Integrating task and data parallelism with taskHPF

S. Ciarpaglini[†], L. Folchi[†], S. Orlando[‡], S. Pelagatti[†], and R. Perego[°]

[†] Dip. di Informatica, Università di Pisa, Corso Italia, 40 - Pisa, Italy
[‡] Dip. di Informatica, Università Ca' Foscari di Venezia, Via Torino, 155 - Mestre, Italy
[°] Istituto CNUCE, C.N.R., Via Alfieri, 1 - Ghezzano (Pisa), Italy

Abstract

Many applications exhibit a large amount of potential parallelism that can be exploited at both data *and* task *levels. In this paper, we consider applications which can be structured as ensembles of independent data parallel HPF modules (hereafter* HPF tasks*), which interact according to static and predictable* patterns*. In order to make easy and effective to program such applications, we devised taskHPF, a coordination language in which programmers can define the interaction patterns among HPF tasks in a declarative way. We examine a small example application to discuss the benefits of our approach, and we show how taskHPF programs can be translated into efficient message-passing code.*

Keywords: Patterns, Coordination languages, Task parallelism, Data Parallelism, HPF.

1 Introduction

Many applications show potential for exploiting both data and task parallelism. Data parallelism is characterized by the application of the same operation to different parts of a related data set. In task parallelism, distinct parts of the program proceed independently on generally distinct sets of processors. For example, applications in signal and image processing are usually composed of a set of potentially data parallel tasks interacting to compute a stream of homogeneous data sets [2, 3, 18]. Other applications benefiting from a mixture of task and data parallelism are multidisciplinary applications, in which different parts are developed independently as data parallel programs and interact at a coarser task level. This has caused a growing interest in models and systems which allow the expression (and the exploitation) of both task and data parallelism [2, 9, 15, 19, 20].

The main contribution of this paper is the definition of taskHPF, a high level coordination language to express task parallelism among a collection of data parallel HPF tasks, and the description of its compilation system. taskHPF allows an application to be organized as a combination of common patterns, such as pipelining or replication. The taskHPF compiler is able to optimize and tune the global application structure and the resources devoted to each part of the task parallel application.

The paper is organized as follows. Section 2 gives an overview of the approach and introduces taskHPF. Section 3 discusses a small example application and some experimental results. Then, Section 4 discusses the taskHPF compiler and implementation issues. Finally, Section 5 draws conclusions.

2 Overview of taskHPF

taskHPF is a high level coordination language which allows to compose together a set of HPF programs, called *HPF tasks*, which process homogeneous streams of data. An HPF task is defined by supplying its HPF computational code along with an input and output interface. Then, these tasks can be composed using predefined structures, called *patterns* or skeletons. The rationale here is to provide the user with an easy way to define the most common task

interaction structures in the class of applications at hand. For example, taskHPF provides a *pipeline pattern*, to pipeline sequences of HPF tasks in a primitive way. Then, it provides directives which help the programmer in balancing the pipelined stages. An `ON PROCESSORS` directive fixes the number of processors assigned to an HPF task, thus devoting more resources to critical stages. A `REPLICATE` directive can be used to replicate non-scalable stages. This improves the pipeline throughput as different data sets can be computed in parallel on different sets of processors. Patterns can be composed together to build complex structures in a declarative way, without explicit management and control of parallelism in the user code.

The taskHPF compilation system translates the user defined structure in efficient message-passing code. Translation exploits a standard HPF compiler and a PVM version of $COLT_{\texttt{HPF}}$, a library which provides mechanisms for loading the HPF tasks on set of processors and for optimized inter-task communications [12, 13].

The main advantage of this approach is to supply programmers with a concise, pattern–based, high-level declarative way to describe the task interaction of their HPF modules. This shortens the development time and allows several structures to be easily tested. Moreover, the explicit declaration of coordination patterns simplifies the job of the compiler, which can exploit a library of specialized "wrappers", associated with the various patterns exploited, to enclose the user-supplied code. The static knowledge of the application structure also permits effective mapping techniques to be applied [5, 18]. Note that no other approaches presented in literature provide language constructs to express coordination patterns for data parallel tasks.

Among the diverse previous proposals to mix data and task parallelism, it is worth citing the Fx language [10], which allows programmers to define a flat ensemble of data parallel tasks (i.e. calls to subroutine, which are written in an HPF dialect), where data dependencies between tasks are expressed by well-defined input/output directives. Clustering or replication of tasks are poorly expressed with mapping directive on a processor grid. The new standard for HPF, HPF 2.0 [11], also permits mixing task and data parallelism. It is not clear at the moment whether this new standard will eventually be embraced by industrial HPF compilers. In HPF 2.0, communications between tasks are explicitly accomplished by simple *assignments* outside specific code blocks associated with independent tasks. Unfortunately, these 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 subroutine boundaries, and non-deterministic communication patterns, useful to implement dynamic policies like load balancing, cannot be expressed.

Finally, differently from the proposals discussed above, taskHPF does not need the adoption of new task parallel HPF constructs to express task parallelism. taskHPF is simply a coordination layer for HPF tasks which are separately compiled an off-the-shelf HPF compiler[1], while the task parallel coordination level is provided by the portable $COLT_{\texttt{HPF}}$ library.

Execution model of taskHPF. A taskHPF program defines a network of cooperating HPF tasks, where each task is assigned to a disjoint set of processors.

For each task, users define a list of input and output variables, and three sections of code: the *startup code*, the *body code*, and the *endup code*. The startup code contains HPF local variables declaration and distribution directives for both local and I/O array variables. Moreover, it may contain HPF code which initializes the state of a task. This code is executed once, before reading input variables. The body code is executed when data is read in the input variables. It produces new values for the output variables. The endup code is exe-

[1] We are currently using the version 3.0-4 of the `pghpf` compiler by the Portland Group Inc. for clusters of Linux PCs.

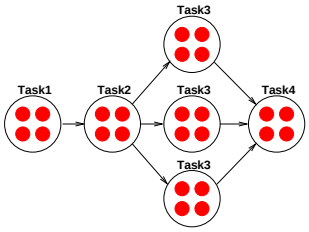


Figure 1: Structure of the parallel computation defined by `ts-pipe`.

cuted before terminating the task. Typically this part specifies how the computation results must be saved on a file or any action the task should do before exiting.

Figure 1 shows an abstract representation of the network of interacting tasks defined by the `ts-pipe` program sketched below, where the called *modules*[2] are instances of four distinct tasks: Task1, Task2, Task3, and Task4. Note that the external pattern exploited by the program is a pipeline one, and that Task3 is declared as a replicated module.

```
PIPE ts-pipe IN() OUT()
   <Task1 module call>
   <Task2 module call>
   REPLICATE(3) <Task3 module call>
   <Task4 module call>
ENDPIPE
```

Task1 is the *source task*, which generates the stream of items to be computed, while Task4 is the *sink task*, which consumes the stream without generating any output. For this reason, the input variable list of a source task and the output variable list of a sink task must be empty. Stream generation is carried out by the body code of the source task, which is repeatedly activated until the user code calls `taskHPF_end()`. `taskHPF_end()` causes the transmission of an `END_OF_STREAM` mark on the output channels (propagating termination), the execution of the endup code of the task, and the termination the task.

The execution of a taskHPF program happens as follows. All the tasks are independent HPF programs. The user defined *body code* of each HPF task defines a data parallel computation to be applied to each input item. The result stream is passed to next stage in the pipelined application structure.

At the beginning tasks configure their communication channels and initialize support variables. Then, all tasks except the source task are blocked trying to receive the first item of the input stream. When a new item is received, it is processed and its results are sent off to the next stage. HPF tasks behave in a different way when they are located immediately before (*dispatcher task*) or after (*collector task*) a replicated stage. When a task is a dispatcher (e.g. Task2), it must deliver the result to only one copy of the replicated next stage (e.g. Task3). This is done transparently by the support of both the stages, which ensures the load balancing between the copies using an on-demand dynamic scheduling policy. When a stage is a collector (e.g. Task4), it can receive input data from any of the copies of the previous stage (e.g. Task3). Again, the support hides the matter by receiving from any replica ready to deliver a results, thus merging the different streams of results in a single input stream from the previous stage. Finally, a stage can be both a collector and a dispatcher and combine both behaviors described above.

This execution scheme works also for nested pipeline patterns, i.e. also if the called modules of the example above were pipeline patterns instead of simple tasks.

3 A simple case study

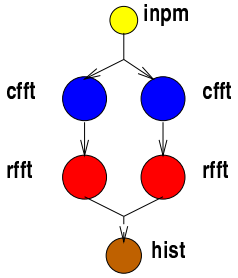
In order to illustrate the potential of taskHPF, we use a small kernel that was already presented in literature [10, 19], and represents the structure of a large class of image and signal processing applications. The kernel application is *FFT-Hist*, where a stream of two–dimensional COMPLEX matrixes is processed sequentially by a *pipeline* of four tasks. Matrixes can be processed in any order, since the computations performed on them are independent 2D FFTs. The first task in the pipeline, `inpm`, reads the matrixes from I/O, and for-

[2] The exact taskHPF syntax of calls is not shown.

```
PIPE pipe-fft IN(COMPLEX A(N1,N2)) OUT(COMPLEX C(N1,N2))
  cfft IN(A) OUT(COMPLEX B(N1,N2))  ON PROCESSORS(4)
  rfft IN(B) OUT(C)                 ON PROCESSORS(4)
ENDPIPE


PIPE main IN() OUT()
  inpm IN() OUT(COMPLEX A(N1,N2))  ON PROCESSORS(1)
  REPLICATE(2) pipe-fft IN(A) OUT(COMPLEX B(N1,N2))
  hist IN(B) OUT()                 ON PROCESSORS(2)
ENDPIPE
```

(a)



(b)

Figure 2: The code **(a)** and the task graph structure **(b)** of our *FFT-Hist* example.

wards them to task `cfft`. `cfft` applies the 1D FFT algorithm to the columns of each matrix received. Therefore, in order to obtain independent data parallel computations, the input matrix must be distributed `(*,BLOCK)`. `cfft`, after computing 1D FFT on every matrix column, sends the matrix to `rfft`, which computes the 1D FFT on the rows. In this case, we have to use a `(BLOCK,*)` distribution. Finally, matrixes produced are sent to task `hist`, which collects and analyzes them, and writes the results to an output file.

Figure 2.(a) shows a portion of the taskHPF program implementing *FFT-Hist*. We do not show the HPF code of the four tasks involved. The structure expressed by the program in Figure 2.(a) corresponds to the interaction graph shown in Figure 2.(b). Note that we have a `PIPE` pattern, called `pipe-fft`, which includes `cfft` e `rfft`[3]. Since the two tasks scale quite well [10, 19], we map each of them on 4 processors (`ON PROCESSORS` directive). This `pipe-fft` module is then composed with `inpm` and `hist` within an external `PIPE` pattern, called `main`. Note that `pipe-fft` is replicated 2 times to improve scalability (`REPLICATE` directive). Finally, `inpm` and `hist` are mapped on a few processors because they perform mainly I/O operations which, in the absence of a parallel file system, do not scale with the number of processors used.

The above structure for the *FFT-Hist* program is only discussed to illustrate the potentiality of our language. In fact, we were not able to conduct experiments with a program exactly structured as shown in Figure 2. The only machine available for the experiments was a cluster of three 2-way Linux PCs (6 nodes) interconnected by a 100BaseT switched Ethernet. Thus, experiments were conducted with less resources than those required by the program structure illustrated in Figure 2, requiring 18 processors. Instead, we exploited a program structured as a single pipeline pattern, composed of the four stages discussed above. The first and the last tasks were run on a single processor, while we assigned two nodes to the most time-consuming tasks, i.e. `cfft` and `rftt`. The structure used can be obtained from the code shown in Figure 2.(a), by only changing the `ON PROCESSORS` and `REPLICATE` directives. To evaluate the results achieved, we compared the execution times of our taskHPF program with the times obtained running a pure HPF version of the same program, where all the available processors were exploited to execute the four tasks. We fed the two versions of the program with a stream of 20 matrixes, and we observed the same advantages cited in literature about the benefits of exploiting parallelism at both the task and data levels on this application. When 256×256 matrixes were used, the taskHPF program took 9.2 *secs*, while the pure HPF one took 15.8 *secs*. For 512×512 matrixes, execution times were 39.69 *secs* and 44.67 *secs*, for taskHPF and pure HPF implementations, respectively. In the former case, the taskHPF program was 1.7 times faster than the HPF one, while in the latter case the

[3]Note the `IN(..)` and the `OUT(..)` lists containing typed elements composing, respectively, the input and the output data streams.

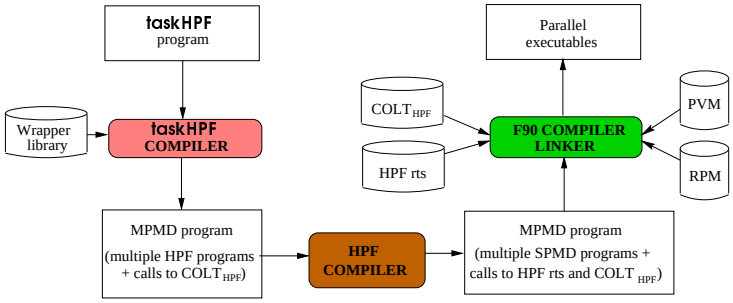


Figure 3: Structure of the taskHPF compilation system.

taskHPF program was 1.12 times faster.

4 Implementation

The integration of task and HPF data parallelism presents several implementation issues, deriving from the need to coordinate parallel tasks organized according to different execution styles. Data parallel tasks usually adopt an SPMD style, while task parallelism is generally exploited among independent entities running in MPMD style. Moreover, task parallel entities tend to be more dynamic needing the ability to spawn new activities during execution. We tackle these issues exploiting the structure and the properties of the patterns used to organize a taskHPF application. Figure 3 shows how taskHPF programs are compiled down to a given parallel machine.

The taskHPF compiler generates a set of independent HPF programs, where the user–defined code of each HPF task defines a data parallel computation to be applied on each input item. This is done in two steps. First, it derives the intra-task interaction structure and then it generates the appropriate wrapper code for each taskHPF task. The wrapper code differs depending on the role and the number of input and output channels of each task. The taskHPF compiler has a library of "canned" wrap codes, called *task wrappers*. Task wrappers encapsulate the common code for all the wrappers of a certain kind and can be specialized by plugging in program-dependent information, such as actual type and sender/receiver identifiers of message passing primitives, and user defined HPF code.

Task wrappers are defined using HPF with calls to $COLT_{\texttt{HPF}}$ library [12, 13], and have special markers for the parts to be plugged in the instantiation phase. $COLT_{\texttt{HPF}}$ provides primitives for optimized communications of data among processes and HPF task spawning.

The program generated by the taskHPF compiler is a flat ensemble of distinct HPF programs interacting by means of $COLT_{\texttt{HPF}}$. All these programs are then passed to an HPF compiler (which produces distinct SPMD f90/f77 parallel programs), and eventually are linked to relevant libraries to be translated in executable code. Note that also the low-level communication middle-wares used by $COLT_{\texttt{HPF}}$ and by the HPF program must be linked. For instance, the `pghpf` compiler we use exploits RPM, the PGI Real Parallel Machine system, while $COLT_{\texttt{HPF}}$ uses PVM.

In the following two subsections, we give some details of the compiler structure and of the characteristics of the $COLT_{\texttt{HPF}}$ library.

4.1 The taskHPF compiler

The taskHPF compiler (Figure 3) is logically split in two parts: the *front-end* and the *code generator*. The *front-end* is fairly standard. It parses the source code, checks types and produces the internal program representation, the *construct tree*. The construct tree is an annotated syntax tree in which each internal node is a pipeline call, and each leaf node is an HPF task call. Annotations record information on directives, types, module names and names of the files in which the HPF code has been saved for subsequent plugging in the wrapper templates.

Code generation works on the construct tree adding further annotations to the nodes. It gives unique identifiers to HPF tasks and to channels between tasks, which are needed by the $COLT_{\texttt{HPF}}$ library, and selects a task wrapper for each task HPF node. The task wrappers have unique names with which they can be located in the task wrapper library.

Tasks wrappers are stored as text files in which immutable HPF code and $COLT_{\texttt{HPF}}$ calls are interspersed with parts to be instantiated/changed.

4.2 $COLT_{\texttt{HPF}}$

The efficient implementation of the primitives to exchange arrays are the key issue to be addressed in taskHPF support. This is delegated to the $COLT_{\texttt{HPF}}$ library primitives which provide native optimized communication among distributed HPF tasks.

Communicating distributed data between data-parallel tasks entails in fact making several point-to-point communications (see Figure 4). Moreover, since the actual boundaries of the array sections allocated on each processor cannot be statically known, all the work must be done during execution. At run time, $COLT_{\texttt{HPF}}$ inspects the HPF support to find out on both the sender and receiver sides the actual mapping of any distributed array exchanged[4]. Then, all processes involved in the communication compute the intersections of their own array partitions with the ones of the processes belonging to the partner task. To this end, $COLT_{\texttt{HPF}}$ uses Ramaswamy and Banerjee's *pitfalls* algorithm [14]. A global *communication schedule* is thus derived. It establishes, for each sender process, the portion of array sections 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 it has to receive from any of the sender processes. 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*. These features help in implementing pipeline patterns, since transmitting a homogeneous data stream entails repeating the same communications several times.

[4]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.

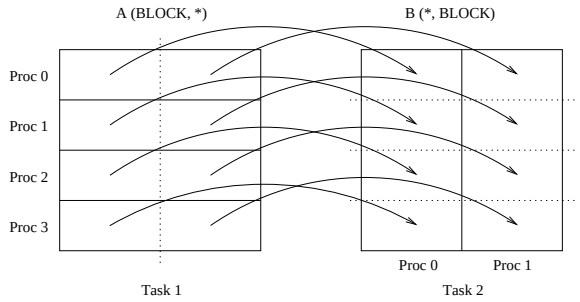


Figure 4: Point-to-point communications to send a two-dimensional array from one HPF task, mapped on 4 processes, to another HPF task, mapped on 2 processes. Data distributions on the sender and the receiver tasks are `(BLOCK,*)` and `(*,BLOCK)`, respectively.

5 Conclusions

We have given an overview of taskHPF and described its compilation system. In recent times, many models proposing the integration of task and data parallelism have been proposed [2, 10, 15–17, 19, 20]. Differently from other proposals [11, 19], our approach does not need the introduction of new task parallel HPF constructs, and programs can be compiled using off-the-shelf HPF compilers. Moreover, we do not need compiler analysis to derive the task interaction structure (pattern) exploited. In our opinion, taskHPF is easier and more intuitive to use with respect to the extension of HPF due to its declarative nature, and to the fact that different HPF modules are well separated and do not need to be changed to define a different coordination structure for the same application. Moreover, patterns provide the compiler with precise information on the intended use of a given HPF code. This information can be used to optimize mapping and scheduling with limited amount of static analysis and profiling information [5, 18].

Systems based on patterns or skeletons have been proposed by different researchers [1, 6, 16]. However, to our knowledge our proposal is the first entirely based on HPF. This allows us to take advantage of the large body of research and compiler technology developed in the HPF community.

We currently have a taskHPF prototype compiler that implements task and pipeline patterns and their composition. We intend to improve this prototype and to investigate our approach on the test-bed of larger real world applications. In the mean time, we are working on the model side to add patterns to the coordination language in order to enlarge the class of applications that can be expressed. In particular, we are thinking of adding a *dag* pattern, modeling directed acyclic graphs, and *loops* to model repeatedly executed dags.

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] H. E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, July 1998.

[3] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

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

[5] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. Optimal Processor Assignment for a Class of Pipeline Computations. *IEEE Trans. on Paral. and Distr. Systems*, 5(4):439–445, Apr. 1994.

[6] J. Darlington, Y. Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proc. of the 15th ACM Symp. on Princ. and Pract. of Paral. Progr.*, 1995.

[7] P. Dinda *et al.* The CMU task parallel program suite. Tech. Rep. CMU-CS-94-131, School of Computer Science, CMU, 1994.

[8] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *J. of Paral. and Distr. Comp.*, 45(2):148–158, Sept. 1997.

[9] I. T. Foster and K. M. Chandi. Fortran M: a language for modular parallel programming. *J. of Paral. and Distr. Comp.*, 26(1):24–35, Apr. 1995.

[10] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Paral. and Distr. Technology*, 2(2):16–26, 1994.

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

[12] S. Orlando, P. Palmerini and R. Perego. Mixed Data and Task Parallelism with HPF and PVM. *CLUSTER COMPUTING: The J. of Networks, Software and Appl.*, Baltzer Science Publishers, in press, 2000.

[13] 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.

[14] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proc. of Frontiers '95*, pages 342–349, Feb. 1995.

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

[16] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *Proc. of ACM SAC'99*, pages 146–155, San Antonio, Texas, Feb 1999.

[17] D. B. Skillicorn. The network of tasks model. In *Proc. of IASTED Paral. Distr. Comp. and Systems*, Boston, MA, Nov. 1999. To appear.

[18] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proc. 8th SPAA*, June 1996.

[19] J. Subholk and B. Yang. A new model for integrated nested task and data parallel programming. In *Proc. of the 6th ACM Symp. on Princ. and Pract. of Paral. Progr.*, pages 1–12, 1997.

[20] T. Brandes. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. In *Proc. of Euro-Par'99*, Toulouse, France, LNCS 1685, pages 833-845, Sept. 1999.