A Coordination Layer for Exploiting Task Parallelism with HPF

Salvatore Orlando[1] and Raffaele Perego[2]

[1] Dip. di Matematica Appl. ed Informatica, Università Ca' Foscari di Venezia, Italy
[2] Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy

Abstract. This paper introduces $COLT_{\texttt{HPF}}$, a run–time support for exploiting task parallelism within HPF programs, which can be employed by a compiler of a high-level coordination language to structure a set of data-parallel HPF tasks according to popular paradigms of *task-parallelism*. We use $COLT_{\texttt{HPF}}$ to program a computer vision application and report the results obtained by running the application on an SGI/Cray T3E.

1 Introduction

Although HPF-1 [8] permits programmers to express data-parallel computations in a portable, high-level way, it is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data-parallel paradigm. The promising possibility of exploiting a mixture of task and data parallelism, where data parallelism is restricted within HPF tasks and task parallelism is achieved by their concurrent execution, has recently received much attention [6, 5]. Depending on the applications, HPF tasks can be organized according to patterns which are structured to varying degrees. For example, applications may be modeled by a fixed but unstructured task dependency graph, where edges correspond to data-flow dependencies. However, it is more common for parallel applications to process streams of input data, so that a more regular *pipeline* task structure can be exploited [11]. When the bandwidth of a given pipeline stage has to be increased, it is often better to replicate it rather than using several processors for its data-parallel implementation. Replication entails using a *processor farm* structure [7], where incoming jobs are dispatched on one of the stage replicas by adopting either a simple round-robin or a dynamic self-scheduling policy.

This paper presents $COLT_{\texttt{HPF}}$ (COordination Layer for Tasks expressed in HPF), a coordination/communication layer for HPF tasks. $COLT_{\texttt{HPF}}$ provides suitable mechanisms for starting distinct HPF data-parallel tasks on disjoint groups of processors, along with optimized primitives for inter-task communication where data to be exchanged may be distributed among the processors according to user-specified HPF directives. We discuss how $COLT_{\texttt{HPF}}$ can be used to structure data-parallel computations that cooperate according to popular forms of task parallelism like pipelines and processor farms [11, 7]. We present

templates which implement these forms of task parallelism and we discuss the exploitation of these templates to design a structured, high-level, coordination language. We claim that the use of such a language simplifies program development and restructuring, while effective automatic optimizations (mapping, choice of the degree of parallelism for each task, program transformations) can be easily devised because the specific structure of the parallel programs is restricted and statically known. Unfortunately, this approach requires a new compiler in addition to HPF, though the templates proposed can also be exploited to design libraries of *skeletons* [4, 1]. However, the compiler is very simple, though its complexity may increase depending on the level of optimization supported.

2 Task-parallel structures to coordinate HPF tasks

To describe the features of $COLT_{\texttt{HPF}}$ we refer to a particular class of applications which exploit a mixture of task and data parallelism. More specifically, we focus on applications which transform an input data stream into an output one of the same length. As regards the structures of task parallelism used to coordinate

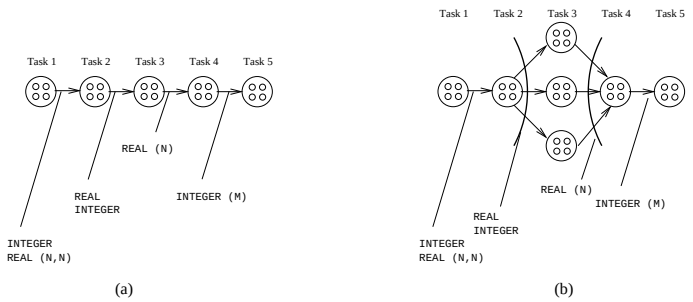

Fig. 1. Two examples of the same application implemented (a) by means of a pipeline, and (b) by hierarchically composing the same pipeline with a processor farm.

these HPF tasks, we focus on a few forms of parallelism, namely *pipelines* and *processor farms*. Figure 1.(a) shows the structure of an application obtained by composing five data-parallel tasks according to a *pipeline* structure, where the first and the last stages of the pipeline only produce and consume, respectively, the data stream. The data types of the input/output channels connecting each pair of interacting tasks are also shown. Figure 1.(b) shows the same application where Task 3 has been replicated. In this case, besides computing their own job by transforming their input data stream into the output one, Task 2 has to dispatch the various elements of the output stream to the three replicas of Task 3, while Task 4 has to collect the elements received from any of the three replicas.

Each form of parallelism can be associated with an *implementation template*. A template can be considered as the code scheme of a set of communicating HPF

tasks which cooperate according to a fixed interaction pattern. In order to obtain the actual implementation of a user application, the template corresponding to the chosen parallel structure must be instantiated by inserting the user-provided code, as well as the correct calls to the $COLT_{\texttt{HPF}}$ primitives to initialize channels and to exchange data between tasks.

Since most of the code production needed to instantiate a template can be automated, we believe that the best usage of $COLT_{\texttt{HPF}}$ is through a simple high-level coordination language. Roughly speaking, the associated compiler should in this case be responsible for instantiating templates for users. To use such a coordination language a programmer should only be required to provide the HPF code of each task, its parameter lists to specify the types of the elements composing the input and the output streams, and finally the language constructs to express how these tasks must be coordinated, e.g. according to a pipeline or a processor farm form of parallelism. A simple coordination language to express this kind of structured parallel programming strategy, P^3L, has already been proposed elsewhere [1], even if the host language adopted was sequential (C) rather than parallel (HPF). For example, a P^3L-like code to express the structure represented in Figure 1.(b) would be:

```
task_3 in(INTEGER a, REAL b) out(REAL c(N,N))
  hpf_distribution(DISTRIBUTE C(BLOCK, *))
  hpf_code_init(<initialiaze the task status>)
  hpf_code(<use a and b, compute, and produce c>)
end


farm foo in(INTEGER a, REAL b) out(REAL c(N,N))
  task_3 in(a, b) out(c)
end farm
```

```
pipe in() out()
  task_1 in()    out(INTEGER a, REAL b(N,N))
  task_2 in(a,b) out(INTEGER c, REAL d)
  foo    in(c,d) out(REAL e(N,N))
  task_4 in(e)   out(INTEGER f(M))
  task_5 in(f)   out()
end pipe
```

Note the definition of Task 3, with the relative input and the output lists of parameters, the specification of the layout for distributed parameters, and the HPF user code. Since Task 3 must be replicated, a *farm* named `foo` is thus defined, whose identical workers are replicas of Task 3. Finally, the farm must be composed with the other tasks to obtain the final *pipeline* structure of Figure 1.(b)[1]. Note the hierarchical composition of the task-parallel constructs: there is a `pipe`, which invokes a `farm`, which, in turn, invokes a simple HPF data-parallel task. The specification of the structure of the application is concise, simple, and high-level. Moreover, by only modifying this high-level description we can radically change the parallel structure of the application to test alternative implementations.
The code shown does not specify the number of processors to be exploited by each task, nor the number of workers of the farm (e.g. the number of replicas of Task 3). Suitable directives could be provided, so that a programmer can tune these parameters to optimize performance (*performance debugging*), although, since we are concentrating on a set of restricted and structured forms of parallelism, the compiler could use suitable performance models, profiling information

[1] For the sake of brevity, the definition of the other tasks of the `pipe` is not reported.

and also architectural constraints (e.g. the number of available processors) to optimize resource allocation [1, 11].

3 *COLT*$_{\texttt{HPF}}$ implementation

The current implementation of *COLT*$_{\texttt{HPF}}$ [9] is bound with an HPF compiler, Adaptor [2], which has been configured to exploit MPI. We believe that our technique is very general, so that a similar binding might easily be made with other HPF compilers that use MPI too. The binding is based on a simple modification of the Adaptor run-time support, so that each HPF task exploits a different MPI communicator. For each disjoint processor group on which the various HPF tasks have to be mapped, we create a distinct communicator, namely `MPI_LOCAL_COMM_WORLD`, by using the `MPI_Comm_split` primitive. To this end, a configuration file must be provided to define the processor groups and the mapping of the various HPF tasks. Thus, while `MPI_LOCAL_COMM_WORLD` is now used for all Adaptor-related MPI calls within each HPF task, the default communicator `MPI_COMM_WORLD` is used for intertask communications implemented by *COLT*$_{\texttt{HPF}}$.

Communicating distributed data between a pair of HPF tasks may involve all the processors of the two corresponding groups. Moreover, when data and processor layouts of the sender and receiver tasks differ, it also entails the redistribution of the data exchanged. Since many of these inter-task communications may need to be repeated due to the presence of an input/output stream, *COLT*$_{\texttt{HPF}}$ provides primitives to establish *persistent typed channels* between tasks. These channels, on the basis of the knowledge about data distributions on the sender and the receiver processor groups, thus store the *Communication Schedule*, which is used by the send/receive primitives for packing/unpacking data and for carrying out the "minimal" number of point-to-point communications between the processors of the two groups. To open a channel, both the sender and receiver have to inquire the HPF run-time support to find out the corresponding array data distributions. This information is then exchanged, and, by using Ramaswamy and Banerjee's *pitfalls* redistribution algorithm [10], each processor derives its *Communication Schedule*. *COLT*$_{\texttt{HPF}}$ also supplies primitives to exchange scalar data between processor groups, where these data are replicated on all the processors of the groups.
Finally, *COLT*$_{\texttt{HPF}}$ provides primitives to signal simple events between tasks, where the reception of messages may be carried out *non-deterministically*. These signals are useful, for example, to implement processor farms that adopt a dynamic self scheduling policy to dispatch jobs. According to this policy, the farm dispatcher, e.g. Task 2 in Figure 1.(b), receives *ready* signals from any of the various farm workers, where these signals state the completion of a job dispatched beforehand and a request for further jobs.

COLT$_{\texttt{HPF}}$ primitives are implemented as `HPF_LOCAL EXTRINSIC` subroutines. When an `EXTRINSIC` subroutine is invoked by an HPF task, all the processors executing the task switch from the single thread of control provided by HPF to an

SPMD style of execution. According to the HPF language definition, `HPF_LOCAL` subroutines have to be written in a restricted HPF language where, for example, it is not possible to transparently access data stored on remote processors, but each processor can only refer its own section of any distributed array.
The techniques adopted in the implementation of $COLT_{\mathtt{HPF}}$ are similar to those exploited by Foster et al. to design their HPF binding for MPI [5]. In [9] we survey their work and other related ones.

4 Template examples

In this section we exemplify the concept of *implementation template* by illustrating the *task template* of a generic pipeline stage, and its instantiation starting from a $P^3L - like$ high level specification of the stage.

A pipeline stage is an HPF task which cyclically reads an element of the input stream and produces a corresponding output element, where an incremental mark is associated with each element of the stream. The transmission of each stream element is thus preceded by the transmission of the related mark. The end of the stream is identified by a particular `END_OF_STREAM` mark.

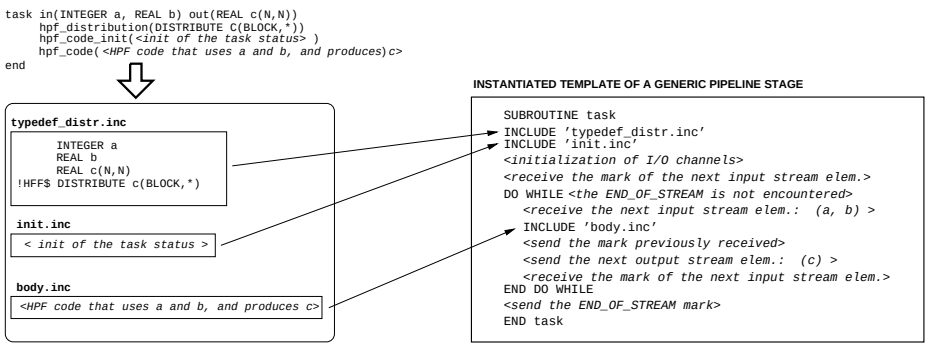

Fig. 2. A *task template* of a pipeline stage, where its instantiation is shown starting from a specific construct of a high-level coordination language.

Figure 2 shows the process template of a pipeline stage and its instantiation. As can be seen, the input/output lists of data, along with their distribution directives, are used by the compiler to generate an include file `typedef_distr.inc`. Moreover, the declaration of the task local variables, along with the relative code for their initialization is included in another file, `init.inc`. Finally, the code to be executed to consume/produce each data stream element is contained in the include file `body.inc`. These files are directly included in the source code of the template which is also shown in the figure. To complete the instantiation of the template, the appropriate calls to the $COLT_{\mathtt{HPF}}$ layer which initialize the input/output channels and send/receive the elements of the input/output stream,

also have to be generated and included. The correct generation of these calls relies on the knowledge of the task input/output lists, as well as the mapping of the tasks onto the disjoint groups of processors.

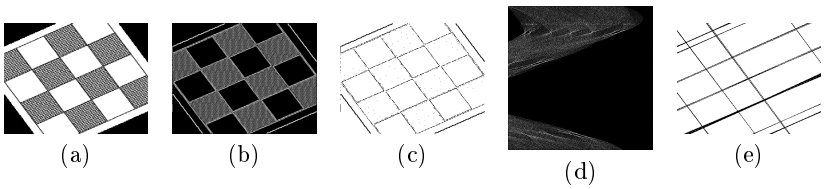

Fig. 3. Example of the input/output images produced by the various stages of the computer vision application: (a)⇒(b): Sobel filter stage for edge enhancement – (b)⇒(c): Thresholding stage to produce a bit map – (c)⇒(d): Hough transform stage to detect straight lines – (d)⇒(e): de-Hough transform stage to plot the most voted straight lines.

5 Experimental results

To show the effectiveness of our approach we used $COLT_{\texttt{HPF}}$ to implement a complete high-level computer vision application which detects in each input image the straight lines that best fit the edges of the objects represented in the image itself. For each grey-scale image received in input (for example, see Figure 3.(a)), the application enhances the edges of the objects contained in the image, detects the straight lines lying on these edges, and finally builds a new image containing only the most evident lines identified at the previous step. The application can be naturally expressed according to a three stage pipeline structure. The first stage reads from the file system each image, and applies a low-level Sobel filter to enhance the image edges. Since the produced image (see Figure 3.(b)) is still a grey-scale one, it has to be transformed into a black-white bitmap (see Figure 3.(c)) to be processed by the following stage. Thus a thresholding filter is also applied by the first stage before sending the resulting bitmap to the next stage. The second stage performs a Hough transform, a high-level vision algorithm which tries to identify in the image specific patterns (in this case straight lines) from their analytical representation (in this case the equations of the straight lines). The output of the Hough transformation is a matrix of accumulators $H(\rho, \theta)$, each element of which represents the number of black pixels whose spatial coordinates (x, y) satisfy the equation $\rho = x \cos\theta + y \sin\theta$. Matrix H can be interpreted as a grey-scale image (see Figure 3.(d)), where lighter pixels correspond to the most "voted for" straight lines. Finally, the third stage chooses the most voted for lines, and produces an image where only these lines are displayed. The resulting image (see Figure 3.(e)) is then written to an output file.

Table 1. Computation and I/O times (in secs) for the HPF implementation of the three stages of the pipeline.

	Sobel&Thresh			Hough	de-Hough		
Procs	I/O	Comp.	Total	Comp.	I/O	Comp.	Total
1	9.6	11.9	21.5	148.3	1.0	21.4	22.4
2	10.0	5.8	15.8	78.0	1.2	17.3	18.5
4	10.2	2.4	12.6	43.5	1.3	13.7	15.0
8	10.4	0.9	11.3	24.7	1.3	12.3	13.6
16	10.5	0.7	11.2	15.9	1.3	11.8	13.1
32	11.6	0.7	12.3	12.3	1.4	11.6	13.0

Table 1 illustrates some results of experiments conducted on an SGI/Cray T3E. It shows the completion times of each of the three stages, where the input stream is composed of 60 256×256 images. Note that the I/O times of the first and the third stage do not scale with the number of processors used. If the total completion times reported in the table is considered, it is clear that it is no point exploiting more than 4/8 processors for these stages. On the other hand, the Hough transform stage scales better. We can thus assign enough processors to the second stage so that its bandwidth becomes equal to that of the other stages. For example, if we use 2 processors for the first stage, we should use 4 processors for the third stage, and 16 for the second one to optimize the throughput of the pipeline. Alternatively, since the costs of the Hough transform algorithm very much depends on the input data [3], we may decide to exploit a processor farm for the implementation of the second stage. For example, a farm with two replicated workers, where the bandwidth of each worker is half the bandwidth of the first and the last stages, allows the overall pipeline throughput to be optimized, provided that a dynamic self scheduling policy is implemented to balance the workers' workloads.

Table 2 shows the execution times and the speedups measured on a Cray T3E executing our computer vision application, where we adopted a self-scheduling processor farm for the second stage of the pipeline. The column labeled *Structure* in the table, indicates the mapping used for the $COLT_{\texttt{HPF}}$ implementations. For example, (4 (8,8) 4) means that 4 processors were used for both the first and last stage of the pipeline, while each one of the two farm workers was run on 8 processors. The table also compares the results obtained by the $COLT_{\texttt{HPF}}$ implementations with those obtained by pure HPF implementations exploiting the same number of processors. 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 60% to 160%.

6 Conclusions

In this paper we have discussed $COLT_{\texttt{HPF}}$, a run-time support to coordinate HPF tasks. We have shown how $COLT_{\texttt{HPF}}$ can be exploited to design implementation templates for common forms of parallelism, and how these templates can be used

Table 2. Comparison of execution times (in seconds) obtained with the HPF and $COLT_{\texttt{HPF}}$ implementation of the computer vision application.

	$COLT_{\texttt{HPF}}$			HPF		HPF/$COLT_{\texttt{HPF}}$ ratio
Procs	*Structure*	Exec. Time	Speedup	Exec. Time	Speedup	
8	1 (3,3) 1	30.7	6.26	49.7	3.87	1.6
16	2 (4,4,4) 2	19.9	9.66	40.3	4.77	2.0
24	4 (8,8) 4	15.8	12.14	38.2	5.03	2.4
32	8 (8,8,8) 8	14.4	13.34	37.6	5.11	2.6

by a compiler of a structured, high-level coordination language. We have also presented some encouraging experimental results, conducted on an SGI/Cray T3E, where pipeline and farm templates have been instantiated to implement a complete computer vision application.

Acknowledgments Our greatest thanks are for Thomas Brandes, for many valuable discussions about task parallelism and Adaptor. We also wish to thank Ovidio Salvetti for his suggestions about the computer vision application and the CINECA Consortium of Bologna for the use of the SGI/Cray T3E.

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: P&E*, 7(3):225–255, 1995, Wiley.
2. T. Brandes. ADAPTOR Programmer's Guide Version 5.0. Internal Report Adaptor 3, GMD-SCAI, Sankt Augustin, Germany, April 97.
3. S. C. Orphanoudakis D. Gerogiannis. Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks. *IEEE TPDS*, 4(9), Sept. 1993.
4. J. Darlington et al. Parallel Programming Using Skeleton Functions. In *Proc. of PARLE '93*, pages 146–160, Munich, Germany, June 1993. LNCS 694, Spinger-Verlag.
5. Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *JPDC*, 45(2):148–158, Sept. 1997, Academic Press.
6. T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distr. Technology*, 2(2):16–26, 1994.
7. A.J.G. Hey. Experiments in MIMD Parallelism. In *Proc. of PARLE '89*, pages 28–42, Eindhoven, The Netherlands, June 1989. LNCS 366 Spinger-Verlag.
8. C.H. Koebel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.
9. S. Orlando and R. Perego. $COLT_{\texttt{HPF}}$, a Coordination Layer for HPF Tasks. Technical Report TR-4/98, Dip. di Mat. Appl. ed Informatica, Università di Venezia, March 1998. Available at `http://raffaele.cnuce.cnr.it//papers.html`.
10. S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Paral. Comp.*, pages 342–349, Feb. 1995.
11. J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proc. of 8th Annual ACM SPAA*, June 1996.