Parallel Processing Letters,

COMPILING DATA-PARALLEL PROGRAMS TO A DISTRIBUTED RUNTIME ENVIRONMENT WITH THREAD ISOMIGRATION*

GABRIEL ANTONIU, LUC BOUGÉ, RAYMOND NAMYST, CHRISTIAN PÉREZ
LIP, ENS Lyon, 46 Allée d'Italie
69364 Lyon Cedex 7, France

ABSTRACT

The compilation of data-parallel languages is traditionally targeted to low-level runtime environments: abstract processors are mapped onto static system processes, which directly address the low-level communication library. Alternatively, we propose to map each HPF abstract processor onto a "lightweight process" (thread) which can be dynamically migrated between nodes together with the data it manages, under the supervision of some external scheduler. We discuss the pros and cons of such an approach and the facilities which must be provided by the multithreaded runtime. We describe a prototype HPF compiling system built along these lines, based on the Adaptor HPF compiler and using the PM2 multithreaded runtime environment.

Keywords: Parallel languages, load balancing, cluster of SMP, distributed multithreaded runtime, thread migration, HPF, Adaptor, MPI

1. Introduction

Data-parallel languages are now recognized as major tools for high performance computing. Considerable effort has been put into 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 [15,14].

Broadly speaking, the compilation method consists in generating an SPMD parallel program from the data-parallel source code. The SPMD processes are typically (plain) Unix processes which communicate with each other by exchanging messages through some standard message-passing library. Each process is in charge of managing a private "slice" of the parallel data structures, and it is in charge of gathering from remote processes the values necessary to update it (*Owner Computes Rule*). The computation cost associated to a given process is thus a function of the data set it has been assigned, and of the locality of this data. Moreover, this amount should be balanced among processes so as to minimize the overall computation time. Therefore, considerable effort has been devoted to designing sophisticated ways to distribute data sets among processes.

Solutions have been proposed in the case of *regular* applications, where the

*This work has been partially supported by the French CNRS ARP Program on Architecture, Networks, Systems and Parallelism, and by the INRIA Cooperative Research Action ResCapA.

amount of work assigned to each process is constant throughout the execution and can be statically estimated. However, it has often been observed that these methods cannot efficiently cope with *irregular* applications, in which the amount of work induced by the individual data sets cannot be predicted at compile-time. Inefficiency has also been reported if the amount of computational power available at the nodes varies in time, even if these variations are partially predictable at compile time. Expensive runtime redistribution of data among processes are then needed, whose specification is left to the programmer. Yet, the load-balancing strategy is deeply embedded in the application design.

In this paper, we propose an alternative compilation technique to address this point. The idea is to use *mobile lightweight processes* instead of heavy, plain processes as a target for the compiler. The compiler still believes that it generates static code as before, but the number of processes is now much larger than the number of physical computing nodes. A runtime library is in charge of monitoring the application behavior. If some imbalance or some lack of locality is detected, then the library determines a new distribution of the lightweight processes among the physical nodes so as to optimize the overall behavior. The processes are moved together with their private data sets. The library is in charge of maintaining the semantic correctness with respect to the application. The dramatic advantage of this approach is that it is completely transparent with respect to the programmer. The load-balancing strategy for an application is now seen as a runtime plug-in, *independent* of the application code.

The problem is of course feasibility! Our proposal is to utilize user-level, POSIX-like threads to implement lightweight processes. Many multithreaded programming environments are available (to cite two of them: Nexus [11] and Athapascan [6]), but the concept of *thread mobility* is still rarely addressed. The PM2 multithreaded programming library [18] developed by our group has been specifically designed to provide the user with such a facility. PM2 threads can be preemptively *migrated* across processes through the communication network, together with their dynamically allocated data. Such a migration is transparent for the thread itself, and it costs as little as 70 μs on our local high-performance PC cluster.

We describe in this paper a prototype compiling environment which has been designed along this idea. It is based on a modified version of the Adaptor HPF compiler [5]. We show that this approach only requires minor changes in the existing compiler. Most changes are located in the runtime library, and are therefore largely independent of the specific compiling schemes. Section describes the general approach. Section provides some insight into the implementation. Section discusses the interest of the approach and the various tradeoffs involved. As a motivating illustration, we report on the performance obtained for a Mandelbrot Function code, a Flame Simulation Kernel code introduced as a *motivating application* in the HPF 2 proposal [13] and an image synthesis code. All three kernel codes exhibit high load imbalance during execution. An extensive presentation of the experimental results can be found in Christian Perez's PhD Thesis [21].

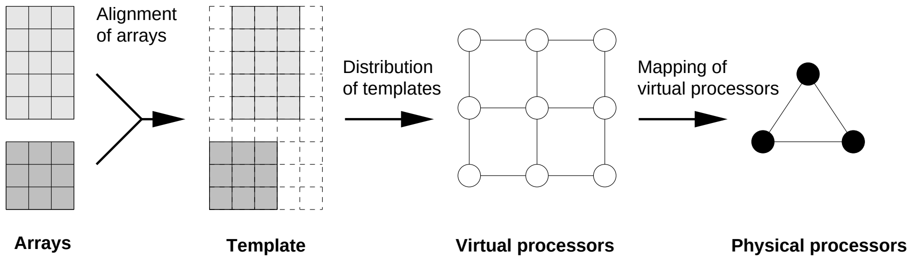

Figure 1: The HPF data placement model. Note that the last level is only partly specified by the current language specification.

2. Generating multithreaded code for data-parallel languages

The High Performance Fortran (HPF) language [10] specifies the allocation of data to processors through a 3-level scheme. In HPF, the parallel data are multi-dimensional arrays. First, arrays are `ALIGN`ed with respect to each other, possibly with respect to optional abstract reference arrays called `TEMPLATE`s. The `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 here. 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 strategies 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 [15, p. 125]) 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 said to be “implementation dependent”.

Most HPF compilers generate 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 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 updating this slice. This is done by exchanging point-to-point messages with the other processes through some standard message-passing communication library. There is thus a perfect overlapping between three distinct notions: 1) the HPF abstract processor as defined by the `PROCESSORS` directive; 2) the system process which implements its behavior and 3) the physical node which runs this process.

However, this simple approach has a number of drawbacks:

Static mapping only. The mapping of abstract processors onto system processes is completely determined at compile-time. At run-time, each system process calls some appropriate function of the underlying communication library (such as `MPI_`

`Comm_rank` in MPI), and the result is used to determine the data sets it is in charge of. This allows to design sophisticated techniques to reschedule the generated code such that the overlapping of communications by computation be maximal. However, 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.

No dynamic adaptativity. The program cannot adapt to dynamic variations in the resources available at each computing node. These resources may consist of computation cycles, but also of cache/memory space, connectivity with other nodes, etc. The only way to take such variation into account is to provide some way to `REDISTRIBUTE` the data among the abstract `PROCESSORS` within the algorithm. Such a redistribution has to be explicitly inserted within the program by the programmer, which is obviously rather inconvenient!

No provision for heterogeneity. The program cannot easily cope with heterogeneous configurations. Sophisticated distribution directives have to be computed for each specific set of resources, and this task is entirely left to the programmer. Examples of such directives are the general block distribution and the user-defined data distributions like in Vienna Fortran [8], Fortran D [22] or Annai [16]. Actually, a major contribution of HPF2 [14] as opposed to the original HPF was to extend the flexibility of the `DISTRIBUTE` directive.

No provision for SMP nodes. Such a compilation scheme is specifically targeted towards current distributed-memory architectures based on single-processor nodes. No provision is made to exploit the power of symmetric multi-processor (SMP) nodes which are bound to become commonplace within the coming years.

We take advantage of the lack of specification for the lowest level of the HPF model to decouple the three notions mentioned above. The goal is to enable abstract processors to react dynamically to the variation of their resources and balance their load. The basic compilation scheme above considers *fixed, coarse-grained* abstract processors. In contrast, we propose to consider *mobile, medium-grained* abstract processors, and to place several of them onto each node, so that they could be *migrated* at run time among the computing nodes. The runtime library is in charge of managing such a dynamic migration. It has to decide when and where to migrate an abstract processor, based on the available monitoring information. It has also to guarantee that such a migration is *semantically correct* with respect to the code generated by the compiler. The load-balancing strategy for a application is seen as a runtime plug-in, *independent* of the application code.

3. A multithreaded runtime with thread migration for an HPF compiler

How can we provide HPF abstract processors with a semantically transparent migration facility? A first idea is to implement abstract processors as data structures which contain all necessary information about the data sets to be managed. The computing nodes are then in charge of "interpreting" these structures and carrying out the real computation. The advantage is that these data structures can be migrated between the nodes with any simple master/slave strategy: a central master dispatches the abstract processors among the computing nodes as soon as they are ready to handle them. This approach is quite flexible, and it could be considered in certain cases. But the common experience is that interpretation may compete with compilation for high performance at the price of a careful and complex design, tightly dependent on the underlying platform.

An alternative idea is to map each HPF abstract processor to a single system process, and to map several system processes onto each computing node. There are operating systems which provide a runtime facility to migrate system processes in a fully transparent way, as in Mosix [2] for instance. Alternatively, one can use some modified version of a traditional communication library such as MPVM [7]. Again, this approach is quite flexible, but migrating a complete system process is rather complex and costly with respect to our problem, and no currently available implementation can do it efficiently enough with respect to our needs.

An intermediate possibility is to map each abstract process onto a user-level, POSIX-like *lightweight process*, also known as a *thread*, instead of a heavy, regular one. A number of multithreaded, distributed programming environment offer a thread migration facility: Millipede [12], PM2 [19], etc. Abstract processors are *compiled* into computation threads, instead of being interpreted: this makes an efficient execution possible. Abstract processors can be migrated efficiently, as only computation threads are moved instead of full system processes (i.e., migrating a thread involves moving its stack, descriptor and private data). We assume the application is executed in a homogeneous environment (same processor, same operating system, same code executed by a process on each node). Since the same code is loaded and allocated at the same virtual address on all nodes, there is no need for any code migration. Also, note that our use of *threads* is rather non-standard, as we do not use their distinguished semantic feature, which is to share memory. Actually, the abstract processors generated by a HPF compiler for a MIMD architecture do not share any data, and all data exchanges are explicitly specified by the compiler through the underlying communication library. This is the reason why we have used the term *lightweight processes* instead of *threads* in the description above.

Retargeting the code generator of an HPF compiler to threads instead of system processes raises a number of challenges. The discussion below is mainly based on our experience with the Adaptor compiler described in Section , but we believe that the conclusions apply to most such compilers.

Transforming processes into threads The original HPF compiler is designed to map abstract processors onto system processes. Each system process is equipped with its own, private memory space and global variables are used. In contrast, threads share a common memory space: local variables are guaranteed not to collide (at least within some reasonable limits), but global variables are shared. A preliminary approach is therefore to modify the compiler to *privatize* all global variables in the generated SPMD code. Also, the `main` function, which is the code executed by each abstract processor, has to be made an ordinary function. Thus, threads that implement abstract processors can call this function after being created.

Adapting the communication library The SPMD code generated by the original HPF compiler is based on an internal communication library similar to MPI. For the sake of portability, the code only uses a restricted number of communication functions: initializing and finalizing the session, sending and receiving a message, possibly a few collective operations such as broadcast, reduction, and synchronization. This library is designed to interconnect system processes, not threads. To make them use threads instead, it has to be made *thread-safe*, that is, to allow several calls to proceed concurrently within the library.

Also, specific wrappers have to be designed so that the messages be correctly sent and received by individual threads within the system processes, even in the presence of migration. The sending abstract processor may have an inaccurate information concerning the real location of the receiving abstract processor. Worst, an abstract processor may migrate while a message is being delivered to it. The communication library has to provide an internal forwarding facility to guarantee correct delivery of messages. Also, it has to guarantee that messages are delivered in order, regardless of the migration of the emitter and/or receiver. Designing such a *migration-safe* communication library is expected to be the most delicate point.

Migrating data along with threads In the SPMD code generated by the compiler, each abstract processor `ALLOCATE`s its private set of data within its own memory. No data may be shared by the abstract processors, as they are supposed to be implemented by stand-alone system processes in the original code generation scheme. As migration should be transparent as far as the generated code is concerned, the data should remain accessible to the abstract processor regardless of its possible migration. A possibility is to rely on some form of *Distributed Shared Memory* (DSM) software layer to provide system-wide accessibility, but this does not provide enough performance in general. An alternative idea is to migrate the abstract processors systematically along with their private data. One can keep track of the allocated data by wrapping the allocation function adequately. Then, the hook functions provided by the migration facility can be used to pack the allocated data together with the thread on the source node, and to unpack them on the destination node, adjusting all pointers on the fly. However, this is not sufficient, as the abstract processors continuously operate on their data, possibly `ALLOCATE`-ing and `DEALLOCATE`-ing them. A better choice is to rely on some *system-aware allocation facility*, which guarantees that all returned virtual addresses be valid upon migration. We have developed such a software library for the PM2 distributed multithreaded system, called `isomalloc` [1], to be described below.

Managing scalar computation The original compiler generates an SPMD code to be replicated within each system process. For efficiency reasons, the scalar part of the data-parallel computation is replicated on each abstract processor, so that potentially costly communication is avoid. If abstract processors are mapped onto threads instead of system processes, this may be questioned: all threads hosted within the same system process could share the (common) result of the scalar computation. Redoing the same scalar computation for each hosted thread is a waste of computing power. However, implementing such a scheme necessarily involves some sort of additional synchronization mechanism, which has to be inserted into the generated code. Unless the code generator is heavily optimized, the number of such local synchronizations may be very high, and the grain of the scalar computations may be too small compared to the synchronization cost. In fact, addressing this problem would require an extensive modification of the code generator to factorize synchronizations. We have not addressed this point further at this time.

4. An implementation within the Adaptor HPF compiler

4.1. The PM2 multithreaded programming environment

General presentation PM2 [19] is a distributed multithreaded runtime system for irregular parallel applications. The main objective of PM2 is to provide a carefully chosen set of basic features on top of which many dynamic load balancing policies are easy to implement. Since PM2 applications may generate a large number of threads with unpredictable lifetimes, the PM2 programming model is based on the concept of *mobile threads*. Each node executes the same code (say, as a Unix process) and multiple threads share the address space and the code of the process. PM2 threads can be migrated *preemptively* from one node to another without any explicit state backup nor global synchronization.

The PM2 threads interact through the *Lightweight Remote Procedure Call* (LRPC) mechanism, which can be made efficiently "migration tolerant". 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 mechanisms such as thread migration or special scheduling functionalities. 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 U-Net) and applications. It provides an interface optimized for *RPC-like* operations which allows zero-copy data transmissions on high-speed networks such as Myrinet or SCI.

Dynamic thread isomigration A PM2 thread may silently migrate from one process to another without any global action whatsoever. Only the origin and the destination processes are involved. The destination process 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.

Migrating a thread consists in moving the thread resources (i.e. its stack, descriptor and private data) from one node to another, where the thread can resume its execution. A difficulty arises as soon as the migrating thread uses pointers.

The migration mechanism must ensure the validity of pointers and must guarantee their safe use after migration. We have solved this problem [1] by designing a mechanism which guarantees that a thread and its data can always migrate while keeping the same virtual addresses on the destination node as on the original node. Our iso-address memory allocator (called `isomalloc`) guarantees that for each dynamic memory allocation carried out by a thread, the returned virtual address range remains available on all nodes, such that iso-address migrations never generate overwriting. Threads may resume their execution on the destination node without any post-migration processing, since all pointers remain valid. Notice that `pm2_isomalloc` and `malloc` are not conflicting: the `malloc` primitive may still be used to allocate memory for non-migratable data.

4.2. A complete implementation within the Adaptor HPF compiler

Adaptor [5] is a public domain HPF compiler developed at the GMD (Germany) by Th. Brandes. It transforms HPF data-parallel programs into SPMD FORTRAN programs with explicit message passing. It provides a suitable platform to experiment with this idea. Adaptor itself consists of two components: `fadapt` and `DALIB`. `fadapt` is a source-to-source translator from HPF to F77 (or F90). `DALIB` (the HPF runtime) handles descriptors for HPF arrays, sections and distributions and also implements the communication routines. We have used the Adaptor version 6.1 of December 1998.

Adapting Adaptor to a multithreaded runtime environment The `fadapt` component has been left unchanged but for the unparsing function. The main FORTRAN `program` is transformed into a `subroutine` and a new main `program` which calls our own initialization subroutine is added.

Substantial modifications have been needed to let the `DALIB` library work on top of a multithreaded environment. All the global variables have been privatized, since several abstract processors may concurrently call runtime routines. Given that these variables have to follow an abstract processor upon migration, functions to transfer them with the migration message have been added to `DALIB`. Also, references to abstract processors are no longer references to system processes, but references to threads. Finally, `DALIB` memory allocation functions that used `malloc` now use `pm2_isomalloc`. It is only a text substitution, since they have the same prototype.

A specific module has been added to the `DALIB` runtime in order to map its generic message-passing interface to the PM2 Remote Procedure Call facility. Each abstract processor owns a mailbox to store messages received by the system process but for which the destination abstract processor has not yet posted a receipt request. This module also manages message forwarding when a message reaches a system process just left by the destination abstract processor. Because of message forwarding, the message order between two abstract processors may be lost. It is recovered by numbering messages, since `DALIB` requires that messages be delivered in order.

Migrating abstract processors Migrating an abstract processor consists in transferring the thread (i.e. stack and internal data structures), its private dynamically allocated memory and some related global variables. PM2 handles thread migration with private data thanks to the iso-address allocator. The global data are migrated by the `DALIB` functions that concatenate them to the migration message.

The interface of the abstract processor migration function is simple. It takes two arguments: the local abstract processor id to be migrated and the process id where to migrate. Various load balancing policies may easily be implemented on top of this building block.

The experiments have been performed on a 8-node cluster of PC consisting of PentiumPro 200Mhz processors with 512 kB of L2 cache and 64 MB of memory. The nodes are connected through a Myrinet network [17]. PM2 is used with the BIP communication library, a low-level protocol for the Gigabit Myrinet network.

We measure an abstract processor migration latency of 1.3 ms and an associated bandwith of 66.3 MB/s. The average latency comes from the fact that an empty abstract processor is around 17 KB because of large arrays related to message forwarding. The bandwidth is very good.

5. Experimental evaluation

In the remainder of this section, we present experimental results obtained from three irregular kernel codes: a Mandelbrot Function code, a Flame Simulation Kernel code and a Dividing-Cubes Algorithm for image synthesis.

All the experiments have been performed on the cluster described in section , using PM2 on top of the MPI/BIP communication library. For this configuration, the one-way communication latency is 25 μs and the bandwidth is 120 MB/s .

5.1. Experiment: Mandelbrot Function

Benchmark description We first present our experimental results with the Mandelbrot code.This code is irregular and embarrassingly parallel: the computation does not generate any message. It is a preliminary test for load balancing strategies.

For such highly irregular and unpredictable codes, work stealing is a efficient yet simple load balancing strategy [3]. Our implementation is quite classical except that it moves abstract processors instead of moving plain data. When a system process is about to run out of active abstract processors (that is, threads), it sends a request asking for active ones to another randomly selected system process. If the target process has more active abstract processors than a user-defined threshold, it preemptively migrates some of them to the requesting process. If it has no abstract processor to let go, a negative answer is issued. In this latter case, the underloaded process retries with another randomly selected target process.

Experiments run with domains that generate unbalanced computation show performance improvement by up to 30 % when run with load balacing. Experiments with more regular domains showed that differences are very small, about 1 %.

Distribution Mode	Initial pattern irregularity			
	None	Low	Average	High
Original ADAPTOR runtime				
`BLOCK`	61	98	246	427
`CYCLIC`	129	136	128	142
Redistribution	100	100	100	100
Modified ADAPTOR runtime, 128 abstract processors				
`BLOCK`, without load balancing	62	73	141	261
`BLOCK`, with load balancing	68	80	132	209

Figure 2: Normalized times for various distributions on the Flame Simulation Kernel benchmark on a 8-node cluster of PC. The grid contains 1600×1600 elements. The underlying communication protocol is MPI/BIP.

In conclusion, the Mandelbrot application shows the suitability of abstract processor migration as a basic mechanism for load balancing. No modification has been needed at the program level. The precise specification of the load balancing strategy is entirely encapsulated in the runtime library.

5.2. Experiment: Flame Simulation Kernel

Benchmark description We have validated our modified Adaptor compiler on the Flame Simulation Kernelmentioned as one of the *motivating applications* in the HPF 2 proposal [13]. The program performs a detailed time-dependent, multi-dimensional simulation of hydrocarbon flames in two phases.

This Flame Simulation Kernel is known to be interesting because its two phases have different requirements. The first phase is well suited for a `BLOCK` distribution of data (it requires neighborhood communication and the computations are regular), whereas the second phase calls for an irregular distribution of data.

- A global `BLOCK` distribution minimizes the communications in the first phase. But, in the second phase, the application is exposed to the effects of the load imbalance.
- A global `CYCLIC` distribution has opposite effects, compared to the `BLOCK` distribution. The load is better balanced in the second phase, but the communications are expensive during the first phase: for every iteration, each node has to send all its data to its 2 (resp. 4) neighboring nodes if one (resp. two) dimension(s) is (are) cyclicly distributed.

Note that the `CYCLIC` distribution has an additional dramatic drawback: a huge waste of memory. To receive data from the neighboring nodes, a node has to allocate 2 (resp. 4) arrays as large as the main array. So, only 33 % (resp. 20 %) of the memory on each node can be used for local data.

A possibility is thus to run each phase with the best suited static data distribution. Then, two data redistributions have to be performed for each iteration of the sequential loop. For the Flame Simulation Kernel, the convection phase is run with a `BLOCK` distribution whereas the reaction phase is run with a `CYCLIC` distribution. This may improve performance if the cost of the redistributions is lower than the expected improvement within each phase. It turns out that this is the case in our experiments. Yet, observe that this method requires the explicit insertion of redistribution directives into the source code from the programmer. Doing such with a reasonable chance of success may only be considered if the programmer enjoys a definite expertise on the behavior of the program. Otherwise, it is quite likely to produce opposite results!

Our work enables a new approach: compile the program with the simplest distribution, and let an external plug-in load balancer redistribute the abstract processors among the nodes on the fly. We choose here the `BLOCK` distribution, which provides best performance for the first phase, and which is the easiest to compile. To handle load imbalance in the second phase, abstract processors are migrated from overloaded nodes to underloaded nodes. Since the load distribution is unknown, migration is managed according to a work stealing load balancing strategy. We have used the same load balancing module as for the Mandelbrot application.

Performance results We have considered four data sets, which differ by their degrees of irregularity for the reaction phase. They range from a regular to a highly irregular computational patterns. The experimental results are displayed on Figure 2. For each data set, we first show the execution times respectively obtained with the `BLOCK` and the `CYCLIC` distributions. The third row contains the times observed with the redistribution-based code. These three codes are compiled with the original Adaptor compiler. Then, we show the times obtained with our modified Adaptor compiler. The data are `BLOCK`-distributed among 128 abstract processors on 8 nodes, respectively without and with activating the external load balancing facility. The results of all experiments have been normalized according to the redistribution-based version.

One can notice that the `CYCLIC` distribution incurs severe overhead for all data sets due to communication. Despite the two redistributions per iteration, the redistribution-based code shows better performance than the `CYCLIC` distribution. However, its performance, just like in the case of the `CYCLIC` distribution, is not very good for the regular and the weakly irregular situations. The `BLOCK` distribution performs well when the computational cost is regular but its performance drops a lot when it becomes irregular. Introducing several abstract processors per node allows communication to be overlapped by computation. It leads to performance better than for the `BLOCK` distribution. Load balancing the application by migrating abstract processors according to a work stealing algorithm leads to good performance in *all* circumstances, even though they are not the best achievable ones. When the load is balanced, this strategy does not generate any significant overhead. When the load is not balanced, performance gets improved significantly.

It should be strongly emphasized that these experimental results are obtained without modifying the original HPF code: as opposed to the insertion of explicit redistribution directives, no expertise at all about the behavior of the program is required. Also, the external load balancing facility is completely independent of the application, and it may be re-used without any modification in other circumstances. Conversely, various facilities could be successively tried out on the same application without any modification of the source code. In our approach, tuning load balancing boils down to selecting an option on the compiler command line!

5.3. Experiment: The Dividing-Cubes Algorithm

```
    SUBROUTINE DividingCubes(vox, nx, ny, nz, ...)

    INTEGER nx, ny, nz
    INTEGER vox(1:nx, 1:ny, 1:nz)
!HPF$ DISTRIBUTE vox (BLOCK, BLOCK, BLOCK)
    INTEGER x, y, z, i, j, k

!HPF$ INDEPENDENT
       DO x, y, z
        IF (intersection(vox(x, y, z))) THEN
           cube(0:A, 0:B, 0:C) = F(vox, x, y, z, A, B, C)
           DO i, j, k
              IF (intersection(cube(i, j, k))) THEN
                  CALL interpolate_gradient()
              END IF
           END DO i, j, k
          END IF
       END DO x, y, z
```

Figure 3: The skeleton of the Dividing-Cubes Algorithm

Benchmark description The third benchmark application is the Dividing-Cubes Algorithm [9]. It is used in medical imagery to extract surfaces from 3D images. The algorithm is presented in Figure 3.

Nicod and al. [4,20] studied this application and proposed a method to statically estimate the load distribution, and to generate from this estimation a data distribution which balances the load accurately. It is therefore an interesting challenge for our approach.

We compare Nicod's solution to a direct solution consisting in using our work stealing module. Nicod's solution raises two problems. First, it must be possible to compute a load estimation, which takes some effort. Second, very irregular data distributions are needed to precisely exploit the load estimation. In contrast, the use of the work stealing module is immediate. However, it is expected to be less efficient. The loss of efficiency induced by our method compared to its essentially free implementation is thus a sensible measure.

Distribution Mode	3D source image	
	CThead	Wrist
Optimal Time: sequential time/4		
`BLOCK`	100	100
`BLOCK`, including load estimation	109	107
Original ADAPTOR runtime		
`BLOCK`	152	162
Modified ADAPTOR runtime, 128 abstract processors		
`BLOCK`, without load balancing	188	148
`BLOCK`, with load balancing	117	121

Table 1: Normalized execution times for two images and for various data distributions for the Dividing-Cubes Algorithm. Programs are run on 4 nodes. The CThead image size is $256 \times 256 \times 113$ and the Wrist image size is $512 \times 19 \times 512$.

Performance results Table 1 reports experimental measurements obtained with our HPF version of the Dividing-Cubes Algorithm. These results have been obtained on a 4 node configuration. The first row reports the sequential time divided by 4. As it corresponds to our optimal time, it is normalized to 100. The second row is obtained like the first row but it includes the load estimation code. Roughly, it costs a little less that 10 % to estimate the load in Nicod's method. The third row shows the execution time for the HPF version with a `BLOCK` distribution. Load imbalance generates a high overhead. The timings of the last two rows are obtained with our modified HPF compiler for 128 abstract processors, respectively without and with load balancing.

It is unclear why adding many abstract processors increases the execution time for the CThead image. The time should have decreased as for the Wrist image because of the better data locality. However, the interesting remark for both images is that load balancing helps a lot. It yields performance at only 20 % of the optimal execution times. If we compare them to the version that contains a load estimation, the gap is only around 10 %. Observe that we do not take into account the data redistribution nor the potential overhead generated in data accesses coming from an irregular data distribution. Also, we have been able to obtain such performance without any modification to the original HPF code, nor any extra knowledge of the algorithm. A general-purpose load balancing facility has been transparently plugged into the HPF runtime. More elaborate strategies can be designed for this problem, possibly integrating Nicod's estimators: much better performance may then be expected.

6. Conclusion and perspectives

We have demonstrated that the lack of specification for the lowest level of High Performance Fortran offers an interesting opportunity to design distributed multi-threaded runtime layers. They provide the external world with a hook function for the dynamic migration of an abstract processor from one node to another in a fully transparent way. Any kind of external load monitoring facility can then be *plugged into* the runtime to make the appropriate migration decisions. Observe that there is no reason why these load monitors should exclusively concentrate on the computing load. They may also consider the free memory/storage space, or the connectivity bandwidth, as demonstrated in the Mosix system.

This approach involves only minimal modifications to the code generation layers of the compiler, at least as long as the replication of scalar computation is not at stake. In simple benchmarks, we could observe considerable improvements with respect to the original execution times without any modification of the program or the code generation. No knowledge whatsoever of the application and/or its data have been used. This should be compared with the explicit insertion of redistribution directives in case of highly irregular data, which yields better results at the expense of a very precise analysis of the program behavior.

These encouraging results were obtained using a simple work-stealing strategy, without any consideration to the structure of the application. If some cooperation could be set up between the load-balancing strategy and the application and/or its data set, then much better improvements may be expected.

Even though our benchmarks are simple, these results seem promising. Mapping abstract processors onto multiples threads instead of single processes already improves the performance! In fact, the cache locality of the virtualization loops gets better, and this effect dominates the replication of the scalar computation and the overhead of thread management. This idea can be probably be applied to other data-parallel languages as well, as demonstrated in detail in Christian Perez's PhD Thesis [21].

Acknowledgements

This work would not have been possible without the help of Phil Hatcher and Thomas Brandes.

[1] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Computer Science*, pages 496–510, San Juan, Puerto Rico, Apr. 1999. Springer-Verlag. Workshop held as part of IPPS/SPDP 1999, IEEE/ACM.

[2] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance. *Cluster Computing, Journal of Future Generation Computer Systems*, 13(4-5):361–372, Mar. 1998.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa-Fe, NM, Nov. 1994.

[4] F. Boumghar, S. Miguet, and J.-M. Nicod. Complexity of discrete surface in the dividing-cubes algorithm. In S. Miguet, A. Montanvert, and S. Ubéda, editors, *Discrete*

Geometry for Computer Imagery (DGCI'96), Lecture Notes in Computer Sciences, pages 269–280. Springer Verlag, Nov. 1996.

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

[6] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: Efficiency for irregular problems. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lect. Notes Comp. Science*, pages 591–600. Springer-Verlag, 1997.

[7] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. Adaptive load migration system for PVM. In *Supercomputing '94*, pages 390–399, Washington, D.C., Nov. 1994.

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

[9] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):64–71, May 1998.

[10] H. P. F. Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, Nov. 1994. Version 1.1.

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

[12] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy parallel programming in available distributed environments. *Software: Practice and Experience*, 27(8):929–965, Aug. 1997.

[13] HPF Forum. HPF-2 scope of activities and motivating applications, Nov. 1994. Ver. 0.8.

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

[15] C. Koelbel, D. Loveman, R. Schreiber, J. G.L. Steele, and M. Zosel. *The High Performance Fortran handbook.* MIT Press, 1994.

[16] 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.

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

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

[19] R. Namyst and J.-F. Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, Sept. 1995.

[20] J.-M. Nicod. *Extraction de surfaces en imagerie médicale : approches parallèles.* Thèse de doctorat, LIP, ENS Lyon, France, Jan. 1997.

[21] C. Perez. *Compilation des langages à parallélisme de données : gestion de l'équilibrage de charge par un exécutif à base de processus légers.* Thèse de doctorat, ENS Lyon, France, LIP, Dec. 1999. Written in French. To appear.

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