Compiling Data-parallel Programs to A Distributed Runtime Environment with Thread Isomigration
Gabriel Antoniu, Luc Bougé, Raymond Namyst, Christian Pérez

To cite this version:
Gabriel Antoniu, Luc Bougé, Raymond Namyst, Christian Pérez. Compiling Data-parallel Programs to A Distributed Runtime Environment with Thread Isomigration. The 1999 Intl Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA ’99), Technical Session on parallel and distributed languages: mechanisms implementations, and tools, 2000, Las Vegas, NV, United States. pp.1756-1762. inria-00563794

HAL Id: inria-00563794
https://inria.hal.science/inria-00563794
Submitted on 7 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Compiling data-parallel programs to a distributed runtime environment with thread isomigration* 

Gabriel Antoniu†  Luc Bougé†  Raymond Namyst†  Christian Perez†

Abstract

Traditionally, the compilation of data-parallel languages is targeted to low-level runtime environments: abstract processors are mapped onto static system processes, which directly address the low-level IPC library. Alternatively, we propose to map each HPF abstract processor onto a “lightweight process” (thread) which can be freely 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 compiler built along these lines, based on the Adaptor HPF compiler and the PM2 multithreaded runtime environment.

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

Citation: This paper is a reprint of a paper presented at the 1999 Intl Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA ’99). Please, refer to the original paper [1] in all citations.

1 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 [10, 9].

HPF 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 ALIGNed, possibly with respect to optional abstract reference arrays called TEMPLATES. Mutually ALIGNed 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 DISTRIBUTed 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 [10, 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 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 updating this slice. This is done by exchanging point-to-point messages with the other processes through some standard message-passing communication library (MPI in the case of Adaptor). 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;
3. the physical node which runs this process.

---

*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.
Data-parallel array variables

ALIGN

TEMPLATE (optional)

DISTRIBUTE

PROCESSORS (Abstract processors)

(Only partly specified in HPF)

Nodes (Physical processors)

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

However, this approach has a number of drawbacks. First, the mapping of abstract processors onto system processes is 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 slice of data it is in charge of.

Second, 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 or connectivity with other nodes. The only way to take such variation into account is to provide some way of REDISTRIBUTEing the data among the abstract PROCESSORS within the algorithm, which is obviously rather inconvenient!

Finally, 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.

In a previous paper [4], we have proposed to combine language-level static distribution directives together with runtime-level dynamic load balancing methods. The idea is to transparently migrate HPF abstract processors at runtime among the computing nodes. For this purpose, abstract processors are mapped onto threads within a suitable multi-threaded runtime environment which provides transparent, preemptive thread migration. An external load balancing strategy is then used to monitor the threads and decide upon their migration.

In this paper, we discuss the various trade-offs involved in this new approach, the technical problems raised by its implementation and the performance improvements which may be expected from it. As a motivating illustration, we report on the performance obtained for a flame simulation code cited as a motivating application in the HPF 2 proposal [8]. The program performs a detailed time-dependent, multi-dimensional simulation of hydrocarbon flames in two phases. Considerable improvements compared to the original BLOCK or CYCLIC distributions have been observed. More details are given in Section 3.3. An extensive presentation of the experimental results can be found in the PhD Thesis of Christian Perez [13].

2 A multithreaded runtime with thread migration for HPF

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 dynamically react to the variation of their resources and balance their load.

A first idea is to implement abstract processors as data structures which contain all necessary information about the data slices to be managed. A number of computing nodes are then in charge of “interpreting” these structures and carry 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 cannot compete for high performance in general.

An alternative idea is to migrate the system processes. The operating system may provide some runtime facility to do it in a fully transparent way, as in Mosix [3] for instance. Or one can use some modified version of traditional communication library such as MPVM [6]. Again, this approach is quite flexible, but migrating a complete system process between nodes is rather complex 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 abstract processes onto special, lightweight processes, also known as threads,
instead of heavy, regular ones. A number of multithreaded, distributed programming environment offer a migration facility: Millipede [7], PM2 [12], 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. Note that this use of threads is rather non standard, as we do not use their distinguished semantic feature, which is to share memory. HPF abstract processors do not share any data in the original HPF model, and all data exchanges have to be 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 above. Such a design decision raises a number of challenges.

**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 variable are guaranteed not to collide (at least within some reasonable limits), but global variables are shared. A preliminary work is therefore to modify the compiler to privatize all global variables in the generated SPMD code, and to make the main function an ordinary function to be called by an external harness.

**Adapting the communication library** The SPMD code generated by the original compiler is usually based on some traditional communication library such as 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 broadcasting, reducing, and synchronizing. Such libraries are designed to interconnect system processes, not threads. To make them use threads instead, they have to be thread-safe, that is, allow several calls to proceed concurrently within the library. Also, specific wrappers have to be designed so that the messages are correctly sent and received by individual threads within the system processes, even in the presence of preemptive migration. This is expected to be the most difficult point.

**Managing scalar computation** The original compiler generates a SPMD code to be replicated on each system process. For efficiency reasons, the scalar part of the data-parallel computation is replicated by each abstract processor. If abstract processors are mapped onto threads instead of system processes, this may be questioned: all threads located within the same system process could share the (common) result. 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.

### 3 An implementation within the Adaptor HPF compiler

**3.1 The PM2 multithreaded programming environment**

**General presentation** PM2 [12] 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. Threads can be preemptively migrated from one node (say, a Unix process) 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
efficiently made “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 features such as thread migration or reduced preemption. Its implementation is currently available on the following architectures: Sparc, i86, 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). It provides an interface optimized for RPC-like operations that 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 without any global action whatsoever. Only the origin and the destination process does not need to wait for the incoming thread: it can proceed 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.
The interface of the abstract processor migration function is simple. It takes two arguments: the local abstract processor id to be (preemptively) migrated and the process id where to migrate. Various load balancing policies may easily be implemented on top of this building block.

Figure 2 compares the cost of migrating an abstract processor respectively using the `pm2_isomalloc` facility and the original `malloc` facility to allocate memory dynamically. The test program consists in migrating a HPF abstract processor (a thread) forth and back between two nodes. The time is the average of 200 one-way migrations. Both memory managers use some (software) cache mechanisms. For the `pm2_isomalloc` version, we have also reported the performance without any memory cache.

The experiments have been performed on a 8-node cluster of PC built up out of PentiumPro 200Mhz processors with 512 kB of L2 cache and 64 MB of memory. The nodes are connected through a Myrinet network [11]. PM2 is used with the BIP communication library, a low level protocol for the Gigabit Myrinet network. In this configuration, the latency of a one-way communication is 10 µs and the bandwidth is 125 MB/s.

If no memory cache is used, the `pm2_isomalloc` version has globally the same performance as the `malloc` version. When using the software memory cache, the `pm2_isomalloc` version is faster. The difference between the performance of the two versions are mainly due to memory cache management. Since the `pm2_isomalloc` version of the runtime allows the free use of pointers whereas the `malloc` version does not, we can conclude that the `pm2_isomalloc` version of the Adaptor runtime provides preemptive abstract processor migration at no extra cost.

3.3 Experiment : Flame Simulation

**Benchmark description** We have validated our modified Adaptor compiler on a Flame Simulation code (see Figure 4) mentioned as one of the motivating applications in the HPF 2 proposal [8]. The program performs a detailed time-dependent, multi-dimensional simulation of hydrocarbon flames in two phases. This Flame Simulation kernel code 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 needs an irregular distribution of data.

- A global BLOCK distribution minimizes the communications in the first phase. But, in the second phase, the application suffers the effects of the load imbalance.
- A global CYCLIC distribution has opposite effects, compared to the BLOCK distribution. The load is quite well 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 are to be performed for each iteration of the sequential loop. For the Flame Simulation, 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 it 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 of the behavior of his program. Otherwise, it is quite likely to produce opposite results!

Our approach enables a new approach: compiling the program with the simplest distribution, and letting an external plug-in facility redistribute the abstract processors.
among the nodes on the fly. We choose here the 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. When a system process is going to run out of active abstract processors, it sends a request asking for active ones to another randomly chosen system process. If the target process has more active abstract processors than a user-defined threshold, it migrates preemptively some of them to the requesting process. If it has no abstract processor to let, a negative answer is issued. In this case, the source process goes on with another randomly chosen target process.

Performance results The experiments have been performed on the cluster described in section 3.2, using PM2 on top of the MPI/BIP communication library. For this configuration, the latency is 25 $\mu$s and the bandwith is 120 MB/s.

We have considered four data sets, which differ in their degrees of irregularity for the reaction phase. They range from a regular to a highly irregular computational pattern. The experimental results are displayed on Figure 3. For each data set, we first present the execution times obtained with the BLOCK and the CYCLIC distributions. The third row contains the result of the redistribution-based code. These three codes are compiled with the original Adaptor compiler. Then, we present 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 see that the CYCLIC distribution incurs severe overhead for all data sets due to communication. Despite of two redistributions per iteration, the redistribution-based code has better performance than the CYCLIC distribution. However, its performance, as for the CYCLIC distribution, is not very good for the regular and the low irregular situations. The BLOCK distribution performs well when the computational cost is regular but its performance drops a lot when the 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 processor according to a work stealing algorithm leads to good performance in all circumstances. When the load is balanced, this strategy does not generate any significant overhead. When the load is not balanced, performance gets significantly improved.

It should be strongly emphasized that these experimental results are obtained without modifying the original HPF
code: in opposite 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!

4 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 multithreaded runtime layers. They provide the external world with a hook function to dynamically migrate 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 a simple benchmark, 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 benchmark is admittingly 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 [13].

References


