An MPI–Based Run–Time Support to Coordinate HPF Tasks

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 describes $COLT_{\texttt{HPF}}$, an MPI–based run–time support for the coordination of concurrent and communicating HPF tasks. $COLT_{\texttt{HPF}}$ is conceived for use by a compiler of a high-level coordination language to structure a set of data-parallel HPF tasks according to popular forms of *task-parallelism*. Since it requires only small changes to the run-time support of the HPF compiler used, $COLT_{\texttt{HPF}}$ is easily portable among different compilation systems. The paper outlines design and implementation issues, and reports the results of experiments conducted on an SGI/Cray T3E.

1 Introduction

Due to its simplicity, the data-parallel programming model allows high-level parallel languages such as HPF [9] to be designed, which hide low-level programming details and let programmers to concentrate on the high-level exploitation of data parallelism. The compiler manages data distribution and translates the annotated source program into an SPMD code with explicit interprocessor communications and synchronizations. Unfortunately, many important parallel applications do not fit a pure data-parallel model and can be more efficiently implemented by exploiting both task and data parallelism [4]. The advantage of exploiting both forms of parallelism is twofold. On the one hand the capability of integrating task and data parallelism in a single framework allows the number of addressable applications to be enlarged. On the other hand, the exploitation of parallelism at different levels may significantly increase the scalability of applications which may exploit only a limited amount of data parallelism [7].

The promising possibility of integrating task parallelism within the HPF framework recently attracted much attention [10, 6, 14, 3]. A new standard for HPF was proposed, HPF 2.0, which introduces task parallelism by allowing `TASK_REGION` blocks to be specified [10]. Within a `TASK_REGION`, we may specify through `ON` blocks that a set of HPF tasks has to be executed concurrently on disjoint subsets of processors, where each task only accesses data distributed on the associated processor subset. Communications between tasks are accomplished by simple (array) assignments outside `ON` blocks, but inside the outer `TASK_REGION` block. The main disadvantage of the approach is its lacking in generality due to the strong restriction in the task interaction capabilities. In fact, the use of explicit assignments for inter-task communications does not allow programmers to specify an HPF task which, non-deterministically, waits for input data from several other HPF tasks, even though this communication pattern may occur very often in task-parallel programs. Other approaches which also requires changes to the original HPF 1.0 language were proposed [14, 6]. In these cases programmers have to specify an input/output list for all the HPF tasks involved rather than explicitly providing assignments to express communications. Here tasks can be either `PURE` HPF-like subroutines or simple statements. Since programmers don't specify either the allocation of tasks on specific subsets of processors, or explicit communications/assignments between tasks, the compiler has to (1) extract a data dependencies graph of the tasks, (2) decide the best allocation of these tasks, and (3) insert the proper communications between tasks. This approach is much more high-level than the one proposed for HPF 2.0, but it does require sophisticated compiling techniques for task allocation [14]. Nevertheless, the type of task interaction that can be specified is still deterministic. A different approach regards the design of an HPF binding for MPI, proposed by Foster et al. [5]. They propose a framework in which concurrent HPF tasks communicate by means of a restricted subset of MPI communication primitives. We believe that the exploitation of a message-passing interface is too much of a "low-level" approach for introducing task parallelism into HPF. The programmer is charged of all the efforts needed to exploit task parallelism, thus loosing most advantages deriving from the use of a high-level parallel language like HPF.

In this paper we present $COLT_{\mathtt{HPF}}$ (COordination Layer for Tasks expressed in HPF), an MPI–based coordination/communication layer for HPF tasks. $COLT_{\mathtt{HPF}}$ 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. Although $COLT_{\mathtt{HPF}}$ can be exploited to implement libraries of *skeletons*, it was primarily conceived for use by a compiler of a high-level coordination language to structure a set of data-parallel HPF tasks according to popular *task parallel paradigms*, such as *pipelines* and *processor farms* [8, 11]. The proposed high-level coordination language is described elsewhere [12]. Here we focus on $COLT_{\mathtt{HPF}}$ implementation details, and we present some performance results. Although the framework is very different, many implementation issues addressed in the paper are common to those discussed by Foster et al [5]. A detailed discussion of the main differences between the two approaches can be found in [12].

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

We implemented $COLT_{\mathtt{HPF}}$ on top of MPI and Adaptor HPF compilation system [2]. We believe that our technique is very general, so that a similar binding might easily be made with other HPF compilation systems that use MPI too.
The main issue to face for coordinating HPF tasks, are: (1) allowing distinct HPF tasks to be started concurrently on the same (MPI) virtual machine; (2) managing the communication of data distributed within HPF tasks. Note that

the communication of data distributed among the task processors entails several point-to-point communications and, when the data and processor layouts of the sender and receiver tasks differ, it also requires the redistribution of the data exchanged; (3) providing an efficient support for signaling events between tasks, where these signals may be received non-deterministically to implement dynamic policies (e.g. dynamic scheduling techniques [12]).

In the following we discuss how $COLT_{\mathtt{HPF}}$ adresses these three important issues. Note that most $COLT_{\mathtt{HPF}}$ functionalities are implemented as `HPF_LOCAL EXTRINSIC` subroutines [9]. This means that when one of our primitives 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 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.

Group definition and task loading

HPF compilers that use MPI as underlying communication layer exploit one or more MPI *communicators*, a powerful MPI abstraction that permits one to specify communications to occur only within groups of specific processors. Adaptor produces a code which exploits the MPI predefined global communicator `MPI_COMM_WORLD` to perform communications and also to ask for the rank identifying each processor and the total number of processors actually involved. Similar run-time inquiries are present in the code produced by any HPF compiler in order to arrange the logical processor layout onto the physical processor grid and consequently distribute data and computations among the physical processors actually involved. In order to permit Adaptor to support task parallelism, we had to slightly modify its run-time support. In particular our modified version of Adaptor produces an executable MPI code which:

1. reads during the initialization a *configuration file* containing mapping information. The file specifies the number M of HPF tasks constituting the parallel application as well as the name of the HPF subroutine executed by each task. Moreover, for each task $t_i,\ i = 1, \cdots, M$, it specifies the number N_i of processors reserved for its execution. Our simple mapping function associates HPF task t_i with the logical processors whose MPI ranks are in the range $[K_{i-1}, K_i)$, where $K_0 = 0$ and $K_i = K_{i-1} + N_i, \forall\ 1 \leq i \leq M$;
2. creates, M different MPI communicators, one for each group of processors executing an HPF task. At this purpose MPI provides a powerful primitive, `MPI_Comm_split`, which creates all the communicators we need simultaneously, all of them with the same name (`MPI_LOCAL_COMM_WORLD`). `MPI_Comm_split` is called by each MPI processor with a *color* parameter equal to its own task identifier $t_i, i = 1, \cdots, M$.
3. uses `MPI_LOCAL_COMM_WORLD` in all Adaptor-related MPI calls. In this way each HPF task still exploits a self-contained environment, and any HPF communication or collective operation only affects the appropriate processors. On the other hand, inter-task communications provided by $COLT_{\mathtt{HPF}}$ exploit `MPI_COMM_WORLD`.

Once a suitable MPI environment has been created, we have to start the distinct HPF codes onto the disjoint MPI groups of processors. To this end, since according to the static SPMD model provided by MPI the same program has to be loaded on each node, the main entry point of the task-parallel HPF program is a "loader", which invokes the right HPF code on each group of processors. At the beginning the loader invokes the Adaptor initialization routine which create the disjoint MPI groups. Then, the loader, which is dynamically built on the basis of the configuration file described above, forces all the processors belonging to a task (i.e. the corresponding processor *group*) to call the appropriate HPF subroutine. HPF tasks have thus to be written as HPF subroutines, where the task local data are defined within the subroutines themselves.

Channel initialization and data transmission
$COLT_{\mathtt{HPF}}$ associates a *descriptor* with each communication channel used to transmit a distributed array. Such descriptors are used to store information which is subsequently used to optimize communications. To fill the descriptors, all the processors of both the sender and receiver groups, by invoking suitable $COLT_{\mathtt{HPF}}$ initialization primitives, have to carry out the following steps for each distributed array D to be exchanged:

1. inquire the HPF run-time support to find out the layout of D on the HPF processor grid associated with their own processor group. To this end, HPF provides an appropriate intrinsic subroutine [9];
2. exchange mapping information so that each processor involved knows the layout of D at both the sender and receiver sides;
3. on the basis of the information exchanged at the previous step, compute the intersections between D distributions at the sender and receiver sides. To this end, we adopted the *pitfalls* algorithm [13];
4. build, on the basis of the result of the *pitfalls* algorithm, the *Communication Schedule* that is used each time a communication actually occurs to pack (unpack) the elements of D, and send (receive) them to (from) each destination (source) processor. The *Communication Schedule* is stored by each processor in the descriptor associated with the communication channel.

When a $COLT_{\mathtt{HPF}}$ send primitive is invoked to transmit a distributed structure, array data are packed by each processor of the sender group on the basis of the information stored in the channel descriptor, and sent to the processors of the receiver group. In the worst case each processor of the sender group may need to communicate with all the processors of the receiver one. However, the channel descriptor contains all the information needed, so that the processors involved carry out the "minimum" number of point-to-point communications needed to complete the task-to-task communication. Data are sent by means of asynchronous MPI send primitives that are invoked in an order which prevent several processors from simultaneously sending messages to the same destination. On the receiver side, the processors involved receive the messages and unpack the data still on the basis of the information stored in their descriptor. The exchange of scalar values between tasks is simpler. No channel setting is needed in this case. Since HPF scalars are replicated on all the processors of a given task, the send primitive is invoked by all the sender processors, but only the root processor of the source group actually broadcasts the scalar to the processors of the receiver group.

Special messages and non-determinism
Messages often need to be received in a non-deterministic way. For example, an HPF task may need to receive data from a task non-deterministically chosen among several possibilities. The problem is that to ensure correctness the same non-deterministic choice must be globally made by all the processors executing the receiving task. In other words, if a task $\overline{t}$ non-deterministically decides to receive first from task t_i, and then from t_j, this order must be maintained in the point-to-point communications performed by all the processors of $\overline{t}$ in order to accomplish the overall communication. Hence our layer provides an appropriate receive primitive (along with a corresponding send) that causes only the root processor of the receiver group to make the non-deterministic choice of the sender, and then to broadcast its choice to all the other processors of its own group. Only when this choice has been signaled to the other receiving processors, can they invoke the right point-to-point receive primitives and actually receive the data from the selected source task as discussed above.

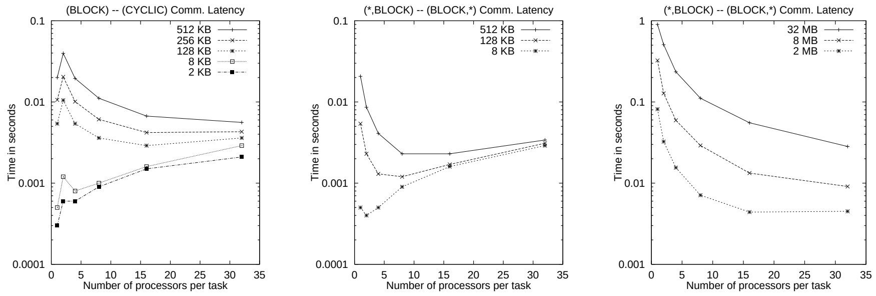
3 Experimental results

In order to characterize the performance of $COLT_{\texttt{HPF}}$ we implemented both synthetic micro-benchmarks and a sample task/data-parallel application. All the experiments were conducted on an SGI/CRAY T3E.
The first micro-benchmark measures the time required to exchange a distributed array between two data-parallel tasks. We executed this sort of "ping-pong" program with 1-D arrays distributed `(BLOCK)` in the source task and `(CYCLIC)` in the destination task, and 2-D arrays distributed `(*,BLOCK)` and `(BLOCK,*)` in the source and destination tasks, respectively. We varied both the array size and the number of processors per-task. The plots reported in Figure 1.(a) show the time required to communicate 1-D arrays. As can be seen, there is a small increase in the communication times when two processors per-task are exploited. This is due to the different data layout exploited which, if several processors are used for each task, entails packing "non-contiguous" data before sending them. Moreover, communication latencies increase with the number of per-task processors for small array sizes, while the opposite effect was measured in the tests involving large volumes of data. This behavior can also be noted in the plots shown in Figures 1.(b) and 1.(c), which report the results obtained with 2-D arrays. In all these plots communication latency decreases up to a minimum and then tends to increase slightly. With very large arrays (e.g. 8, 32 MB) the decrease is constant up to the maximum number of per-task processors tested. The curves thus behave as expected: for small data volumes the communication startup time dominates the overall latency, while for larger arrays the main contribution to communication latency is given by the message transfer time.

Note that this transfer time is directly proportional to the length of messages transmitted and thus indirectly proportional to the number of processors onto which the exchanged array is distributed.

The other micro-benchmark implemented demonstrates the effectiveness of supporting non-deterministic choices for the reception of incoming messages. To this end we built a simple pipeline application composed of three stages. The first stage produces a stream of arrays and sends one array to the next stage every T_1 secs. The second stage is replicated in five copies: each replica performs, on each array received, a dummy computation $\mathcal{C}$ before forwarding it to the third stage. Finally, the third stage simply consumes, every T_3 secs, an element of the stream received from the various replicas of the second stage. It is worth noting that the replicated second stage entails exploiting a processor farm structure within the original pipeline. The first stage, in fact, besides computing its own dummy job, has to dispatch the various arrays of the stream to the various replicas of the second stage (i.e. the farm workers), while the third stage collects the results from them. This application was implemented in two different ways: the first version exploits a *Round Robin* (RR) technique to dispatch the arrays to the five replicas of the second stage, while the second version exploits a dynamic *Self Scheduling* (SS) technique. Moreover, with both the versions we conducted three series of experiments, by changing the cost of the dummy computation $\mathcal{C}$. In the first series of experiments the $\mathcal{C}$'s costs were determined according to an exponential distribution with average μ, while a uniform distribution with the same average μ was instead used for the second series of experiments. Finally, the costs used in the third series were exactly equal to μ for all the stream elements. The value of μ was forced to be equal to 0.2, 0.4 and 0.8 secs, while T_1 and T_3 were fixed to $\mu/5$ (μ divided by the number of workers) to balance the pipeline stages. We used four processors within each data-parallel task, and we fixed to 400 the number of stream elements processed, where each element is a 256×256 array of 8 byte integers. The SS version gave performances from 13% to 14% better than the RR one for exponentially distributed costs. The improvements ranged instead from 11% to 12% in the case of uniformly distributed costs, while in the balanced case, the difference between the results of the two implementations was negligible with a slight performance loss measured for the SS version. These results demonstrate the utility of employing dynamic scheduling strategies when the computational costs are *unbalanced* and unknown until run-time. On the other hand, when execution times are *balanced*, and thus no dynamic scheduling should be needed, the overheads introduced by our implementation of self scheduling are negligible.

The sample application implemented is a 2-D Fast Fourier Transform, (2-D FFT) which is probably the most used example of the utility of exploiting a mixture of task and data parallelism [4, 5, 12]. Table 1 shows the per-input array execution times (in secs) for different problem sizes. The results are reported as a function of P, the number of processors used. Hence the mixed task and data-parallel version ($COLT_{\mathtt{HPF}}$) exploits $P/2$ processors for each of the two communicating HPF tasks implementing the 2-D FFT. As can be seen, the $COLT_{\mathtt{HPF}}$ version considerably outperforms the HPF one. The columns labeled **Ratio** reports the ratio between the pure HPF and the $COLT_{\texttt{HPF}}$ execution times. The performance improvement obtained is significant, and ranges from 11% to 134%. The largest improvements were obtained when 32 or 64 processors were used on small/medium sized problems. These results are particularly interesting because many image and signal processing applications require 2-D FFTs to be applied in real-time to streams of data sets of limited size [7].



(a) (b) (c)

Fig. 1. Average task-to-task communication latencies as a function of the size of the data exchanged and the number of per-task processors.

Table 1. 2-D FFT execution times obtained with HPF and $COLT_{\texttt{HPF}}$.

	256 × 256			512 × 512			1024 × 1024		
Procs	**HPF**	$COLT_{\texttt{HPF}}$	**Ratio**	**HPF**	$COLT_{\texttt{HPF}}$	**Ratio**	**HPF**	$COLT_{\texttt{HPF}}$	**Ratio**
2	0.118	0.106	1.11	0.451	0.367	1.23	1.834	1.452	1.26
4	0.058	0.052	1.12	0.229	0.193	1.19	0.919	0.747	1.23
8	0.029	0.026	1.11	0.115	0.096	1.20	0.468	0.375	1.25
16	0.018	0.014	1.28	0.058	0.047	1.23	0.236	0.190	1.24
32	0.014	0.009	1.51	0.034	0.025	1.36	0.123	0.093	1.32
64	0.017	0.007	2.34	0.028	0.016	1.78	0.074	0.050	1.48

4 Conclusions

In this paper we have discussed $COLT_{\texttt{HPF}}$, a run-time support to coordinate HPF tasks, implemented on top of MPI and Adaptor compilation system. We have also presented some encouraging performance studies, conducted on an SGI/Cray T3E. Although $COLT_{\texttt{HPF}}$ can be exploited to implement parallel libraries (*skeletons*), it was primarily conceived for use by a compiler of a simple high-level coordination language [12]. The main idea behind this language is to allow programmers to coordinate HPF tasks according to specific paradigms for task parallelism. These paradigms, for example pipelines and processors farms, are the most commonly encountered in parallel applications, and, more importantly, can be associated with simple analytic performance models that, on the basis of profiling information, can be used to automatically solve the problem of optimal resource allocation [1]. However, other less structured forms of task parallelism could be supported as well (e.g. computations modeled by general

task graphs), where heuristic techniques can be exploited for task allocation and resource optimization [14]. According to this approach, programmers have only to specify the HPF source code of each task, and the high-level constructs which specify the coordination among the various tasks. The associated compiler, depending on the specific paradigm, generates the suitable "system" code for carrying out task creation and interaction, and integrates the HPF user-provided code with the compiler-generated $COLT_{\mathtt{HPF}}$ calls for inter-task communications. For example, if the paradigm is a pipeline, it generates the code to process streams of data, while, for a processor farm paradigm, it produces the code which dynamically schedules incoming stream elements in order to balance the workload.
Acknowledgments. We wish to thank Thomas Brandes of GMD-SCAI, and the support of PQE2000 project and CINECA Consortium.

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. T. Brandes. ADAPTOR Programmer's Guide Version 5.0. Internal Report Adaptor 3, GMD-SCAI, Sankt Augustin, Germany, April 97.
3. B.M. Chapman, H.P. Zima, M. Haines, P. Mehrotra, and J. Van Rosendale. OPUS: A Coordination Language for Multidisciplinary Applications. TR 95-6, Inst. or Software Technology and Parallel Systems, Univ. of Vienna, Oct. 1995.
4. 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.
5. Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *J. of Parallel and Distr. Comp.*, 45(2):148–158, Sept. 1997.
6. T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*, 2(2):16–26, 1994.
7. T. Gross, D. O'Halloron, E. Stichnoth, and J. Subhlok. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.
8. A.J.G. Hey. Experiments in MIMD Parallelism. In *Proc. of Int. Conf. PARLE '89*, pages 28–42, Eindhoven, The Netherlands, June 1989. LNCS 366 Spinger-Verlag.
9. HPF Forum. *HPF Language Specification*, May 1993. Ver. 1.0.
10. HPF Forum. *HPF Language Specification*, Jan. 1997. Ver. 2.0.
11. H.T. Kung. Computational Models for Parallel Computers. C.A.R. Hoare editor, *Scientific applications of multiprocessors*, pages 1–17. Prentice-Hall Int., 1988.
12. S. Orlando and R. Perego. $COLT_{\mathtt{HPF}}$, a Coordination Layer for HPF Tasks. Technical Report TR-4/98, Dipartimento di Mat. Appl. ed Informatica, Università di Venezia, March 1998. Available at `http://raffaele.cnuce.cnr.it/papers.html`.
13. S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Frontiers'95: The Fifth Symp. on the Frontiers of Massively Par. Comp.*, pages 342–349, Feb. 1995.
14. S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Trans. on Parallel and Distr. Systems*, 8(11):1098–1116, Nov. 1997.