Integrating task and data parallelism with taskHPF

S. Ciarpaglini†, L. Folchi†, S. Orlando‡, S. Pelagatti†, and R. Perego*

† Dipartimento di Informatica, Università di Pisa, Pisa, Italy

‡ Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy

* Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), 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*). We also assume that HPF tasks interact according to a static and predictable structure. We describe taskHPF a coordination language in which the user can define the HPF tasks with a clean input and output interface, and program the global interaction structure among 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.

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 the related data set. In task parallelism, distinct parts of the program proceed independently on generally distinct sets of processors.

It is commonly accepted that a number of applications can benefit from exploiting both kinds of parallelism. For instance, 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, 20, 21]. In these applications data parallel modules usually do not scale beyond a given threshold due to the limited size of the data involved. In this case, using different sets of processors to compute independently different elements of the input stream can considerably increase the system throughput [21]. 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, 16, 21–23]. In the general case, integration of task and data parallel models is rather difficult as these models tend to enforce different programming and implementation styles [2]. However, most of the applications benefiting from a mixed style shows a fairly static and predictable structure [7, 10, 18] and works on a stream of independent data sets. Moreover, in many applications data parallelism is present at the inner level, inside modules which are then glued together using task parallelism [17, 22].

These features can be exploited to derive a mixed model which, even if not solving the interaction between task and data parallelism in the most general sense, puts the basis for an environment oriented to a (potentially) large class of target applications. The paper takes this pragmatic view, and addresses the problem of expressing in an efficient way a task parallel interaction structure for data parallel tasks expressed in HPF and working on data streams.

Our approach consists in providing a high level coordination language to compose together a set of HPF programs, called *HPF tasks*, computing homogeneous streams of data. An HPF task is defined by supplying its computational code along with an input and output interface. HPF 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 patterns in the class of applications at hand. For example, our coordination language 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 allows us to fix the number of processors assigned to an HPF task, devoting more resources to critical stages. A REPLICATE directive allows us 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. Then, 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 $COLT_{\mathtt{HPF}}$, a library which provides mechanisms for loading the HPF tasks on set of processors and for optimized inter-task communications [13, 14].

The main advantage of this approach is to supply programmers with a concise, portable and high-level declarative way to describe the task parallel interaction of their HPF modules. This shortens the development time and allows several structures to be easily tested. Moreover, the explicit declaration of the coordination pattern allowed us to devise a simple and effective compiler which exploits specialized "wrappers" to enclose the user-supplied code. The static knowledge of the application structure also permits effective mapping techniques to be applied [5, 20]. Finally, differently from other approaches for mixing task and HPF data parallelism, we do not need the adoption of new task parallel HPF constructs [11]. In fact, we compile each HPF tasks using an off-the-shelf HPF compiler[1], while the task parallel coordination level is provided by the portable $COLT_{\mathtt{HPF}}$ library [12].

The remainder of the paper is organized as follows. Section 2 gives an overview of the approach, details taskHPF constructs, and applies the approach to a simple example discussing some results. Section 3 describes the taskHPF compilation system and discusses implementation issues. Section 4 discusses related work and draws some conclusions.

2 taskHPF overview

taskHPF uses a two-tier model for combining task and data parallelism. A set of data parallel tasks, expressed in HPF, are coordinated using a restricted set of task parallel coordination patterns (skeletons). Only two patterns are implemented in the current taskHPF compiler prototype: *task* and *pipeline*. The *task* pattern is a "wrapper" which defines HPF tasks. The *pipeline* pattern allows HPF tasks to be pipelined on a stream of input data. Pipeline instances can be nested together and pipeline stages can be replicated to build more complex (and possibly balanced) structures.

Other patterns under study and not yet implemented in the prototype compiler are *dag* and *loop*. The former models *directed acyclic graphs*, where nodes are HPF tasks and edges represent data flow dependencies. The latter models *dag*s which have to be repeatedly executed, and is useful, for example, to program iterative methods.

2.1 The task pattern

The task pattern defines an HPF task with well-defined input/output behavior. A task declaration is as follows.

```
TASK name IN(...) OUT(...)
STARTUP ${
C   Local Vars declaration + HPF Distribution directives
C   (of both Local and I/O List Vars)
C   HPF code executed before starting computation
$}
BODY ${
C   HPF code consuming Input Vars and producing Output Vars
$}
ENDUP ${
C   HPF code executed before exiting
$}
ENDTASK
```

where TASK and ENDTASK are delimiters and name is the name of the module. IN(...) and OUT(...) enclose the F90 declaration of the lists of input and output variables, respectively. Brackets ${..}$ enclose legal HPF code. The scope of input/output variables is all the HPF code in the TASK definition.

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

The user HPF code is enclosed within three sections: STARTUP (*startup code*), BODY (*body code*), and ENDUP (*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 executed before terminating the task. Typically this part specifies how the computation results must be saved on file or any action the task should do before exiting.

A task call is of the form

```
name IN(...) OUT(...) ON PROCESSORS(N)
```

where the input and output lists are instantiated with the actuals, while the optional ON PROCESSORS directive gives the number of processors to be used for executing the parallel HPF task.

2.2 The pipeline pattern

The pipeline pattern "pipelines" a sequence of HPF tasks. It implicitly states the existence of a data stream. When a pipeline pattern is used, the body code of each task is executed for each element of the input stream. The syntactic layout for a pipeline pattern is as follows:

```
PIPE name IN(...) OUT(...)
  <module_call>
  ...
  <module_call>
ENDPIPE
```

Again PIPE and ENDPIPE delimits the module declaration, where name is the module name. The body of a pipeline instance is given by a sequence of calls to taskHPF modules. The ith call defines the ith pipeline stage. The input variables of the ith stage must match the output of stage $i-1$. This constraint will be relaxed in future versions of the compiler to allow general dependencies among stages to be expressed. Here is a simple two stage pipeline:

```
PIPE two_stages  IN(REAL a(100,100)) OUT(REAL c(100,100), REAL d(200))
  stage1 IN(a) OUT(REAL b(100,100))
  stage2 IN(b) OUT(c,d)
ENDPIPE
```

Note that each stage can only name as input a variable produced by the previous stage. A call of a stage i also declares its output variables, thus introducing new *single assignment* variables, which can be modified by stage i only. The first and last stage can be regarded as special cases, since the input of the first and the output of the last have to match the input and the output of the whole pipeline pattern, respectively.

Replication. When allowed by the semantics of stream computation, the throughput of a pipeline can be improved by replicating a non-scalable stage instead of devoting more processors to its data parallel implementation [19, 20]. In this case, we have multiple equivalent copies of the stage and each input item is scheduled to one of the copies which carries out its computation independently of the others. The implemented scheduling policy ensures the load balancing among different copies.

Any taskHPF module can be replicated by adding a REPLICATE directive immediately before the call. An user can also specify how many times a module should be replicated. In the following example, the third module of the pipe ts-pipe is replicated 3 times.

```
PIPE ts-pipe IN(...) OUT(...)
   module_call1
   module_call2
   REPLICATE(3) module_call_3
   module_call4
ENDPIPE
```

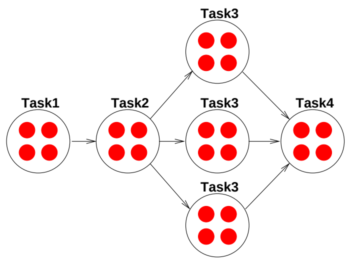


Figure 1: Structure of the parallel computation defined by `ts-pipe`.

2.3 Execution model

A taskHPF program defines a network of cooperating HPF tasks. Each task is assigned to a disjoint set of processors. Fig. 1 shows an abstract representation of the interacting task network defined by the `ts-pipe` program sketched above, where the called modules are instances of the task pattern.

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 of the sink module, 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 on 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 (Task2), it must deliver the result to only one copy of the replicated next stage (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 (Task4), it can receive input data from any of the copies of the previous stage (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 dispathcer and combine both behaviors described above.

This execution scheme works also for nested pipeline patterns.

2.4 A simple case study and some results

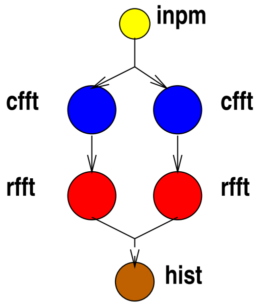
In order to illustrate the potentiality of taskHPF, we use a small kernel that was already presented in literature [10, 22], and represents the structure of a large class of image and signal processing applications. The kernel application is *FFT-Hist*, where a stream of two–dimensional COMPLEX matrixes 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 forwards 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*. For the sake of brevity, 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`. Since the two tasks scale quite well [10, 22], we map each of them on 4 processors.

```
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
```

(b)



(a)

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

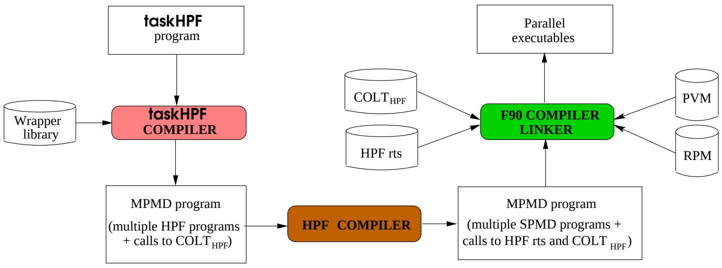


Figure 3: Structure of the taskHPF compilation system.

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. 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 taskHPF program was 1.12 times faster.

3 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 HPF programs corresponding to the execution model of Sec. 2.3. 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 whether the task is a dispatcher, a collector, a source, a sink or a simple pipe stage. 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, HPF code (endup, body and startup), etc.

Task wrappers are defined using HPF with calls to $COLT_{\mathtt{HPF}}$ library [12–14], and have special markers for the parts to be plugged in the instantiation phase. $COLT_{\mathtt{HPF}}$ 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 SPMD f90/f77 parallel programs, interacting by means of $COLT_{\mathtt{HPF}}$. All these programs are then passed to an HPF compiler and linked to relevant libraries to be translated in executable code. Note that also the low-level communication middlewares used by $COLT_{\mathtt{HPF}}$ and by the HPF program must be linked. For instance, the `pghpf` compiler we use exploits RPM, the PGI Real Parallel Machine system, while $COLT_{\mathtt{HPF}}$ uses PVM.

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

3.1 The taskHPF compiler

Figure 3 shows how the taskHPF compiler interacts with the HPF compiler and the libraries needed to generate the actual taskHPF executable files.

The taskHPF compiler 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 $COLT_{\mathtt{HPF}}$ library, and selects a task wrapper for each task HPF node. The task wrappers are of different kind depending on the role of the task (dispatcher, collector, sink...) and have unique names (e.g., `pipe_sink`) with which they can be located in the task wrapper library.

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

- the calls to the $COLT_{\mathtt{HPF}}$ library needed to define the communication channels and to transmit distributed data. This code depends on the task input and output variables and on task identifiers.
- the loop which activates the body code on each stream element. When an `END_OF_STREAM` mark is read, the loop is ended, termination propagated on the output channels, and the endup code executed.
- a configuration file which will be used by the $COLT_{\mathtt{HPF}}$ library to spawn the different tasks.

3.2 $COLT_{\mathtt{HPF}}$

The efficient implementation of the primitives to exchange arrays are the key issue to be addressed in taskHPF support. This is delegated to the $COLT_{\mathtt{HPF}}$ 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, $COLT_{\mathtt{HPF}}$

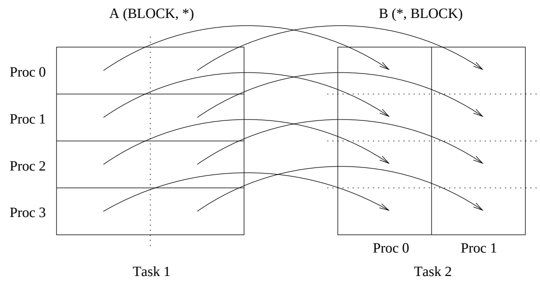


Figure 4: Point-to-point communications to send a two-dimensional array from one HPF task, mapped on 4 processes, to another HPF task, mapped on 2 processes. Data distributions on the sender and the receiver tasks are `(BLOCK,*)` and `(*,BLOCK)`, respectively.

inspects the HPF support to find out on both the sender and receiver sides the actual mapping of any distributed array exchanged[2]. 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 [15]. 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.

Another important implementation choice regards the low-level communication middleware used for point-to-point communications. $COLT_{\texttt{HPF}}$ runs on both PVM and MPI. For the taskHPF compiler, we chose the PVM version mainly because it supports a dynamic task model inherited from PVM. This allows taskHPF tasks, which are compiled as separate executables, to enroll themselves in the PVM virtual machine at run-time and thus to participate in the mixed task and data parallel application. The execution of the various HPF tasks is started by a specific $COLT_{\texttt{HPF}}$ daemon, called the *colt daemon*. As soon as it is started, the colt daemon reads a *configuration file* produced by the taskHPF compiler which, for each task to be started, stores the following: *(a)* the name of the associated HPF *executable module*, *(b)* the degree of parallelism for the task, *(c)* the list of PVM virtual processors onto which the task has to be executed, and *(d)* a unique identifier (called the `colt_tid`). Using the mechanisms provided by the OS and the specific HPF compilation system, the colt daemon starts the various HPF tasks specified in the configuration file. As soon as a spawned HPF task starts executing, all the processes running the task (i.e., the processes participating in the corresponding SPMD subprogram generated by the HPF compiler) call the $COLT_{\texttt{HPF}}$ initialization primitive to make all the processes running the task enroll in PVM and receive their PVM identifiers (hereafter `pvm_tids`). `pvm_tids` are used within $COLT_{\texttt{HPF}}$ to exchange PVM point-to-point messages among the processes running the various cooperating tasks. Thus, $COLT_{\texttt{HPF}}$ has to know the correspondence between the various HPF tasks, each one identified by a distinct `colt_tid`, and the sets of `pvm_tids` corresponding to the groups of processes running the tasks themselves. This information is collected by the colt daemon and broadcast to all the PVM processes running the application.

4 Conclusions & related work

In recent times, many models proposing the integration of task and data parallelism have been proposed [2, 10, 16–18, 22, 23].

The attempt closest to our approach is Fx [10], which is designed as an extension to HPF. Fx allows the definitions of a flat ensemble of tasks in task parallel sections and relies on compiler time analysis to have a pipelined execution among independent parts of the same task section. Stages with a low throughput can

[2] 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.

be replicated by users, but a simple round robin scheduling strategy is adopted to dispatch stream items to the copies. HPF 2.0 introduces task parallelism using `TASK_REGION` [11] and was inspired by a more recent work by the Fx group [22]. Inter-task communications are expressed by means of simple, deterministic, global assignments. Note that this proposal maintains the adherence to the general philosophy followed by HPF: an HPF 2.0 program can always be compiled with a Fortran 90 compiler that interprets HPF directives as simple comments.

Our approach, instead, 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 a compiler with precise information on the intended use of a given HPF code and can be used to optimize mapping and scheduling with limited amount of static analysis and profiling information. For example, optimal mapping algorithms for pipeline structures composed of data parallel stages [5, 20] can be profitably used. When an acyclic task graph pattern is used, effective heuristic techniques can be adopted, as mapping dag graphs is generally NP-complete [16, 18].

A different approach to integrate task and data parallelism regards the design of an HPF binding for a standard message-passing library such as MPI [8]. Low-level message passing primitives are provided which allow the exchange of distributed data among concurrent HPF tasks. We believe that the exploitation of a message-passing interface is too much low-level for effective usability. Other approaches extend an object-oriented coordination languages with data parallelism (e.g., Opus [4]). These models show a greater dynamism with respect to taskHPF and for this reason delegate much of the optimization effort to the run time. In addition, they do not support the use of common organization patterns.

Finally, patterns have been proposed to coordinate data parallel modules in different proposal [1, 6, 17]. 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 dags repeatedly executed.

Acknowledgments

This project has been partially founded by HPCC/SEA contract number EU1063. L. Folchi is founded by INFN under grant 6484/97.

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 Transactions 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 Transactions on Parallel and Distributed 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[7] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subholk, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, 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. *Journal of Parallel and Distributed Computing*, 45(2):148–158, Sept. 1997.

[9] I. T. Foster and K. M. Chandi. Fortran M: a language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, Apr. 1995.

[10] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*, 2(2):16–26, 1994.

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

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

[13] S. Orlando, P. Palmerini and R. Perego. Coordinating HPF programs to mix task and data parallelism. To appear in Proceedings of the 2000 ACM Symposium on Applied Computing (SAC'2000) - Special Track on Coordination Models, Languages and Application, Villa Olmo, Como, Italy, March 19-21, 2000.

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

[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] 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.

[18] D. B. Skillicorn. The network of tasks model. In *Proc. of IASTED Parallel Distributed Computing and Systems*, Boston, MA, Nov. 1999. To appear.

[19] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 134–143, Santa Barbara, CA, July 1995.

[20] 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.

[21] J. Subholk, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism in a multicomputer. In *Proc. of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, CA, May 1993.

[22] J. Subholk and B. Yang. A new model for integrated nested task and data parallel programming. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1997.

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