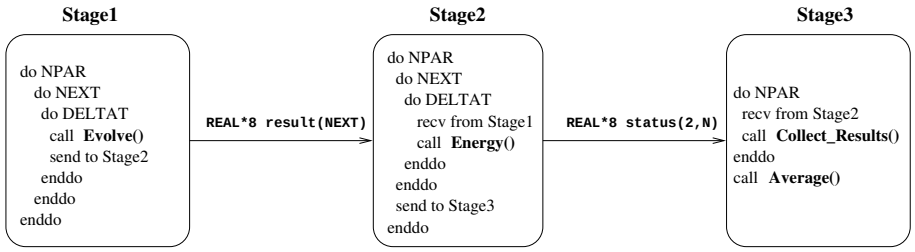

Figure 6: Structure of the three stage pipeline implementing the case study application.

Table 1: Completion times (in seconds) obtained by the $COLT_{\texttt{HPF}}$ and the pure HPF implementations of the test-case application.

	Problem Size=1024				Problem Size=2048			
Procs	**HPF**	$COLT_{\texttt{HPF}}$ **(pipeline)**		**Ratio**	**HPF**	$COLT_{\texttt{HPF}}$ **(pipeline)**		**Ratio**
	sec.	*Structure*	sec.		sec.	*Structure*	sec.	
1	34.6	–	-	-	148.0	–	-	-
2	38.2	–	-	-	87.2	–	-	-
3	36.4	$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	15.6	2.33	70.4	$[\frac{1}{2}\ (2)\ \frac{1}{2}]$	56.0	1.26
4	41.7	$[1\ (2)\ 1]$	14.7	2.84	69.3	$[1\ (2)\ 1]$	55.8	1.25
5	35.3	$[\frac{1}{2}\ (2,2)\ \frac{1}{2}]$	7.7	4.57	55.2	$[\frac{1}{2}\ (2,2)\ \frac{1}{2}]$	31.0	1.78
		$[\frac{1}{2}\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ \frac{1}{2}]$	7.6	4.64		$[\frac{1}{2}\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ \frac{1}{2}]$	27.8	1.98
6	34.0	$[1\ (4)\ 1]$	9.9	3.43	52.1	$[1\ (4)\ 1]$	32.6	1.59
		$[1\ (2,2)\ 1]$	7.6	4.47		$[1\ (2,2)\ 1]$	30.2	1.72
		$[1\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ 1]$	7.5	4.53		$[1\ (\frac{2}{2},\frac{2}{2},\frac{2}{2},\frac{2}{2})\ 1]$	26.8	1.96

dictable 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. 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.

The completion times measured by running the master-slave implementation on an un-loaded system are very similar to those obtained with the pipeline version of the application when the same number of processors is exploited. 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 2 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 $COLT_{\texttt{HPF}}$ 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 un-loaded system) and the master pays additional overheads for initializing the $COLT_{\texttt{HPF}}$ 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.

Table 2: Dynamic vs. Static master-slave implementations.

	Problem Size=2048			
	Static **master-slave**		*Dynamic* **master-slave**	
LOAD	**slave no.**	**sec.**	**slave no.**	**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

4. CONCLUSIONS

We have presented $COLT_{\texttt{HPF}}$, a coordination layer for HPF tasks, which exploits PVM to implement optimized inter-task communications. $COLT_{\texttt{HPF}}$ 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; 16], 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 $COLT_{\texttt{HPF}}$ 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.

We showed, by using a simple test-case application, implemented with both a pipeline and a master-slave skeleton, the usefulness of a mixed task and data-parallel approach, which 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.

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 $COLT_{\texttt{HPF}}$ 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].

5. 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*, 1999. 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] High Performance Fortran Forum. *HPF Language Specification*, Jan. 1997. Ver. 2.0.

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

[15] 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.

[16] 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.

[17] 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.

[18] Marco Vanneschi. PQE2000: HPC Tools for industrial applications. *IEEE Concurrency*, 6(4), October-December 1998.