INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A multithreaded runtime environment with thread migration for HPF and C data-parallel compilers*

Luc Bougé
LIP, ENS Lyon

Phil Hatcher
Dept. CS, UNH

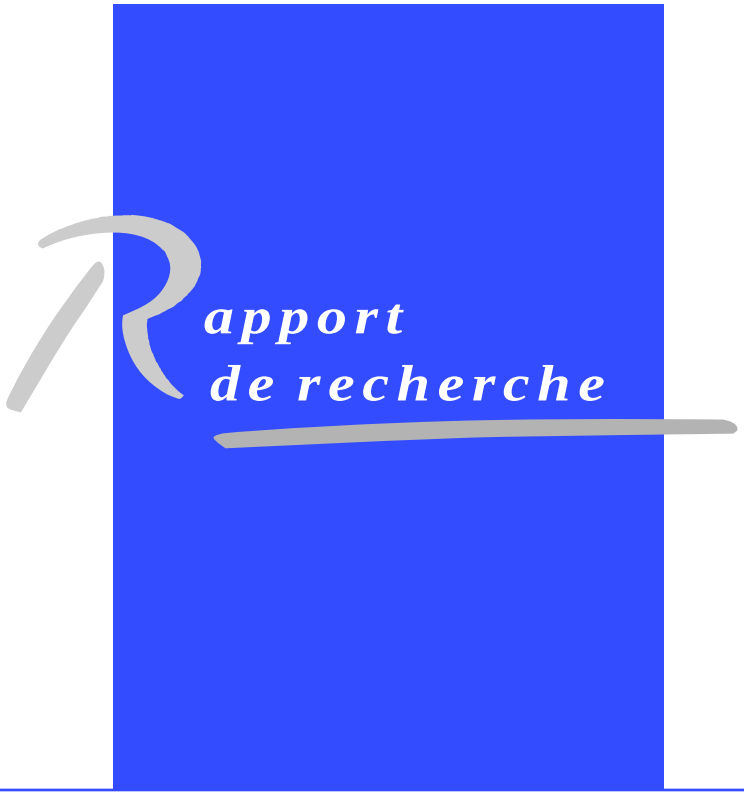
Raymond Namyst
LIP, ENS Lyon

Christian Perez
LIP, ENS Lyon

No 3560

November 1998

THÈME 1



ISSN 0249-6399

RHÔNE-ALPES

A multithreaded runtime environment with thread migration for HPF and C* data-parallel compilers

Luc Bougé* LIP, ENS Lyon
Phil Hatcher† Dept. CS, UNH
Raymond Namyst* LIP, ENS Lyon
Christian Perez* LIP, ENS Lyon

Thème 1 — Réseaux et systèmes
Projet ReMap

Rapport de recherche n ° 3560 — November 1998 — 19 pages

Abstract: This paper studies the benefits of compiling data-parallel languages onto a multithreaded runtime environment providing dynamic thread migration facility. Each abstract process is mapped onto a thread, so that dynamic load balancing can be achieved by migrating threads among the processing nodes. We describe and evaluate an implementation of this idea in the Adaptor HPF and the UNH C* data-parallel compilers. We show that no deep modifications of the compilers are needed, and that the overhead of managing threads can be kept small. As an experimental validation, we report on an HPF implementation of the Gauss Partial Pivoting algorithm. We show that the initial `BLOCK` data distribution with our dynamic load balancing scheme can reach the performance of the optimal `CYCLIC` distribution.

Citation. An abridged version of this report has been published in the Proceedings of the IEEE PACT'98 Conference [4]. Please, mention this reference in any citation.

Key-words: Data-parallel languages, compiler, HPF, C*, thread migration.

(Résumé : tsvp)

This work has been supported by the NSF/INRIA *C*IT* Cooperative Research Grant. The LIP Laboratory is jointly supported by ENS Lyon, CNRS and INRIA (UMR 8512).

* Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon, F-69364 Lyon, France. Email : {Luc.Bouge,Raymond.Namyst,Christian.Perez}@ens-lyon.fr
† Dept CS, Univ. New Hampshire, Durham, NH 03824-3591, USA. Email : Phil.Hatcher@unh.edu.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Un exécutif multiprogrammé avec migration de processus légers pour compilateurs à parallélisme de données HPF et C*

Résumé : Ce papier étudie les avantages de la compilation de langages à parallélisme de données sur des environements multiprogrammés offrant la migration dynamique de processus légers. Chaque processus abstrait est mappé dans un processus léger. Ainsi, l'équilibrage dynamique de charge peut être réalisé par la migration de processus légers entre les nœuds de calcul. Nous décrivons et évaluons une implémentation de cette idée dans les compilateurs à parallélisme de données Adaptor (HPF) et UNH-C*. Nous montrons que des modifications profondes des compilateurs ne sont pas nécessaires et que le surcoût de gestion des processus légers peut être maintenu faible. Comme validation, nous présentons une implémentation en HPF de l'algorithme du pivot partiel de Gauss. Nous montrons que la distribution de données initiale (`BLOC`) peut atteindre les performances optimales de la distribution `CYCLIC` avec l'aide d'un équilibrage de charge.

Citation. Une version abrégée de ce rapport a été publiée dans les actes de la conférence IEEE PACT'98 [4]. Merci de mentionner cette référence dans toute citation.

Mots-clé : Langages à parallélisme de données, compilateur, HPF, C*, migration de processus léger.

Table des matières

Introduction

Data-parallel languages are now recognized as major tools for high performance computing. Considerable effort has been put in designing sophisticated methods to compile them efficiently onto a variety of architectures, including MIMD clusters of commodity processors interconnected by very high-speed networks. As of today, this effort has been mainly concerned with the compilation of the High Performance Fortran (HPF) language [17], and to some extent to the C* language, a data-parallel version of C designed by Thinking Machine Corporation in 1990 for the Connection Machines series [31]. It is the base for the Data-Parallel C Extension (DPCE) definition of an ANSI committee [10]. The compilation method consists in generating a SPMD parallel program from the data-parallel source code. The SPMD processes are typically (plain) Unix processes which communicate one with the other by exchanging messages through some standard message-passing library. Each process is in charge of managing a "slice" of the parallel data structures. Considerable effort has been devoted to minimizing the overhead induced by inter-process communications. It has mainly focused on designing sophisticated ways to distribute data slices among processes.

It has often been stressed that this approach cannot efficiently cope with irregular computations where the amount of work induced by the individual slices may vary in time, even if these variations are partially predictable at compile time. Expensive runtime redistribution of data among processes are then needed.

In a previous paper [26], we have proposed a new compilation technique to address this point. The idea is to use *mobile lightweight processes* (thereafter called threads) instead of heavy, plain processes. Many multithreaded programming environment are available (to cite two of them : Nexus [13] and Athapascan [8]), but the concept of *mobility* is still rarely addressed. Such mobile threads are provided by the PM2 multithreaded programming environment [21]. In our case, dynamic load balancing is achieved by migrating the threads from one node to another. We show that this approach only requires minor changes in the core of existing compilers. Most changes are located in the runtime library, and are therefore largely independent of the specific compilation scheme.

We describe in the paper two prototype compiling environments which have been designed along this idea. The first one is a modified version of the Adaptor HPF compiler [6]. The second one is a modified version of the UNH C* compiler [16]. In both case, we provide preliminary performance evaluations which demonstrate the interest of the approach and the various tradeoffs involved.

Section 1 describes the general approach. It lists the problems found by the classical compilation scheme and presents the benefits of using a multithreaded runtime environment. Section 2 provides a preliminary performance evaluation of the HPF implementation which demonstrates the interest of the approach and the various tradeoffs involved. The C* implementation is reviewed in Section 3. Section 4 is the conclusion.

1 Multithreaded runtime environment for data-parallel languages

1.1 Compiling data-parallel languages to MIMD architectures

This section describes the general principles of data-parallel compilation schemes, as used for instance for HPF and C*. Both are based on the Owner Computes Rule (OCR) : the computation necessary to update a slice of data is carried out by the processor which stores it.

The HPF language The High Performance Fortran (HPF) language [17] specifies the allocation of data slices to processors through a 3-level scheme. In HPF, the parallel data are multidimensional arrays. First, arrays are mutually `ALIGN`ed, possibly with respect to optional abstract reference arrays called `TEMPLATE`s. Mutually `ALIGN`ed array elements are guaranteed to be eventually stored into the same physical memory. This step is entirely handled at compile-time, and is of little interest to us. Second, arrays are `DISTRIBUTE`d onto the virtual topology of processors as defined by the `PROCESSORS` directive. These processors are called *abstract processors* in the HPF terminology. Several distribution strategy can be used : `BLOCK`, `CYCLIC`, etc. Third, abstract processors are mapped onto the real topology of the physical processors of the architecture.

Throughout the paper, we refer to the physical processors as *nodes*. In most HPF compilers, both topologies have the same number of processors. They may only differ in their geometry, so that this last level is mostly trivial. Moreover, the HPF document itself (see [17, p. 19]) does not specify precisely what should be done if it is not the case, that is, if the `PROCESSORS` directive is not trivial. It is specified to be implementation dependent.

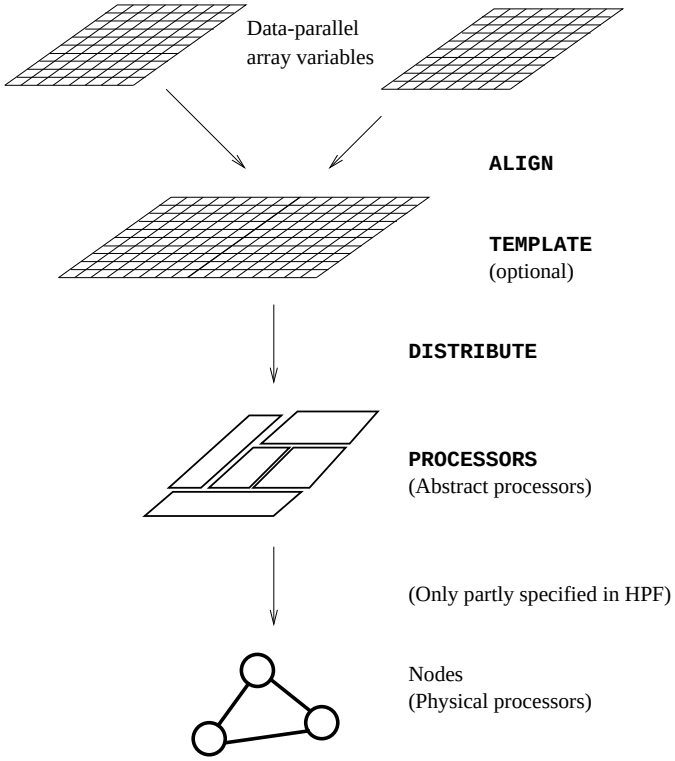


FIG. 1: The HPF data placement model. Note that the last level is only partly specified by the current language specification.

Most HPF compilers generate an SPMD code based on this assumption. Each abstract processor defined by the `PROCESSORS` directive is emulated ("*virtualized*") by a separate process placed on a separate node. We will call them *system processes*. The process stores the data slice it is in charge of managing, and it is responsible for fetching the remote data which are involved in the updating of this slice (*Owner Computes Rule*). This is done by exchanging point-to-point messages with the other processes through some standard message-passing communication library (MPI). There is thus a perfect overlapping between three distinct notions :

- the HPF abstract processor as defined by the `PROCESSORS` directive ;
- the system process which implements its behavior ;
- the physical node which runs this process.

The C* language The case of the C* language is somewhat simpler [31]. Parallel data are are aligned by construction onto abstract arrays called `shapes`. There is no freedom in alignment as in HPF. In the C* terminology, each element of a `shape` is called a virtual processor (somewhat inconsistently with HPF, hence the term *abstract processor*). Then, `shapes` are distributed onto the physical nodes. In the UNH C* compiler, this is done according to a `BLOCK` strategy. Each of the blocks can be seen as an abstract processor in the sense described above, but here is no notion of a virtual topology there. Finally, each physical node runs a system process to emulate ("*virtualization loop*") all the `shape` elements it manages. Again, we have

a perfect overlapping of the notions : the system process which emulates the parallel actions on a block of `shape` elements and the physical node which runs the process.

Runtime Both the Adaptor HPF and the UNH C* compilers are based on extensive runtime libraries which implement a number of high-level communication primitives such as sending and receiving a point-to-point message, broadcasting a scalar value, reducing a parallel value down to a scalar, etc. For portability purpose, these libraries are based on a small number of low-level routines, typically found in most standard message-passing libraries such as MPI. Of course, the efficiency of the generated code crucially depends on the efficiency of the runtime library.

Problems The compilation scheme sketched above raises a number of difficult problems.

- As already mentioned, the runtime management of the communications between processors has a dramatic impact on the performance. A huge effort has thus been spent in optimizing this aspect. Sophisticated techniques have been proposed to reschedule the generated code such that the overlapping of communications by computation is maximal. All these techniques are based on a static analysis of the program, so that they work poorly as soon as the behavior cannot be fully predicted.
- The compilation scheme is based on the notion of an owner for each element of parallel data. The load of a processor depends on the number of data elements it owns or/and their value. An optimized distribution of data is thus the key toward performance. Sophisticated techniques have been devised to provide the programmer with a large panel of alternative distribution strategies. Examples of them are the general block distribution and the user-defined data distributions like in Vienna Fortran [7], Fortran D [28] or Annai [19]. One of the main advances of HPF2 [17] with respect to the original HPF was to extend the flexibility of the `DISTRIBUTE` directive. Again, such static strategies cannot adapt to dynamic variations in the amount of work required by the data, nor in the amount of computing power or memory space available at the nodes (in the case of a cluster shared by several users, for instance). The only possibility is then to `REDISTRIBUTE` the data among the abstract processors, which is a very expensive operation.
- Finally, such compilation schemes are specifically targeted towards current distributed-memory architectures based on single-processor nodes. No provision is made to exploit the power of multi-processor nodes which are bound to become commonplace within the coming years.

1.2 Why to use a multithreaded runtime environment ?

From coarse-grain to medium-grain abstract processors The basic compilation scheme above considers *coarse-grain* abstract processors. Each abstract processor is mapped onto a separate system process, which is run on a separate node. In contrast, we could consider *medium-grained* abstract processors, each mapped onto a separate system process, and then place many of them onto a single node. The benefits of this approach would be the following.

- Overlap communications by computations thanks to the native system process scheduling on the node. Much work has been devoted to explore this aspect [1, 2, 30, 25], though some researchers [12] question the impact of this approach.
- Achieve load balancing by moving abstract processors from one node to another, instead of moving data from one abstract processor to another. This approach has been explored in the context of task scheduling. When the task grah is (statically) predictable, on can look for an optimal mapping of the abstract processors on the nodes [11, 33, 9]. For certain kinds of HPF loops, it is also possible to dynamically map the abstract processors to balance the load with a work stealing technique [23]. In contrast, we propose to let abstract processors *migrate* dynamically between the nodes within their lifetime.
- Take advantage of multi-processor nodes where available, just through the native process scheduling [27].

– As an extra bonus, distributing the data among a larger number of abstract processors improves the locality of their accesses, so that the cache hit gets increased. This side-effect can improve the efficiency quite noticeably, as reported in [2, 25, 26].

Going from coarse-grain abstract processors to medium-grained ones amounts to use non-trivial `PROCESSORS` directives in HPF. Say for instance `!HPF$ PROCESSORS PROC(64)` on a 8-node architecture, so that there will be 8 abstract processors on each node. This lets a data-parallel compiler generate an SPMD code for 64 processes. As modern message-passing libraries are able to send messages even between processes located on the same node, just launching 8 processes on each node will do the work. No modification whatsoever is needed here. Of course, the main problem with this approach is performance.

– Using MPI to exchange messages between processes which share the same physical memory is wasteful.
– The context switching time between system processes is high, so that the overhead of running many small processes instead of one large one is considerable.
– No standard communication library allows processes to be moved from nodes to nodes. Some prototype implementations have been described, but the reported performance cannot be sufficient for our needs [18].

Yet, the problem lies in the cost of process managment, not in the idea ! Observe that the processes generated by a data-parallel compiler make little if no use at all of process facilities such as signals or I/O. Downstripped lightweight processes may as well do the work, at least as long as I/O are not involved.

Our proposition We therefore propose to use a multithreaded runtime environment for data-parallel languages. POSIX-like multithreaded programming environments are now widely available. We chose the PM2 programming environment, as it is available on a large variety of systems, and it provides a primitive to migrate threads from a process to another process in a almost fully transparent way. Only minor modifications to be discussed later have been needed. Using lightweight threads instead of system processes to run medium-grain abstract processors brings the following additional benefits, besides the ones listed above.

– Very short context switching time (a couple of microseconds for PM2), which provides a very good overlapping of communications by computation. As soon as a thread gets blocked on a reception, it is suspended and a ready thread is resumed.
– The semantic model implemented by threads is an extension of that of processes, as long as pure computation is concerned. The code generated by current compilers for processes can be run by threads as well.
– Thread migration is provided by PM2 at no extra cost. It is very efficient, as discussed later.
– The runtime library can be rewritten so as to implement communications between threads within the same process as direct copies. The underlying message-passing library needs only to be called when exchanging data between remote threads.
– If multiples processors are available at some of the nodes, then they can be exploited at no extra cost.

1.3 The PM2 multithreaded programming environment

We only sketch in this section the aspects of the PM2 multithreaded programming environment which are central to this work. An extensive presentation of the environment and its implementation details can be found in [22, 21].

General presentation PM2 [22, 21] is a distributed multithreaded environment designed as an efficient runtime for irregular parallel applications. The main objective of PM2 is to provide a carefully chosen set of basic functionalities on top of which many dynamic load balancing policies are easy to implement. Because PM2 applications may generate a large number of threads with unpredictable lifetimes, the PM2 programing model is based on the concept of *mobile threads*. In this way, threads can be preemptively migrated from one node (say, a Unix process) to another one without any explicit state backup nor global synchronization.

Even the most efficient thread migration mechanism is not usable if further interaction between migrated threads are costly. Therefore, interactions between threads are done through well defined decomposition operators (such as the *Lightweight Remote Procedure Call* mechanism) that can be efficiently made "migration tolerant" by PM2.

The current implementation of PM2 is based on two software components : a POSIX-compliant thread package (*Marcel*) and a generic communication interface (*Madeleine*).

Marcel Compared to a classical Pthread library, the Marcel package introduces some original features which are needed by PM2 to implement functionalities such as thread migration or reduced preemption. Its implementation is currently available on the following architectures : Sparc, ix86, Alpha, PowerPC and Mips.

Madeleine The Madeleine communication layer was designed to bridge the gap between low-level communication interfaces (such as BIP, SBP or UNET). It provides an interface optimized for *RPC-like* operations that allow zero-copy data transmissions on high-speed networks such as Myrinet or SCI.

Performance We present here some results to give the reader a rough idea of the efficiency. Each node consists in a 200 MHz PentiumPro processor. The operating system is Linux 2.0.29. The nodes are interconnected by a Myrinet network from Myricom [3, 20] accessed through the BIP low-level communication interface [32].

Figures 2 and 3 show the performance of PM2/Madeleine over Myrinet. This implementation uses BIP as the underlying communication layer. On figure 2, the performance of an optimized implementation of MPICH on top of BIP is provided as a reference. It shows that although PM2 remote invocations are more complex interactions than MPI message transmissions [29, 15], they can be efficiently implemented thanks to an interface such as Madeleine.

Figure 2 shows that the latency of PM2 over Myrinet is very low (8 μs for a remote invocation with no parameter). The overhead over raw BIP is small. This is mainly due to the transfer of additional data needed for service designation and flow-control processing. Because this overhead is constant, the full bandwidth is reached for large enough messages as shown by Figure 3. This shows that the zero-copy implementation of Madeleine has a positive impact on performance. Thus, communications under PM2 are far from being the bottleneck in the experiments detailed below.

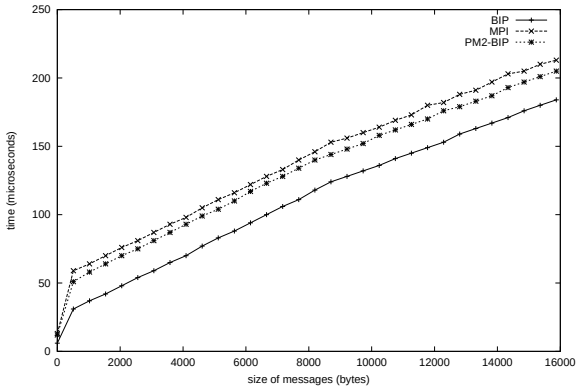


FIG. 2: Latency of asynchronous LRPCs of PM2 compared with low-level and high-level communication libraries.

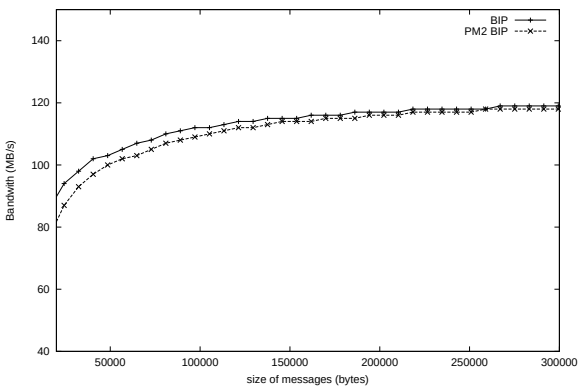


FIG. 3: Bandwidth of PM2 compared with low level communication libraries.

Dynamic thread migration A PM2 thread may silently migrate from one process to another without interrupting the application. Only the origin and the destination processes are active during migration. No global action whatsoever is needed. The destination does not need to wait for the incoming thread : it can

proceed with its local threads as long as no dependency with the incoming thread is involved. Thus, thread migration can be overlapped by computation and a minimal cost is induced. Also, several threads may migrate together between two processes.

The migration of a PM2 thread requires the destination process to allocate a fresh stack, possibly at a new address. Thus, all pointers have to be recomputed after a migration. PM2 automatically relocates the thread stack and updates its internal pointers, but it cannot cope with the user pointers which may have been pushed as values on the stack. It rather provides a primitive which allows the user to register these pointers. After a migration, PM2 automatically updates all registered pointers. Thanks to this facility, thread migration do not induce any global synchronization nor any global address space reservation like in UPVM or Millipede [18, 14].

In our case, we mainly have to cope with pointers to memory blocks allocated by the runtime library in the heap. PM2 provides the user with a number of hooks attached to the migration routines. One hooked function may be called just before sending the migration message and another one when the thread is ready but not yet inserted in the ready queue at the destination process. With these functions, we can pack all the memory blocks allocated in the heap into the migration message at the origin process, unpack them at the destination process (notice the compiler does not generate any pointers, only integer indices !). These hooks are also used to compute some internal values and possibly adjust internal pointers.

1.4 Putting everything together

Let us now review the main modifications needed to retarget data-parallel compilers towards a multithreaded programming environment. We focus here on the Adaptor HPF compiler developped by Thomas Brandes, GMD, and on the UNH C* compiler, developed by Phil Hatcher, UNH.

Code generation No deep modification of the generated code is needed here, at least, when I/O are not concerned. The only point is to let the compiler "believe" it generates code for a parallel archiecture with n nodes, where n is unusually large : up to 32 threads per node on 8 nodes in the preliminary expriments described below, that is $n = 256$.

Yet, achieving *full* performance requires to redesign the way scalar computation is handled by the compiler. Both the Adaptor HPF compiler and the UNH C* compiler replicate scalar computations on each node, so as to avoid broadcasting each time a scalar variable is udpated. This scheme remains correct when using threads instead of processes, but it leads to a significant waste of computing power : the same computation is done again and again by each of the threads run by a node, whereas its result could be directly shared. Implementing such an optimization requires some deeper modification of the compilers, which we are currently investigating. The experimental study presented below does not include this optimization.

Runtime The runtime libraries of the Adaptor HPF compiler and the UNH C* compiler have to be extensively redesigned. All the global variables have to be privatized, as several threads may call runtime routines concurrently. References to remote abstract processes have to be changed to references to threads instead of system processes. The communication primitives have to be updated so that communicating within a single process bypasses the underlying communication library by issuing a physical copy. This, in turn, necessitates some extra buffer management by the threads. Also, each node has to maintain a global table to determinate the location of all the threads throughout the application. Finally, the multithreaded programming environment PM2 we use is based on Remote Procedure Calls (RPC) rather than message passing. Though it is straightforward to implement message passing calls on top of remote procedure calls, it is more efficient to redesign a number of high-level communication routines directly in terms of RPC. All these modifications are somewhat tedious, but routine.

Considering threads instead of processes also allows for some alternative, optimized implementations of many collective operations such as broadcast or global synchronization. The idea is to first perform the operation at the node level between threads, using a shared-memory model, then at the global level between

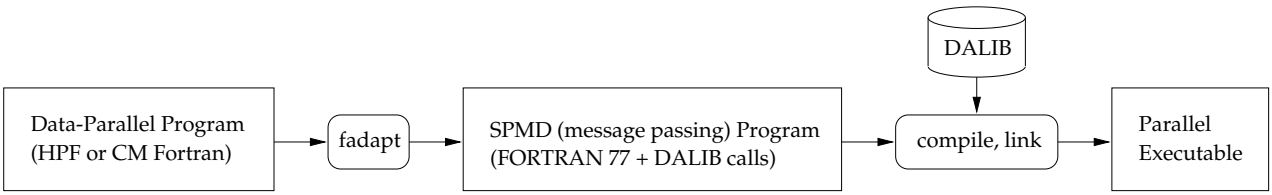


FIG. 4: The Adaptor system

nodes, using a distributed-memory model. The dominating cost is by far the global phase , so that little penalty is induced by using multithreaded nodes here.

Underlying multithreaded programming environment PM2 is a multithreaded programming environment designed to be used for a large class of applications. Using it as an efficient target for data-parallel compilers requires a number of specific tuning of some functionalities, as detailed below.

The global synchronization operation is at the very heart of data-parallel compilation schemes, and the efficiency of inter-thread synchronization is of dramatic concern. Local thread synchronization is implemented as a two-phase operation. Let n be the number of threads to synchronize. In the first phase, $n-1$ threads get blocked on a semaphore waiting for the nth one; in the second phase, the nth thread releases the $n-1$ blocked threads. Using threads for data-parallel computing allows a number of optimizations, as all computing threads have the same priority. All blocked threads can therefore be inserted back into the ready list by a single operation. However, the cost of a synchronization is still $O(n)$ because of the first phase. The optimization only reduces the value of the multiplicative constant from 4.8 μs to 3.1 μs per thread on a Pentium 133 MHz Linux machine. This is a 35% improvement. On this machine, the PM2 thread context switch roughly takes 1.5 μs.

PM2 is a preemptive environment. But it turns out that full preemption is not well-suited when threads are used to run data-parallel programs. Consider the case where the computation task is approximately balanced between threads. As all the threads have the same priority, they progress in close synchronization. The threads thus reach the communication calls at approximately the same time. With n identical threads, the delay is at most $n-1$ times the time slice. Only a small overlapping of communications by computation is then possible. To maximize this overlapping, a thread should not be preempted until it blocks. On the other hand, the network has to be regularly polled to accept incoming messages. The specific preemption mechanism we have implemented in PM2 does not allow a working thread to preempt another thread. Only the communication thread may regularly preempt working threads until it get blocked on a message reception. Then, the control is yielded back to the preempted thread to insure maximal overlapping.

2 A preliminary implementation within the Adaptor HPF compiler

Adaptor [6] is a public domain HPF compiler developed at the GMD (Germany) by Th. Brandes. It transforms HPF (or CM-Fortran) data-parallel programs into FORTRAN programs with explicit message passing. Adaptor itself consists of two components. `fadapt` is a source-to-source translator from HPF to F77 (or F90), and `DALIB` is the HPF runtime system that handles descriptors for arrays, sections and distributions and which implements the communication routines. We have used Version 4.0a of May 1996. Figure 4 outlines the main components of the Adaptor system.

2.1 Adapting Adaptor to multithreaded code generation

The `fadapt` component has been left unchanged but for a simple postprocessing script. It transforms the main `program` into a `subroutine`, so that it can be run by a thread, and adds a new main `program` which calls our own initialization routine. A specific module has been added to the the `DALIB` runtime component

to map its generic message-passing interface to the PM2 Remote Procedure Call facility. The `DALIB` runtime component has been substantially modified, as described in Section 1.4.

2.2 Evaluation

The experimental evaluations described in this section have been done with our modified version of the Adaptor compiler. All the measures have been obtained on a cluster of PentiumPro 200 MHz machine with 256 KB of cache and 64 MB of EDO RAM. The operating system is Linux 2.0.29. The communications use TCP over a Myrinet network. The C compiler used is GCC 2.7.2 with the `O4` optimization option. We use Version 1.6 of the Portland Group FORTRAN 77 compiler. The FORTRAN 77 compiler options are `O2`, `dalign`, `recursive` and `reentrant`. We exactly place one Unix process on each physical node.

2.2.1 Purely scalar code

The goal of this experiment is to demonstrate that the scalar computations are replicated in all threads. The test program is a scalar loop containting a scalar expression. To avoid cache effects, the scalar expression is a call to an empty function. We turn off all compiler inter-procedural optimizations by compiling the empty function in a different file. This test is executed on a single-processor machine. The parameters are the number of abstract processors, i.e., the number of threads, and the size of the loop. The normalized time displayed at Figure 5 is the experimental time divided by the loop size and the number of threads. It can be seen that the time is proportional to the loop size and to the number of threads : the scalar code is replicated.

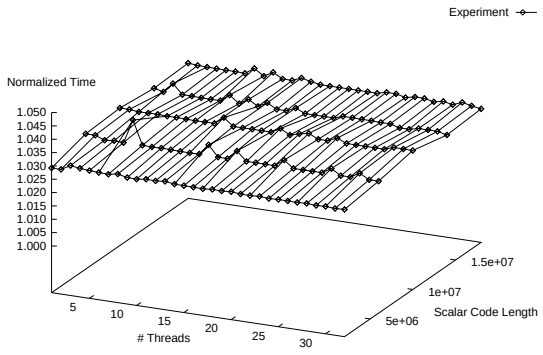
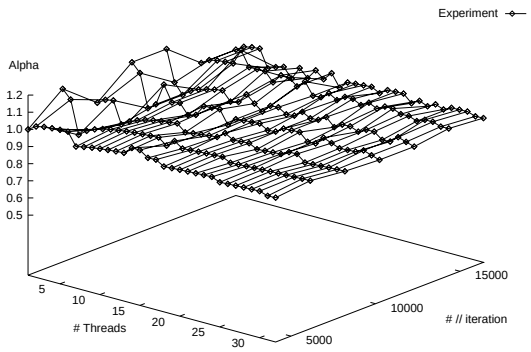


FIG. 5: Normalized time of purely scalar code as a function of the number of threads and the size of the scalar code. The normalized time is the experimental time divided by the loop size and the thread number.

2.2.2 Purely parallel code

This experiment aims at estimating the overhead induced by running multiple medium-grain abstract processors on a single-processor machine, instead of running a single coarse-grain one. The test program is a fully parallel loop embedded in a scalar loop of 4000 iterations in order to increase granularity. In the generated code, we have changed array operations to function calls to avoid cache effects. The tests are executed on a single-processor machine. The parameters are the number of abstract processors and the parallel loop size. The time for such a program is of the form $t = \alpha \times n_t + \beta \times p + \gamma$ where t is the time, n_t the number of threads and p the parallel loop size. Figure 6 tabulates the normalized constant α in function of the number of threads and of the parallel loop size. We have used the least squares method to obtain an average value of α.



Number of threads	Parallel loop size		
	4096	8192	16384
1	0.867	1.733	3.464
2	0.869	1.735	3.466
4	0.873	1.738	3.470
8	0.880	1.746	3.476
16	0.896	1.761	3.492
32	0.925	1.791	3.522
Plain Adaptor	0.867	1.732	3.464

FIG. 6: Normalized α value and execution time of a fully parallel code without communication as a function of the number of threads and the size of the parallel code on a single-processor machine.

The values of α tabulated in Figure 6 are in the range $[0.91 \ldots 1.16]$, which validates the above equation. The thread overhead in a parallel loop depends on the number of threads and not on the parallel loop size. For this experiment, the overhead is around 2 ms per thread for 4000 iterations of the parallel loop. A deeper analysis shows that the replication of the scalar loop costs around 1 ms. Thus, the whole thread management cost is around 1 ms per thread, which is far less than 1%. The overhead is negligible.

2.2.3 Communications in parallel code

The next question addresses the communications cost in parallel code. What is the overhead compared to the original version without threads ? We have already seen that the PM2 communication overhead is small compared to the performance of the underlying communication library. The test program for this section is the usual Jacobi program, whose communication/computation ratio can be controlled by using various matrix sizes.

Program Version	Matrix size			
	256	512	1024	2048
Adaptor without thread	0.54	1.70	6.38	29.35
Adaptor with threads	0.44	1.55	6.31	29.28

FIG. 7: Time in seconds of a Jacobi program on 8 processors. The version with threads has only one thread per process.

The communication overhead induced by threads is not noticeable when there is one thread per process as shown in Figure 7. The version with threads is slightly faster because of side effects of the alarm signal used in PM2 on the UNIX process scheduler [25].

2.2.4 Migrating abstract processors

This section reports on the experimental cost of migrating an abstract processor. The parameters are the volume of user data migrated together with the abstract processor and the number of user arrays in which these user data are scattered. This is a significant parameter as each array is individually packed. The packing phase is done automatically in the migration phase. The experiment consists in migrating an abstract processor forth and back between two nodes. The migration time includes the packing, the transmission and the unpacking (including numerous memory allocations) of the abstract processor. Figure 8 shows the result of this experiment done with PM2 using a TCP protocol on a Myrinet network.

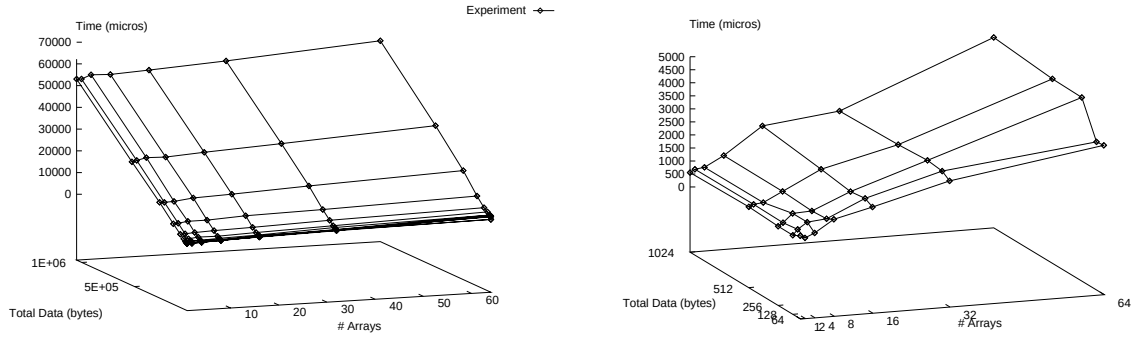


FIG. 8: Time needed to migrate an abstract processor between two nodes as a function of the total volume of user data and the number of arrays. The right figure is a zoom of the left figure.

The user data bandwidth is around 19.3 MB/s. The maximal bandwidth on PM2 for this configuration (based on the TCP protocol on Myrinet), is around 20.4 MB/s. Thus, the user data are sent using 95% of the available bandwidth. Migrating an array takes around 84 μs which is spent in packing/unpacking the array itself and its internal structures. Memory allocation of small size is very fast on Linux, it takes less than 1 μs, while large memory allocation takes as much as 40 μs. As there is only one large memory allocation (the one for the data), and all other are small, 84 μs appears to be a good figure. Finally, there is a constant cost of 70 μs which represents the time spent by the thread migration function of PM2. In conclusion, the cost of managing the migration of an abstract processor is small compared to the cost of transferring its data. For example, for 2 arrays of 20 kB each, the user data transfer represents 90% of the migration time.

2.3 Experimental validation : Partial Gaussian Pivoting

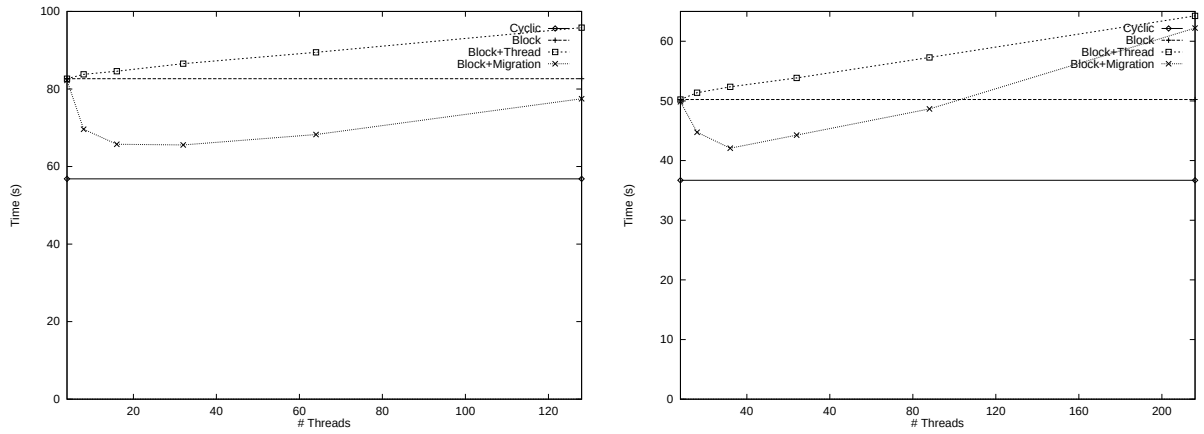
This last experiment aims at validating the whole compilation chain. We consider a classical algorithm, Partial Gaussian Pivoting, which exhibits an irregular, yet fully predictable, behavior. A default `BLOCK` distribution of the matrix leads to a very imbalanced behavior : by the end of the algorithm, only a few blocks are still working. In contrast, it has been shown that there exists an optimal static distribution for the data, which minimizes the overall running time : it is the `CYCLIC` distribution. The question is : can we achieve the performance of the optimal `CYCLIC` when starting from the default `BLOCK` and moving abstract processors around according to some predefined strategy ?

Our test matrices are very regular to avoid column exchanges. As the data distribution is different for the cyclic and the block distribution with load balancing, a column exchange may generate inter-node communications in one distribution and not in the other. To avoid this, we always choose the pivot in the current column : at the kth iteration, the pivot is in column k.

The load balancing strategy consists in computing a new abstract processor mapping at each iteration by moving an abstract processor to the node which has sent the last pivot to other nodes. The choice of the abstract processor to be sent is not arbitrary. One chooses the abstract processor with the highest line number among the abstract processors allocated to the most loaded node. This is the kind of information that could be detected at compile time. If the load in the new mapping is better balanced than the current one, then the abstract processor is migrated. This strategy may be not the best one, but it is simple. The overhead introduced by computing the new mapping is quite small, around 2% of the total time.

In [26], we reported on a similar experiment done on hand-written code. We describe here the result of the experiment run with the program generated by our version of the Adaptor HPF compiler without any additional modification. Only the load balancing strategy has been hand-coded. The results are displayed in

Figure 9 and Figure 10. Figure 10 displays the efficiency of the BLOCK distribution with threads version with respect to the CYCLIC distribution without thread.



(i) 2048 × 2048 matrix on 4 processors

(ii) 2048 × 2048 matrix on 8 processors

FIG. 9: Experiments with the Partial Gaussian Pivoting. The x-axis is the total number of threads. The y-axis is the total running time. For each figure, we have plotted the running time of the HPF program compiled with the modified Adaptor compiler with the CYCLIC and the BLOCK distribution without threads (the horizontal lines) and with BLOCK distribution with threads, with and without dynamic load balancing.

These experiments yield a number of conclusions. First, the effect of replicating scalar code appears to be quite important as witnessed by the significant slope of the curve for the BLOCK distribution with threads but without migration. The scalar code consists of the management of the columns and of the rows. We can observe that the favorable cache effects cannot compete with the overhead of scalar replication in any situation as the curve for the BLOCK distribution without migration never decreases. Second, each case shows a range of numbers of threads for which dynamic load balancing migration improves the running time with respect to the static BLOCK distribution without thread.

Number of Processors	Matrix size			
	256	512	1024	2048
2	1.21	1.19	1.16	1.09
4	0.84	1.00	1.08	1.15
8	1.16	1.20	1.22	1.15

FIG. 10: Ratio between the BLOCK distribution with thread migration version, and the CYCLIC distribution without thread version.

We can remark in Figure 10 that the efficiency usually lies between 1.10 and 1.20, in spite of the overhead induced by scalar code replication. We can also note that for one experiment, 4 processors and 256 × 256 matrix, the multithreaded version with thread migration achieves better times than the CYCLIC version. This is due to cache effects.

3 A more elaborate implementation with the UNH-C* compiler

The UNH C* programming environment [16] consists of a public domain compiler for the C* language [31]. It has been developed at the Univ. New Hampshire by Ph. Hatcher. It transforms a data-parallel C* program into a C program which makes calls to a runtime library. This library implements the high-level

communication functions of the C* language using a low-level library for communication management. Our implementation is based on Revision 3.1 of the UNH C* driver.

3.1 Modifying the UNH C* compiler

The modifications have been made to the code generator and to the runtime library. The modifications of the runtime library are quite similar to those made for Adaptor, as described in Section 1.4. The modifications of the code generation are more elaborate. As discussed in Section 1.3, the main problem to be addressed concerning thread migration lies with pointer relocation. Our approach is to generate array indexes wherever it is possible, so as to avoid pointer translation problems. This technique does not solve the problem of scalar pointers but enables a first evaluation of thread migration in a C* environment.

To emulate the data-parallel operations on its own slice of parallel data, each abstract processor generates virtualization loops. On entering such a loop, the array indices are converted into pointers. It allows faster access to data, and provides room for clever optimizations by the underlying C compiler. Of course, this makes load balancing within a virtualization loop impossible. In fact, this restriction is handled at the code generation level by an elegant side-effect. The load-balancing phase is triggered by a call to a C* function which is not declared as "pure". Only "pure" function calls may occur within virtualization loops, so that the code generator splits the loops into parts on such "non-pure" calls.

3.2 Experimental evaluation

The Adaptor HPF compilers and the UNH C* compilers are very close with respect to our concerns. They both generate generic code that relies on a high-level runtime library, which in turn is based on a low-level library for explicit message passing routines. Thread support has been included with the same methodology in both cases, and similar results are thus to be expected. Yet, the impact of transforming pointers into array indices has to be careful assessed. This section presents the scalar and parallel code experiments. Then, it focuses on the cost of using array indexes. It finishes by the evaluation of the thread migration cost.

All the measures have been obtained on a cluster of 200 MHz PentiumPro processors with 256 kB of cache and 64 MB of EDO RAM. The operating system is Linux 2.0.29. The communications are done through the TCP protocol over a Myrinet network. The underlying C compiler is GCC 2.7.2 used with the O4 optimization option.

3.2.1 Purely scalar code

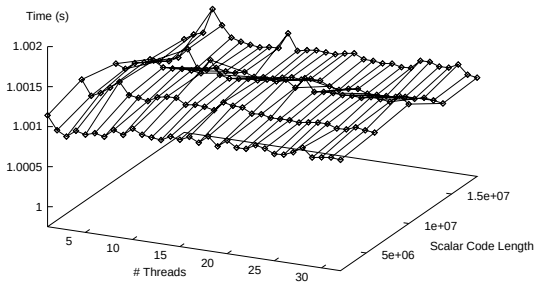
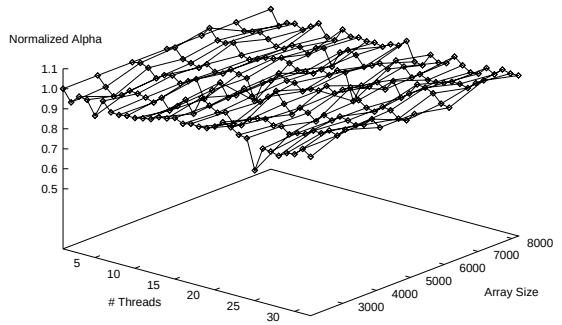


FIG. 11: Normalized time of a scalar code as a function of the number of threads and the size of the scalar code. The normalized time is the experimental time divided by the loop size and the thread number.

Like in Section 2.2.1, the test program is a scalar loop over a function call. The parameters are the loop size and the thread number. The normalized time displayed in Figure 11 has been obtained by dividing the experimental time by the thread number and the loop size. The resulting curve is flat : the scalar code is replicated over the threads.

3.2.2 Purely parallel code

Like in Section 2.2.2, the execution time of a fully parallel C* expression is linear in the number of threads and in the shape size : $t = \alpha \times n_t + \beta \times p + \gamma$ where t is the time, n_t the number of threads and p the parallel variable size. α represents the thread overhead. We validate this expression and we evaluate α by using a test program which is a scalar loop over a data-parallel C* expression. The shape of the C* expression is unidimensional. The parameters are the thread number and the shape size. Memory accesses have been changed into function calls to eliminate cache effects. Figure 12 displays the results.



# thread	Parallel loop size		
	2048	4096	8192
1	0.665	1.334	2.632
2	0.658	1.331	2.660
4	0.670	1.317	2.662
8	0.671	1.341	2.634
16	0.690	1.343	2.683
32	0.717	1.382	2.691

FIG. 12: Normalized alpha and time of a fully parallel code without communication as a function of the number of threads and the shape size on a single-processor machine.

The normalized α values shown in Figure 12 are in the the range $[0.85 \ldots 1.07]$. Thus, the equation is validated. The least squares method gives $\alpha = 2.331\ ms$. As for Adaptor, the cost of using threads is around 2 ms per thread. The thread overhead is still very small.

3.2.3 Overhead introduced by changing pointers into array indexes

This section aims at measuring the overhead induced by using array references instead of pointers. The overhead is mainly due to the recomputation of direct pointers from the array indexes on entering the virtualization loops.

The test program consists of a data-parallel expression iterated by a scalar loop so as to increase the granularity. The parameter is the shape size of the data-parallel variable used in the data-parallel expression. The shape is unidimensional. We compare the UNH C* version without threads to the modified version with threads. Only one working thread is used and the preemption by the message polling thread is disabled.

For a shape size of one, the version with array indexes takes 2085 μs instead of 282 μs for the version with direct pointers. This is the cost of translating array indexes into pointers on entering the loop. This cost could probably be reduced because the generated code is not optimized and contains many redundant indirections. The break of the curve which is observed for a shape size of 32768 is typical of cache effects when the data no longer fit into the processor cache.

As shown in Figure 13, the execution time becomes smaller for the version with array indices as the shape size increases. A look to the assembly code reveals that the processor registers are allocated differently in the two versions. It seems that the C compiler uses less registers on accessing data with the array indexes version than with the direct pointer version. The difference maybe stems from a lack of the C-compiler optimization module. In brief, converting array indexes into direct pointers is costly but this cost is constant with respect

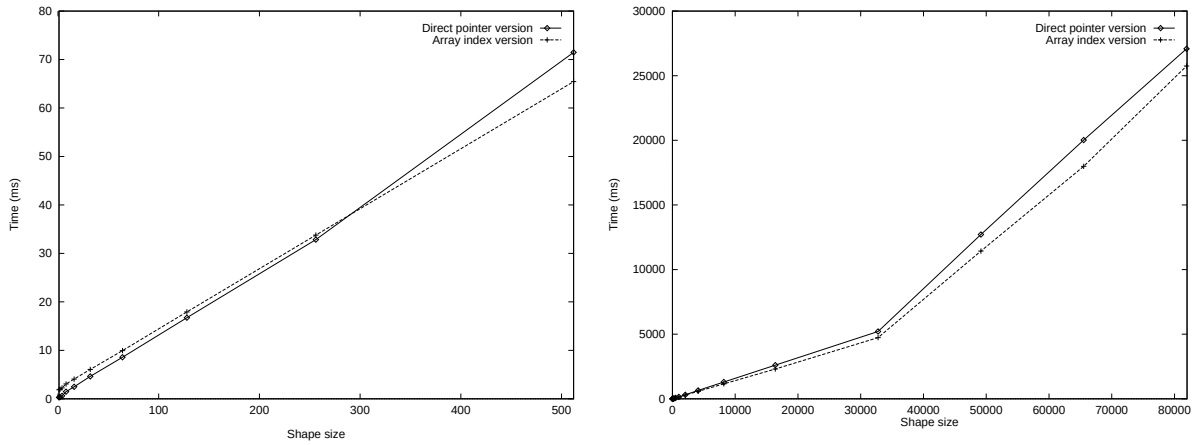


FIG. 13: Time of the array index version and the direct pointer version in function of the unidimensional shape size.

to the shape size. It is very small in percentage for shape size larger than a few hundreds of elements for an element-wise increment of one like in the test program.

3.2.4 Migrating abstract processors

The management of allocated memory is different in the HPF and in the C* runtime libraries. In C*, several data-parallel arrays may be allocated in the same block of memory. On migrating an abstract processor, the data are packed according to these blocks. On the other hand, additional information about the respective shapes of the arrays have to be added and managed. Figure 14 displays two views of the same experiment. An abstract processor is migrated from a node to another and then back. The parameters are the number of arrays and the total size of the arrays.

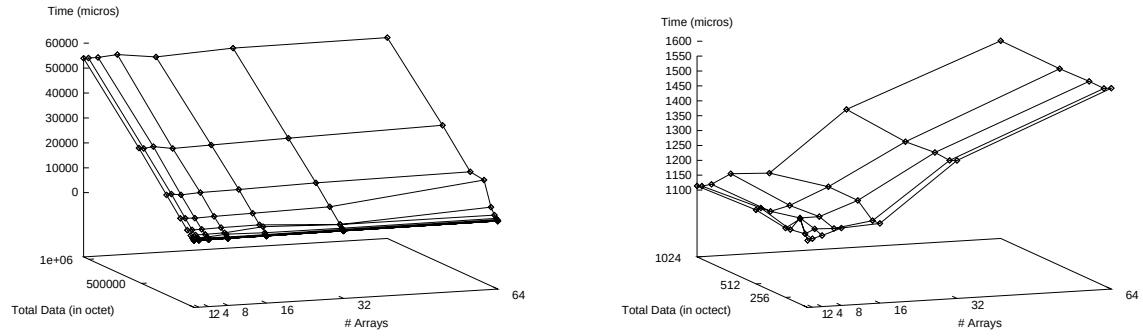


FIG. 14: Time needed to migrate an abstract processor as a function of the total volume of arrays and the number of arrays. The right figure is a zoom of the left figure. Experiments done with TCP over a Myrinet network.

Except for small arrays, the curve can be approximated as a plan. Its equation is of the form $t = \alpha \times n_d + \beta \times n_a + \gamma$ where t is the time, n_d is the cumulated size of all arrays (in kB) and n_a the number of arrays. The least squares method yields $\alpha = 59.82$, $\beta = 53.57$ and $\gamma = -973$ when t is expressed in microseconds. The data bandwidth is 19.1 MB/s. The bandwidth for the data is a little lower than for Adaptor because many additional internal data have to be sent. For example, the state of the C* abstract

processors have to be sent. β is lower than for Adaptor because we have a smaller number of different data to send though they are larger. This saves calls to the pack and unpack functions and also saves computation to rebuild the global data at the destination process.

4 Conclusion

We have described in this paper a modified implementation of the Adaptor HPF and the UNH-C* compilers which integrates a multithreaded runtime library based on the PM2 programming environment. Usual data-parallel compilers use a *coarse-grained* approach toward code generation : one *abstract processor* is mapped to each physical node of the target architecture. In contrast, we propose to use a *medium-grained* approach, in which several abstract processors are mapped to each node. Abstract processors are mapped onto migratable threads. The *dynamic thread migration facility* of PM2 allows us to move threads between nodes to dynamically balance the load.

Our implementation has left unchanged the core of both Adaptor and UNH-C* compiler. Only the runtime libraries have to be extensively modified. Several threads have to share a common communication interface within a process. Intra-thread communications should be handled in different ways depending on whether the partners live in the same process or not. On a thread migration, all the internal tables have to be adjusted.

The experiments we have presented show that the overhead induced by managing the threads is quite small compared to the new opportunity for load balancing. This makes this solution a quite attractive alternative to sophisticated static distribution directives, as proposed in HPF 2 for instance. Yet, addressing the unnecessary replication of the scalar code in all threads necessitates a modification of the internal compilation phase. We are currently working on this aspect : the abstract processors should now be able to share the scalar variables, instead of replicating them.

This work is now being extended into two directions. The first one consists in integrating into the existing data-parallel compilers a form of sharing for scalar variables. The second one if to improve the migration facility so as to guarantee that the abstract processes will be relocated at the *same address* in the virtual space. Then, all pointers remain valid, and our method can adapt any particular use of pointers within the code generator.

Références

[1] Françoise André. A Multi-Threads Runtime For The Pandor e Data-Parallel Compiler. Technical Report 986, IRISA, February 1996. available at URL : http ://www.irisa.fr/EXTERNE /bibli/pi/pi96.html,.

[2] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10) :37–47, October 1995.

[3] Nannette J. Boden, Danny Cohen, Robert E. Fe lderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-K ing Su. Myrinet : a gigabit-per-second local area network. *IEEE-Micro*, 15(1) :29–36, February 1995.

[4] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine : an efficient and portable communication interface for multithreaded environments. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)*, pages 240–247, ENST, Paris, France, October 1998. IFIP WG 10.3 and IEEE. Preliminary electronic version available as [5].

[5] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine : an efficient and portable communication interface for multithreaded environments. Research Report RR1998-26, LIP, ENS Lyon, France, May 1998. Extended version of [4]. Also published as Research Report RR-3459, INRIA Rhône-Alpes.

[6] Thomas Brandes. ADAPTOR (HPF compilation system), developed at GMD-SCAI. available at URL `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[7] B. Chapman, H. Zima, and P. Mehrotra. Extending HPF for advanced data-parallel Applications. *IEEE Parallel and Distributed Technology*, 2(3) :59–70, 1994.

[8] M. Christaller, J. Briat, and M. Rivière. Athapascan-0 : Concepts structurants simples pour une progra mmation parallèle efficace. *Calculateurs Parallèles*, 2(7) :173–196, 1995.

[9] M. Cosnard and E. Jeannot. Automatic coarse-grained parallelization techniques. In Grandinetti and Kowalik, editors, *NATO workshop : Advances in High Performance Computing*. Kluwer academic Publishers, 1997.

[10] DPCE Subcommittee. Data-parallel C extensions. Technical Report 1.6 X3J11/94-080, Numerical C Extensions Group of X3J11, 1994.

[11] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessor systems. *Computer*, 28(12) :27–38, 1995.

[12] T. Fahringer, M. Haines, and P. Mehrotra. On the utility of threads for data parallel programming. Research Report 95-35, ICASE, NASA Lamgley Research Center, May 1995.

[13] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal Parallel Distr. Computing*, 37 :70–82, 1996.

[14] Roy Friedman, Maxim Goldin, Ayel Itzkovitz, and Assaf Schuster. Millipede : Easy parallel programming in available distributed environments. *Software - Practice and Experience*, 27(8) :929–966, 1997.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skellum. A high performance, portable implementation of the MPI messa ge passing interface standard. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1996.

[16] P. J. Hatcher. UNH C*. Available at URL `http://www.cs.unh.edu/pjh/cstar/cstar.html`.

[17] HPF Forum. *High Performance Fortran Language Specification*. Rice University, Texas, October 1996. Version 2.0.

[18] Ravi B. Konuru, Steve W. Otto, and Jonathan Walpole. A migratable user-level process package for PVM. *J. of Parallel and Distributed Computing*, 40(1) :81–102, 1997.

[19] A. Müller and R. Rühl. Extending High Performance Fortran for the support of unstructured computations. In *Proc. 9th ACM Int. Conf. on Supercomputing*, pages 127–136, July 1995.

[20] Myricom. Myrinet link and routing specification. Available at URL `http://www.myri.com/myricom/document.html`, 1995.

[21] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Univ Lille 1, France, January 1997. Available at URL `http://www.lifl.fr/~namyst/these.html`.

[22] R. Namyst and J.-F. Mehaut. PM2 : Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)*, pages 279–285. Elsevier Science Publishers, September 1995.

[23] S. Orlando and R. Perego. SUPPLE : an efficient run-time support for non-uniform parallel loops. Research Report CS-96-17, Dip. di Matematica Applicata ed Informatica, Universita Ca'Foscari di Venezia, December 1996.

[24] Christian Perez. Load balancing HPF programs by migrating virtual processors. Research Report RR1996-33, LIP, ENS Lyon, France, October 1996. Also published as Research Report RR-3037, INRIA Rhône-Alpes. Published as [26].

[25] Christian Perez. Utilisation des processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale. Research Report RR1996-09, LIP, ENS Lyon, April 1996. Also published as Research Report RR-3036, INRIA Rhône-Alpes.

[26] Christian Perez. Load balancing HPF programs by migrating virtual processors. In *Second Intl Workshop on High-Level Programming Models and Supportive Environments (HIPS'97)*, pages 85–92, Geneva, Switzerland, April 1997. IEEE. Extended electronic version available as [24].

[27] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, and M. Weeks. SunOs 5.0 multithreaded architecture. In *Proc. Winter 1991 USENIX Conf.*, pages 65–79, 1991.

[28] Rice University. The D system–tools for machine independent data-parallel programming. Available at URL `http://softlib.rice.edu/fortran-tools/DSystem/DSystem.html`.

[29] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference*. MIT Press, 1995.

[30] Andrew Sohn, Mitsuhisa Sato, Namhoon Yoo, and Jean-Luc Gaudiot. Data and workload distribution in a multithreaded architecture. *J. Parallel and Distr. Comp.*, 40(2) :256–264, 1997.

[31] Thinking Machines Corp., 245 First St., Cambridge, MA 02142. *C* programming guide*, November 1990. Version Number 6.0.

[32] B. Tourancheau and L. Prylli. BIP messages. Available at URL `http://lhpca.univ-lyon1.fr/bip.html`.

[33] T. Yang and A. Gerasoulis. DSC : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transaction on Parallel and Distributed Systems*, 5(9), September 1994.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399