Load Balancing HPF Programs by Migrating Virtual Processors

Christian Pérez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Load balancing HPF programs by migrating virtual processors

Christian Perez

N ° 3037

Novembre 1996

THÈME 1

ISSN 0249-6399

INRIA
RHÔNE-ALPES

Load balancing HPF programs by migrating virtual processors

Christian Perez

Thème 1 — Réseaux et systèmes
Projet ReMap

Rapport de recherche n° 3037 — Novembre 1996 — 20 pages

Abstract: This paper explores the integration of load balancing features in the data parallel language HPF targeting semi-regular applications. We show that the HPF virtual processors are good candidates to be the unit of migration. Then, we compare 3 possible implementations and show that threads provide a good tradeoff between efficiency and ease of implementation. We finally describe a preliminary implementation. The experimental results, obtained with the Gaussian elimination with partial pivoting are promising.

Key-words: Data parallel languages, HPF, compilation, multi-thread environment, thread migration.

(Résumé : tsvp)

This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS.

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

Équilibrage de charge de programmes HPF par la migration de processeurs virtuels

Résumé : Ce papier étudie l'intégration dans le langage HPF d'un équilibrage de charge visant les applications semi-régulières. Nous montrons que les processeurs virtuels du langage sont de bons candidats pour être l'unité de migration. Nous comparons alors 3 implémentations possibles et il apparaît que les processus légers représentent un bon compromis entre l'efficacité et la facilité d'implémentation. Nous décrivons ensuite notre implémentation préliminaire. Les résultats expérimentaux obtenus avec l'élimination de Gauss avec pivot partiel sont prometteurs.

Mots-clé : Langages à parallélisme de données, HPF, compilation, environnement d'exécution *multi-thread*, migration de processus légers.

Contents

1 Introduction

HPF-like data parallel languages seem to have found an increasing popularity thanks to the continuous improvement of the compilers. Efficient at the beginning on a small set of very regular programs, the current compilers include a lot of optimizations. For example, they can overlap communications by computations by introducing pipelined techniques [5]. They can also optimize irregular communications when an irregular pattern is used many times [16].

However, the load balancing issues are seldom taken into account. Some papers [3, 7] propose irregular data distributions in order to provide mechanisms that allow the user to balance statically the load. More recently, HPF 2.0 includes now general block distributions and irregular distributions as approved extensions [9]. Yet, this is a redistribution notion which has not all the dynamic load balancing features. One of the advantages of a dynamic load balancing system over irregular data redistributions is that it is almost transparent for the programmer. The user has only to give some knowledge to the compiler if the compiler does not succeed extracting the properties of the program. So, the code remains basically the same and the user does not have to bother too much with load balancing problems. The compiler can introduce calls to the load balancing module according to some user specifications or can do it automatically. The portability and the maintenance of the code are improved. Moreover, a dynamic load balancing system allows an execution to be efficient on a static and/or dynamic heterogeneous environment. The load balancing module may be interfaced with the operating system to share informations about the application needs and the system constraints.

The aim of this paper is to show how the HPF definition allows the notion of virtual processor to be extended in order to constitute the unit of a load balancing system. The second goal is to show that the threads are good candidates to encapsulate migrating virtual processors as the migration of threads is becoming available.

The rest of this paper is divided as follows. Section 2 sums up the HPF definitions concerning alignment and distribution. Section 3 deals with the different levels of HPF data mapping where irregularity can be introduced. We compare 3 solutions of implementation in Section 4. In Section 5, we review some implementation issues based on our experiment. Preliminary experimentations are described in Section 6.

2 The HPF model

High Performance Fortran [8, 9], abbreviated as HPF, is a data parallel language based on Fortran. HPF 1.0 was based on Fortran 90 and HPF 2.0 on Fortran 95. A very important feature of this language is that it defines a three level mapping of data, which represents in fact arrays. The first level is an alignment level. Arrays are aligned relative to one another or to a `template`, which is an abstract reference array. The language provides the `ALIGN` directive for this purpose. Then, arrays are distributed onto abstract processors using the `DISTRIBUTE` directive of HPF. This distribution is defined onto a rectilinear arrangement of virtual processors. The last level is declared to be an optional implementation-dependent directive ([8], p. 21). It consists in mapping virtual processors onto physical processors. Currently, most compilers map exactly one virtual processor onto one process. Processes are assumed to be one per physical processor. So, they use the virtual onto physical processor mapping directive very poorly. A schema of the 3 levels of mapping and distribution of the HPF model is displayed on Figure 1.

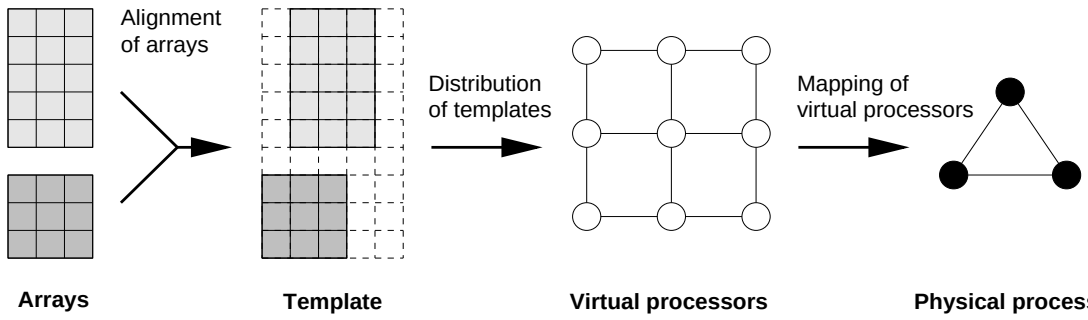
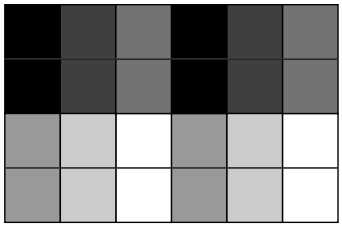


Figure 1: *The HPF model: arrays are aligned with a template which is distributed onto abstract (or virtual) processors. These processors are mapped onto physical processors.*

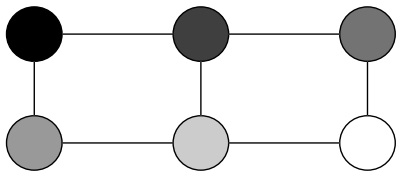
The `ALIGN` directive: Usually, the alignment phase aims at minimizing the communications by forcing elements of different arrays to be aligned at the same location as some elements of the template. Whatever the distribution is, all the elements of arrays aligned with an element of the template are guaranteed to be mapped onto the same processor.

The `DISTRIBUTE` directive: The distribution phase deals with spreading data onto virtual processors. The basic distributions are the block (`block`) and cyclic (`cyclic(1)`) distributions which can be seen like special cases of the block-cyclic distribution (`cyclic(n)`). The distribution of a k-dimensional array representing a `template` onto a l-dimensional array of virtual processors, with l smaller than k, is specified dimension by dimension. If l is strictly smaller than k, the remaining dimensions are replicated onto all the virtual processors.

The `PROCESSOR` directive: This directive defines a user-declared Cartesian mesh of abstract (or virtual) processors. Abstract processors represent memory regions ([8], p. 21). The implementor can use the same number or a smaller number of physical processors to implement it. The language also provides an intrinsic function that returns the number of physical processors. It is useful to compute the number of virtual processors according to the number of physical processors. Figure 2 illustrates the distribution of an array according to the two basic distributions.



ARRAY A(4,6)



PROCESSOR P(2,3)

Figure 2: *Example of distribution. The array* `A(4,6)` *is mapped on the array of virtual processor* `P(2,3)` *by the directive of distribution* `DISTRIBUTE A(BLOCK,CYCLIC) ONTO P`. *The first dimension of* `A` *is distributed blockwise whereas the second is cyclicly distributed. The different colors of the array elements represent the abstract processors on which they are mapped. The alignment phase to a template is often by-passed when there is only one array to distribute.*

3 Irregularity in HPF

3.1 Regularity and irregularity

HPF is well-known for its efficiency in the case of very regular computations. Some extensions of the language propose irregular data distributions. Their goal is to allow HPF to be efficient with irregular computations. The problem that arises is that the load balancing problems become a burden for the programmer. Indeed, irregular data distributions are used to balance dynamically the load. Our goal is to improve HPF by integrating to it dynamic load balancing features. A dynamic load balancing system brings with it efficiency in the execution in an heterogeneous environment. We however do not want to restrict such an extension to a purely dynamic load balancing. We want to set up a framework that supports load balancing in order to study the relation between compile-time knowledge and dynamic load balancing. The goal is to guide the dynamic load balancing by the properties of the source code. At this time, properties are assumed to be detected and declared to the compiler by the user. Later, we may envisage to automate the detection of relevant properties. Let us review the benefits of regularity that we want to preserve and the irregular problems that we want to handle.

Regularity

We do not want to loose regularity because it is a concept with a lot of advantages:

Efficient accesses to the memory: Regularity allows memory to be linearly described. As a consequence, the cache hit and thus the bandwidth of the memory is very high because the usual data cache management policies are optimized for linear memory runs.

Simple computation of regular communications: The regularity of the data distribution enables simple representations of the data distributions. So, functions that map data onto virtual processors and their inverses are simple functions. A particular consequence is the easiness of the computation of communication schedules in regular codes.

Base of HPF compilation techniques: The notion of regularity in the distributions is the foundation of the HPF language. Almost all previous compilation techniques are based on this concept. By keeping it valid, they also remain valid. The techniques we are referring to are relative to linear algebra that is useful for regular computation compilation [1].

Irregularity

We need to handle irregularity because it is a key for high performances.

Real codes have a part of irregularity: Irregularity may come from an unbalanced distribution of computations or from a repeated irregular communication pattern. In these two cases, an irregular distribution may balance the load and/or reduce the overall communication cost in terms of number of messages and/or in terms of communication volume according to the architecture of the underlying network of communication.

Heterogeneous environments of execution are irregular: Dedicated massively parallel machines are still an important issue, but networks of workstations (NOW) are becoming more and more popular as parallel machines for intensive computations. In these environments, irregularity stems from the heterogeneity of the processors and/or from the multi-user environment. Irregular distributions can adapt to the distribution of the computation to the static and dynamic specificities of these environments: non-uniform processor power, variations of available processor power, etc.

The definition of the HPF language can be extent in two different ways. The first one consists in extending the diversity of data distributions allowed for the distribution of arrays onto abstract processors. This is achieved by extending the second level of HPF model. The second one is to control the mapping of abstract processors onto physical processors by extending the third level. Let us discuss these two possibilities.

3.2 Extending the second level: irregular data distributions

The second level describes the distribution of arrays onto abstract processors (Figure 1). The usual distribution is `cyclic(n)`. It is not well-suited for describing irregular distribution. Now, assume that we have a directive that specifies for each element of the array the virtual processor to which it must be assigned and that this assignment can change. Thus, we have the most general definition for a distribution directive that can support all distributions. The main drawback is a complete loss of regularity. It leads to a high overhead for codes that have a part of regularity. That is why more regular – but still irregular – distributions have been proposed.

This approach constitutes the basic idea of a lot of works because the limitations of the basic distributions have been quickly reached. Real codes like unstructured mesh codes or molecular dynamics codes need irregular distributions to do load balancing.

Vienna Fortran [7] has many way to handle irregular distributions. It provides a general block distribution which is a generalization of the usual block distribution by allowing the block size to vary. A more general distribution is the indirect distribution which uses an array to specify on which processor each element must be mapped. Finally, user defined distribution functions constitute the most general mechanism to specify distributions: they can express any arbitrary mapping between array indices and processors, including partial or total replication.

Annai [18] provides general block distributions and user defined data distributions. User defined data distributions consist of indirect distributions and mapping functions. Mapping functions are functions that

map global to local indices, local to global indices, global indices to processors and define the size of the local array which may be different on each processor. In contrary to the user defined distribution functions of Vienna Fortran, mapping functions are assumed to introduce more overhead than indirect distributions [7, 18].

Fortran D [26] has two different approaches of the problem. The first one [22] is, like other compilers, an index-based distribution. The user may specified indirect distributions through an indirection array. The second approach [27] is a value-based distribution. It consists in partitioning distributed arrays according to their values or to the values of an another array rather than according to their indexes. It can be seen as an indirect distribution where the indirection array is computed automatically by the run-time system.

High Performance Fortran Forum decides to include general block distributions and irregular distributions as approved extensions in the definition of HPF 2.0 [9]. The general block distribution and irregular distributions are restricted to arrays of dimension one. They can use a variable to define the mapping when apply with the `REDISTRBUTE` directive.

3.3 Extending the third level: irregular virtual processor mapping

The third level controls the mapping of abstract processors onto physical processors as shown in Figure 1). Irregularity can be obtain by using an irregular distributions of virtual processors. If we have more virtual processors than physical processors, we can balance the load by distributing virtual processors in such a way that the load of each physical processor is as close as possible to the average load. Figure 3 shows an irregular distribution of virtual processors onto 3 physical processors.

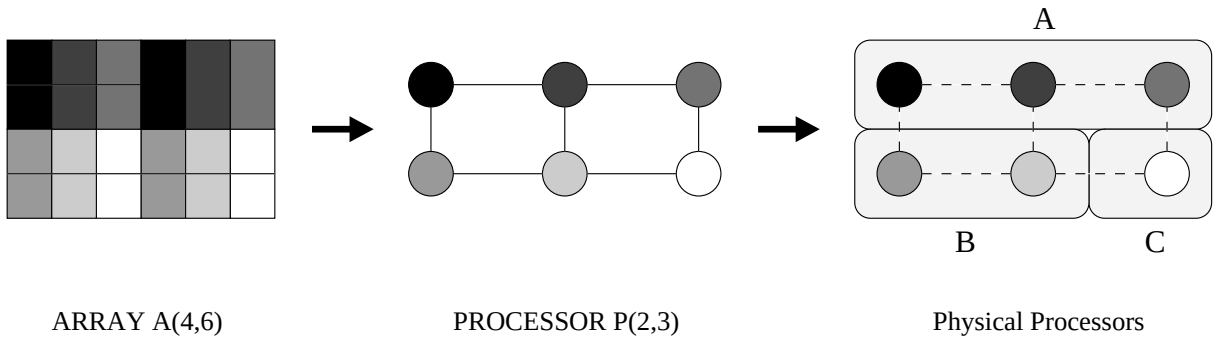


Figure 3: *Example of an irregular virtual processor distribution. Let processor A be 3 time as fast as processor C and processor B 2 time as fast as processor C. The array A(4,6) is distributed onto the virtual processor mesh P(2,3) using the directive* `DISTRIBUTE A(BLOC,CYCLIC) onto P`. *Then, the load balancing system can map these 6 virtual processors onto the 3 physical processors in an balance way because it has some knowledge of the execution environment.*

The regularity is embedded in the notion of virtual processor. The code of a virtual processor may be seen as the code currently generated at the process level by the HPF compilers (for the purpose of this discussion, we leave aside, for a while, the management of the virtual processors).

The irregularity is achieved by an irregular distribution of the virtual processors onto the physical processors. As this possibility was not taken into account (to our knowledge), we see two complementary approaches of using and defining irregular distributions of virtual processors. One is predicative whereas the other is reactive.

- First, the programmer may give informations to the compiler (and so to the run-time system) by some new directives when some knowledge about the code behavior is available in advance.
- Second, when good distributions cannot be predicted, dynamic techniques must be used in order either to balance the load of physical processors or to reduce communication costs (and to maintain load balancing).

This approach is a tradeoff between regularity and irregularity. We transform a fine-grained parallelism into a coarse-grain parallelism. The virtual processor becomes the central notion for irregularity and regularity: it

embeds a regular part of memory allowing static optimizations and it becomes the unit of irregularity. It makes sense because coarse grain level appears to be better that fine grain for the following reasons.

- The data level seems to be too low for current machines because their latencies and their bandwidths are better adapted for coarse grain applications according to their computing power. Furthermore, the load balancing decision is taken when the benefits exceed the communication costs of the balancing phase. This condition is generally not satisfied for a small volume of data.
- Another advantage of the coarse grain level over low level comes from communication costs. Reducing communication overheads, even at the price of an unbalanced distribution of the load, should be better, in terms of performance, than having a perfectly balanced distribution with a lot of overhead in communications. Overhead may result from communication computation costs and bad use of the memory cache as well as from the increase of the communication volume generally generated by very irregular data distributions. This last effect is especially important when a computation depends on its neighbors, that is to say when locality is a main feature of the application.

In conclusion, we propose the use of the third mapping level of the HPF model to handle the load balancing issues and not to lose the regularity. The two major advantages are the transfer of the main part of the load balancing responsibility to the system and the ability for the system to take into account external heterogeneity. The user has only to distribute the arrays in a way that maximizes regularity inside a virtual processor without generating too many communications across virtual processors.

4 Virtual processors as migrating threads

We now have to face to the problem of implementing virtual processors. We have 3 main choices. We can embed a virtual processor into a process, into a virtualization loop or into a thread. To decide, we list the main advantages and drawbacks of each of these solutions. The features we consider are the easiness of implementation, the overhead generated in time, the behavior of intra-processors communications (between virtual processors mapped onto the same physical processor), the number of virtual processors per physical processor (memory overhead) and the time for virtual processor migration.

4.1 One virtual processor per process

This usual solution allocates each virtual processor to an independent process. (See Figure 4). This is what current HPF compilers assume.

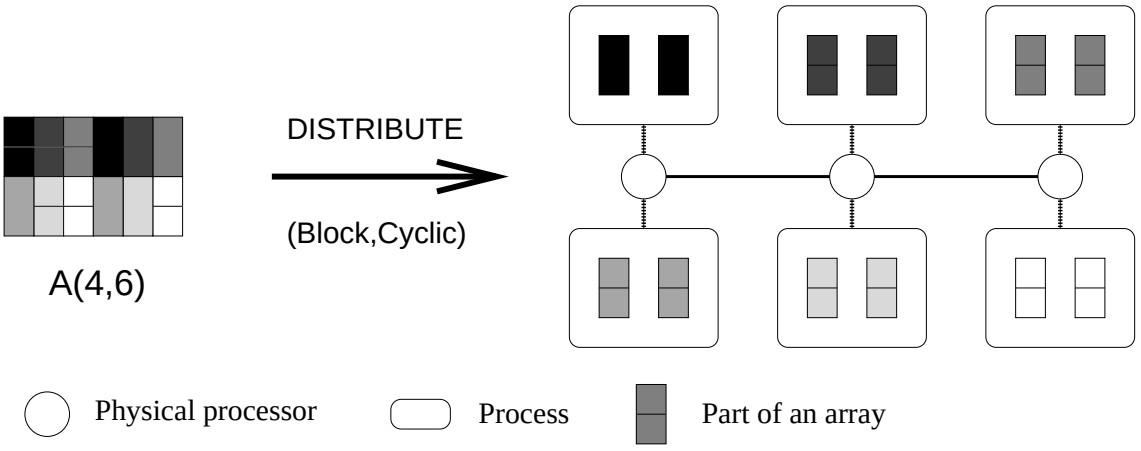


Figure 4: *Array A is distributed onto a 2×3 grid of 6 virtual processors. Each virtual processor is embedded into a process. The physical machine has only 3 processors, so each physical processor receives 2 processes.*

Advantage:

- ▷ This solution is very easy to implement. Current HPF compilers can generate more processes than physical processors. So, we can have several processes per physical processor, leading us to allocate several virtual processors per physical processor. The migration of process can be achieved by one of the numerous systems that support process migration like MPVM [6] or Cocheck [24].

Drawbacks:

- ◁ The context switch of processes is expensive.
- ◁ The communications between processes on the same processors are very slow compared to memory accesses. Thus, the communications between virtual processors onto the same physical processor – but in different processes – are slow.
- ◁ The replication of the process code limits the number of processes per processor: it is a huge waste of memory.
- ◁ The migration of processes is expensive due to the process size that represents millions of bytes. Even on a distributed file system like NFS, it still represents a lot of bytes.

4.2 Virtual processors emulated by virtualization loops

This approach maps several virtual processors into a process (and then onto a processor, whether we consider a process per processor). The process code essentially emulates the virtual processors by virtual loops. Basically, for a sequential machine, it consists of embedding each data parallel instruction in a loop that enumerate all virtual processors. This technique is often used in compiling explicit data parallel languages like C* [25]. Figure 5 illustrates this approach.

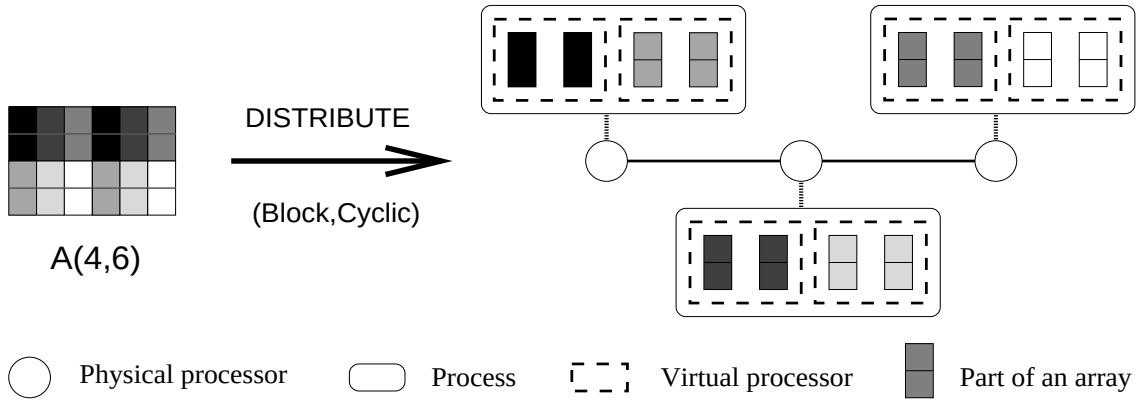


Figure 5: *Array A is distributed onto a* 2×3 *grid of 6 virtual processors. Virtual processors are emulated by virtualization loops. The physical machine has only 3 processors, so that each physical processor supports 2 virtual processors emulated by an unique process.*

Advantages:

- ▷ Direct memory communications between different virtual processors are available.
- ▷ As a small memory overhead is induced by the virtualization loops, a process can manage many virtual processors.
- ▷ The virtual processor migration is almost minimal because only the data and some management variables need to be sent.

Drawbacks:

- ◁ Contrary to explicit data parallel languages, dependences exists in HPF. Thus, this solution does not seem to be easy to implement because of the need of a scheduling of the virtual processors. The scheduling is needed to satisfy the dependences in virtualization loops.
- ◁ The virtualization loops create a kind of context switch overhead which is very low. But, the overall overhead is very high because of the intensive use of it.

4.3 Virtual processors as threads

A parallelism of processes can easily be transformed into a parallelism of threads when we do not consider interference of the operating system because a thread is a process without the interface to the outside world. This solution consists in embedding a virtual processor into a thread as shown in Figure 6.

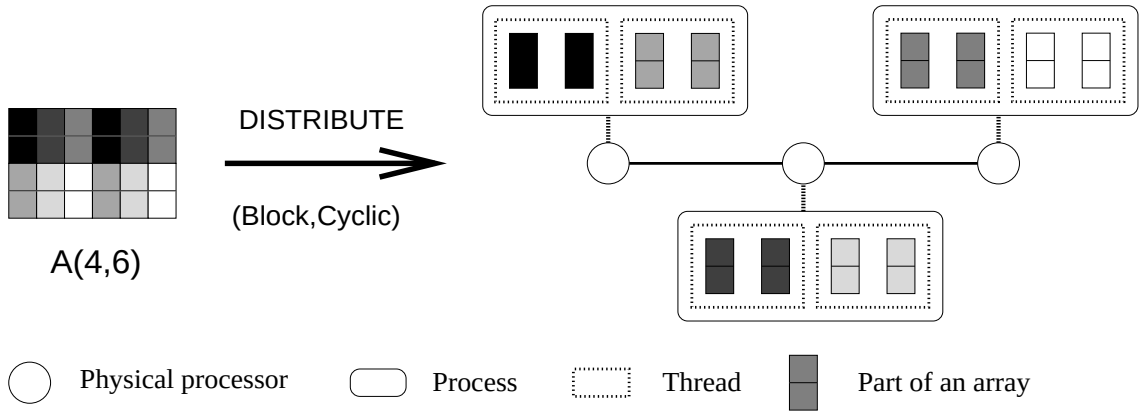


Figure 6: *Array A is distributed onto a* 2×3 *grid of 6 virtual processors. Each virtual processor is embedded into a thread. The physical machine has only 3 processors, so each physical processor receives 2 threads that are embedded in a process.*

Advantages:

- ▷ This solution may be seen as an extension of the previous HPF compilation approaches. Indeed, the code of a thread looks like the current process code. That is why this solution is thus relatively simple to implement.
- ▷ Direct memory communications between threads in the same process may be implemented, allowing efficient virtual to virtual processor communications.
- ▷ A thread only needs a private stack. Hence, the memory overhead is not too big, and, many threads can reside on a processor.
- ▷ The thread migration cost is low because a migration message contains the data, the used stack and some thread management variables. The data are generally the main part of the message.

Drawback:

- ◁ The main limitations stem from the multi-threaded environment. The limitations concern – but are not restricted to – portability, efficiency in thread creation and thread synchronization, thread context switch, integration of communication, etc.

4.4 Summary

We sum up all the previous observations in Table 1. Embedding a virtual processor into a process does not seem to be a good choice for high performance computing mainly because a process needs a lot a memory. It limits the number of processes per processor and represents a lot of data to be transfered. The virtualization looks like a good solution if it does not need a lot of work and if virtualization loops do not generate so much overhead. We choose to embed a virtual processor into a thread because of the good behavior of current multi-thread environments and the ease of implementation of this solution.

	Processes	Virtualization	Threads
Easiness of implementation	Easy	Lot of work	Medium
Context switch cost	high	very high (overall overhead)	low
Number of virtual processors per physical processor	few	a lot	many
Migrating data	code + data + stack	data + virtualization management variables	data + stack + thread management variables
Migration cost	expensive	cheap	cheap

Table 1: *Features of some implementation solutions for virtual processors.*

4.5 HPF and threads: related work

Some compilers generate several threads per process but their purpose is to overlap communications with computations. The multi-threaded approach offers a good solution according to the numerous works that are using it. However, we could not find any that considers threads as a load balancing facility in a HPF context. Threads are just used as an optimization technique.

The Paradigm compiler [3] of the University of Illinois integrates threads for the purpose of overlapping communications by computations. Their first approach was to generate more virtual processors than physical processors and to use a multi-threaded runtime package. They observe that this technique minimizes the compiler support required to obtain multi-thread code, but at the expense of a complex, machine dependent runtime support [14]. Then, they consider a message driven model that simplifies the design of runtime system, moving the complexity to the compiler and also increasing portability. Multi-threading generates super linear speedup because of cache effects. The main problem, however, is to determine the optimal number of threads per processor.

In [23] and [2], multi-threading is also used to overlap communications by computations in HPF-like languages. Whereas in [23], Sohn and al. use one thread per virtual processor, in [2], André uses only four different threads. A thread is responsible for sending the data to others processes. The *Receive* thread receives the message and stores it. The *Receipt manager* thread deals with received data. When the data needed for a computation are received, the *Computation* thread, informed by the *Receipt manager thread*, starts this computation. So, the number of thread is constant whatever the number of data.

In [21], the use of threads to support the execution of data parallel programs is studied. The overhead induced by the multi-threaded environment like thread creation, thread synchronization, communication and thread migration are experimentally studied. This paper shows that the choice between thread creation and thread synchronization is not obvious.

We also find systems like Athapascan[15], Nexus [11] or Chant [13]. Athapascan is the run-time support of the APACHE project and is divided in 2 parts. Athapascan-0 is a portable standard library that allows a parallel application to be described in terms of parallel procedure decomposition. Athapascan-1 is an interface for applications that provides load balancing facilities. Nexus and Chant are portable runtime interface for task parallel programming languages. They provide support for multi-threading and offer a global memory model. A particular aspect of these two systems is their integration of communications in multi-threaded system [12, 13]. Nexus is the compiler target of two task parallel languages which are Fortran M [10] and Compositional C++ [20]. Thread migration is not supported by any of these systems at the best of our knowledge.

5 Description of a preliminary implementation

In this section, we describe the most relevant aspects of our experimental implementation. Our hand-coded program is directly derived from the code generated by the source-to-source compiler Adaptor [4]. It is a FORTRAN 77 code with calls to a run-time library. The run-time library is written in C and implements high-level functionalities in cooperation with the multi-thread environment PM2 [19] like scalar and array broadcasts, global synchronization, data migration, etc. Our goal was not to create a complete environment but only to have some running examples in order to test our approach. The feasibility was one of our main criteria.

We use the multi-thread environment PM2 because it has thread migration. But, it is not an environment specially designed to be the run time system of a data parallel language compiler output. The use of a system like Nexus or Athapascan [15] – if only they supported thread migration – would have simplified either the run time appropriateness to a data parallel program or the integration of advanced load balancing features.

5.1 Global and local variable issues

There are two levels of variables: global variables defined in the main and local variables used in subroutines. Global variables are unique in a process and have the important property of being allocated at the same address in all processes. This result stems from the fact that global variables are statically allocated into the heap. This property is very important for thread migration because it allows a migrated thread to have direct access to global variables. Local variables are local to threads and do not have the property of being at a fixed address because there are allocated into the stack. Hopefully, the multi-thread system PM2 deals with the stack migration. It can compute the shift to apply to each pointer addressing a local variable. The pointer needs only to be declared to PM2. However, the FORTRAN 77 compiler we use generates a lot of pointers if we try to optimize the code. As it does not know about threads, it does not declare the pointers to PM2. That is why our experiments are done with a low level of optimization (flag `-O3` on Alpha).

5.2 Data and computation distributions

Assume that we have the HPF code listed on Figure 7. We must generate $n1 * n2$ threads (one per virtual processor) distributed in at most $n1 * n2$ processes. Each thread has to handle a block of the array `A`. This block is defined by 4 variables (2 per dimension): `A_LOW1` and `A_HIGH1` for the first dimension and `A_LOW2` and `A_HIGH2` for the second. These variables are initialized at run-time according to the value of `n`, `n1`, `n2` and the logical number of each thread. We give in Figure 8 the FORTRAN 77 code of the (hand-) compiled program. We can see that it is similar to the code currently generated by HPF compilers.

```
       real A(N,N)
!hpf$ processors P(n1,n2)
!hpf$ distribute A(block,block) onto P

!hpf$ independent
       do j=1,N
!hpf$ independent
          do i=1,N
             A(i,j)=i+j
          endif
       endif
```

Figure 7: *A HPF code example: a nest of loops of depth 2 without dependence.*

As previously mentioned , the main advantage of this approach is that we encompass in a thread HPF compilation knowledges like the one we have present here. So, a multi-thread extension of an HPF compiler does not seem to be difficult. The first major change is that the address space is modified. The second major change is the the need of introducing synchronization barriers.

```
A_LOW1  = (( N *((mod(rg_t,n1))    )) / n1 ) + 1
A_HIGH1 =  ( N *((mod(rg_t,n1))  +1)) / n1
A_LOW2  = (( N *((rg_t  /  n1)     )) / n2 ) + 1
A_HIGH2 =  ( N *((rg_t  /  n1)   +1)) / n2

call allocate(rg_t,A,A_LOW1,A_HIGH1,A_LOW2,A_HIGH2)

do j=A_LOW2,A_HIGH2
   do i=A_LOW1,A_HIGH1
      A(i,j)=i+j
   endif
endif
```

Figure 8: *The FORTRAN 77 (hand-) compiled program of the previous HPF program.* `rg_t` *is the logical number of thread and is in [0,n1*n2-1].*

5.3 Thread localization

When a thread needs to communicate with another one, it has first to locate its partner. The localization of a thread is not static because threads can migrate. In each process, there is an array (hidden in the run-time library) which contains for each logical thread number the logical process number where the thread lives.

This solution has the benefit of being simple and efficient but at the price of a global knowledge of thread migration. If it is not the case, pointers, for example, must be set by migrating thread so that a message can follow the pointer chain until it reaches the thread. Further optimizations may collapse chains of pointers.

5.4 Data localization

Once a thread is located, the base address of one of its array has to be found. Once again, it is a dynamic value as arrays are re-allocated after migration. Moreover, re-allocation is generally done at a different address.

To find the base address of a distributed array, we have used the same technique as for locating a thread: there is one array per process and per distributed array which contains the base address of the block of each distributed array for each thread (the address is valid only if the thread is present). These arrays are updated at array allocation at the array creation and after a migration. An illustration of this technique can be found in Figure 9.

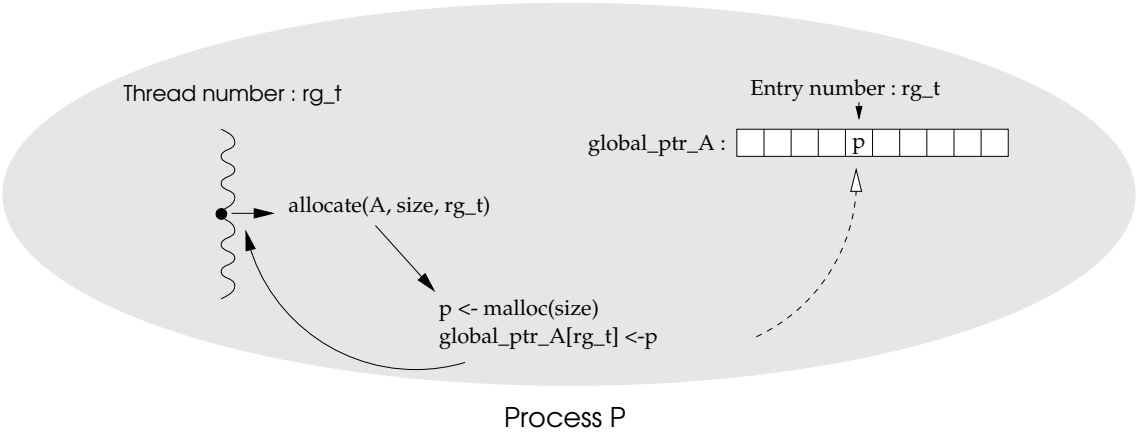


Figure 9: *Illustration of the dynamic array management. The allocation function of the virtual processor's part of the distributed array* `A` *writes the base address of the allocation into the adequate global array. The cell written is at the index corresponding to the logic number of the thread doing the allocation.*

The benefits are still the easiness and the efficiency but this time the drawback does not stem from a global knowledge but from a memory overhead. Indeed, only some entries of each array are used per process in average. These entries are the entries related to the arrays of the present threads. As the percentage of present threads decreases as the number of processes increases, most entries will not be used when many processes are used, leading to a waste of memory.

A solution may be to store the base array addresses of present threads only. So, the used memory becomes related to the number of present threads instead of the number of threads in the system. However, accesses are slowed down by a factor that depends of the storage structure used. Accesses are in average in O(log (number of present threads)) (resp. in O(number of present thread)) if a tree-structure (resp. a list) is used instead of O(1) for the full array storage.

5.5 Scalar codes

Scalar codes in parallel routines have generally to be executed once per process. There are 3 main possibilities to implement them:

1. a scalar thread executes scalar codes,
2. destruction of parallel threads at the beginning and re-creation at the end of each scalar code,
3. one thread among the "parallel" threads executes scalar codes. This is achieved by using mutual exclusion to a boolean test which specifies whether the code has been executed.

We have chosen the last solution because thread synchronization that is needed for its implementation is cheaper than thread creation in PM2 (see [21]). Moreover, the extraction of scalar codes required for the first approach is a complex problem in general whose benefits do not seem better that the third solution.

The most general scheme for our implementation of scalar code is shown in Figure 10.

- General scheme that embeds scalar code:

```
(1) call barrier(sync_local)
(2) call give_permission_to_the_first_thread(bool)
(3) if (bool) then
(4)      ! scalar code
(5) endif
(6) call barrier(sync_local)
(7) call end_of_permission(bool)
```

- Explanation:

  `(1)`: Intra-process synchronization of threads before the scalar part to ensure that all previous parallel code has terminated.
  `(2)`: Only one thread among the "parallel" threads sees its variable `bool` set to `TRUE`. All others see `bool` set to `FALSE`.
  `(6)`: Intra-process synchronization to wait for the completion of the scalar code and to be sure that all threads have gone beyond the test guarding the scalar code.
  `(7)`: End of the critical section.

Figure 10: *General FORTRAN 77 scheme that embeds scalar code and its explanation.*

This implementation is not optimal because `give_permission_to_the_first_thread()` and `end_of_permission()` have mutual exclusions that may lead to generate context switches. A more efficient solution would be to merge these calls with those of the barriers. Further optimizations can focus on removing these calls. If two scalar parts are not separated by a call to the load balancing module, the inner calls to `give_permission_to_the_first_thread()` and `end_of_permission()` can be removed.

Another delicate point of this implementation is that there must always be at least one thread per processor to execute the scalar code. It seems that it is possible to extend this implementation to take into account situation where a processor has no thread but this is not currently done.

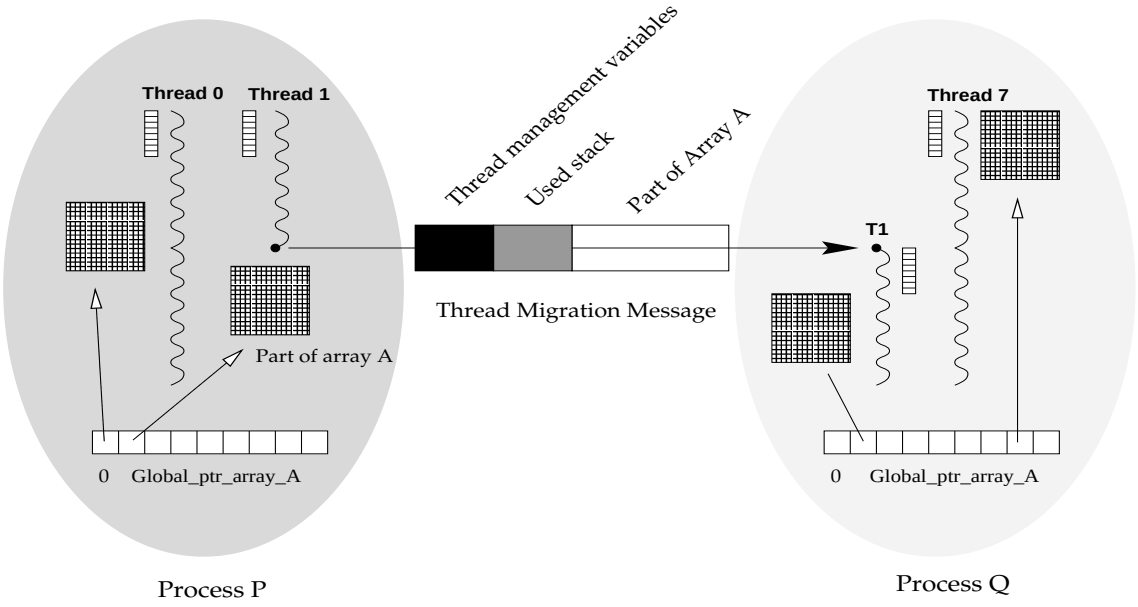


Figure 11: *The migration of a thread consists in sending the thread management variables, the stack and the arrays allocated by the thread. These arrays are not migrated by the multi-thread system PM2 but we can add them to the migration message. At the destination process, the migrated thread has to re-allocate its arrays and to set global variables correctly.*

5.6 Thread migration

The thread migration decision is currently known by each process. This knowledge allows processes to maintain the array of thread location easily(see Section 5.3).

The PM2 multi-thread system deals with stack migration. But it is not responsible for moving allocated data. Hopefully, it provides functions to add and remove data to the migration message. So, we have used this possibility to transfer allocated arrays with the migration message as shown in Figure 11. The re-initialization of the thread in the destination process consists in re-allocating arrays and in correctly setting global variables related to the migrated thread like for example the number of local threads for the synchronization barriers.

6 Experimental results

In order to estimate the performance of our approach, we use a well-known program which is Gaussian elimination with partial pivoting. The classical solution of this problem is to use a cyclic distribution for the columns and to swap the column holding the pivot with the current column `k` at column iteration `k`. In our program, we use a block distribution of columns. Load balancing is achieved by migrating threads.

We chose this test program because the block-distribution approach needs to be balanced and because we know the cyclic distribution which transforms this problem into an regular one. So, wee can see how far we are of the time of the regular problem which achieves the lower bound. As our code is derived from the code produced by Adaptor, we use Adaptor as the reference.

All the experiments were done on the Alpha farm of the LIFL [17]. It is composed of 16 DEC 3000 model 400 AXP whose processors are DECchip 21064 running at 133 MHz with 512 Ko of memory cache and 64 Mo of main memory. The interconnection network is a FDDI crossbar of optic fiber. Each link has a bandwidth of 200 Mb/s. We use PVM with the `PvmDontRoute` flag with gives a startup cost of 2.9 ms and a bandwidth of 13.3 Mb/s. This constraint is imposed by Version 1.6 of PM2. It is removed in Version 2.0 which will be available by the end of 1996.

Our test matrices are very regular because the pivot is always in the right column. That is to say that on the k^{th} iteration, the pivot is in column `k`. Thus, a cyclic distribution always achieves a perfect load balance whereas a block distribution needs load balancing.

First, we examine the behavior of our program for various matrix sizes for one processor without multi-threading. Next, we introduce multi-threading to measure the speedup achieved by cache effects. Finally, we report on the tests concerning the load balancing version.

6.1 One processor, one thread

In order to compare between our implementation and the code generated by Adaptor, we have tried to be close to the code produced by Adaptor. The differences are essentially the calls to the thread library like barriers calls and the use of LRPC (*Light Remote Procedure Call*) instead of send/receive commands. LRPC can be seen as active messages and are useful for implementing remote get and put operations. Our small run time library is less general that Adaptor's one and thus suffers less overhead.

	size of the matrix			
	256	512	1024	2048
Adaptor `block`	0.56	6.64	62.0	520
Adaptor `cyclic`	0.56	6.65	62.0	516
PM2	0.51	6.65	60.1	484
Difference	-9%	=	-3%	-6%

Table 2: *Times in seconds for several programs and several matrix sizes.* `block` *and* `cyclic` *are the distribution directive used in the HPF version compiled by Adaptor. The PM2 version uses a block distribution of the matrix. The difference is computed between Adaptor* `cyclic` *and PM2.*

As shown in table 2, the single thread uniprocessor time of both programs are similar. The differences do not exceed 10% and the average is less than 5%. Adaptor's version is slower because it has to handle array alignment and distribution for communications whilst we do not.

6.2 One processor, many threads

We also want to measure the impact of multi-threading. So, we plot the execution times for various thread numbers running on one processor. The results are reported in Figure 12.

First, a super linear speedup appears for some thread numbers and some matrix sizes. As discussed in [21], it results from cache effects. For matrix sizes of 512 and 1024, we observe a maximal gain of 7.3%.

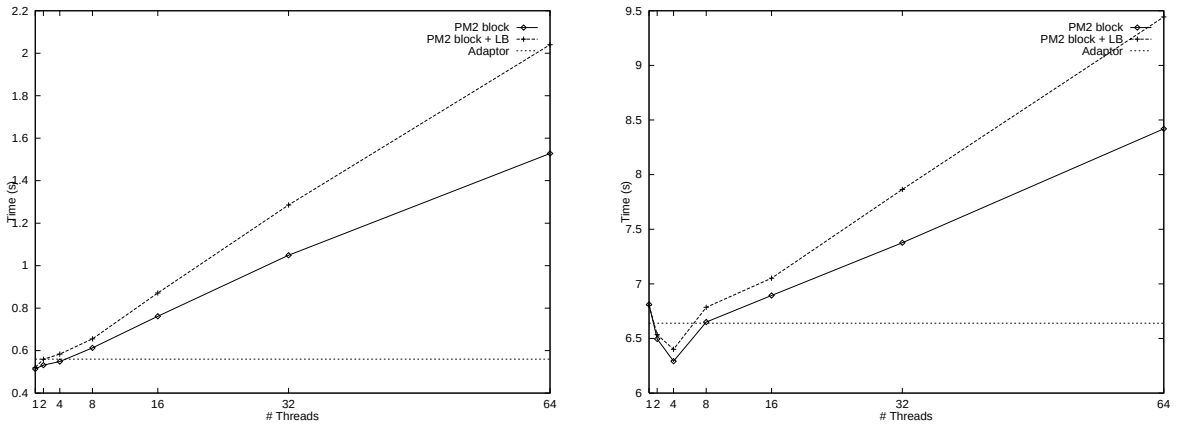
Second, the thread overhead is mainly due to the thread synchronization barriers. We know (see [21]) that, in PM2, the completion time of one barrier is linear with respect to the number of threads calling it. This is why the curve becomes linear once all the memory accesses are in the cache. The slope of the curve for the linear part is around 50 μs per thread. As there are 4 barriers per loop, we obtain an average completion time of 16 μs per thread and per barrier.

6.3 Overhead of the load balancing module

In Figure 12, we have also plotted the sequential time for the same program but with the load balancing module. As there is only one processor, no migration occurs. We can see that the behavior is the same and that the slope of the linear part is higher. This time, it is around 100 μs per thread. The explanation is that we have introduced two more barriers. However, the completion time per thread and per barrier is still around 16 μs.

The overhead introduced by the load balancing module is quite small since it is around 2% as shown in Table 3. The module tries to compute a new distribution by moving a thread to the processor which sends the last pivot from other processors. The choice of the thread to be sent is not arbitrary. One chooses the thread with the highest line number among the threads allocated to the most loaded process.

The goal of such an heuristic is to minimize the number of migrations. We have tried to keep it simple enough to be done automatically. The idea behind this heuristic is based on the knowledge that loop iteration increases from 1 to N. At each iteration `k`, only the array `A[k:n,k:n]` is written while the whole array is `A[1:n,1:n]`. Thus, the last line to be processed in a process is the line that has the maximal rank number. By migrating it, we hope postpone the date of the next migration.



(a) matrix size: 256 × 256

(b) matrix size: 512 × 512

(c) matrix size: 1024 × 1024

(d) matrix size: 2048 × 2048

Figure 12: *Time in second of Gaussian elimination with partial pivoting for different matrix sizes and different programs.* `LB` *stands for the load balancing module. The curves are plotted in function of the number of thread.*

	size of the matrix			
	256	512	1024	2048
PM2 block	0.51	6.65	60.1	479
PM2 block + LB	0.52	6.80	61.2	481
Difference	+2%	+2.3%	+1.2%	+0.5%

Table 3: *Time in seconds for the thread implementation without and with the load balancing module for one thread on one processor.*

6.4 Performance of our implementation

As we can see in Figure 13, the time obtained with the load balanced version is about the time of the cyclic version for the first experiment and 5%-7% worst than the Adaptor program for the second one. For some thread numbers, it achieves the same performance.

We are able to achieve the performance of the cyclic distribution because communications are slow compared to the computation power. Thus, an unbalance distribution (but not too much) is masked by the communication.

The load balancing modules brings an improvement of 17% in time between the best time for the program without and with the load balancing module. In general, the time of thread migration represents less than 2 seconds for the whole execution of the program.

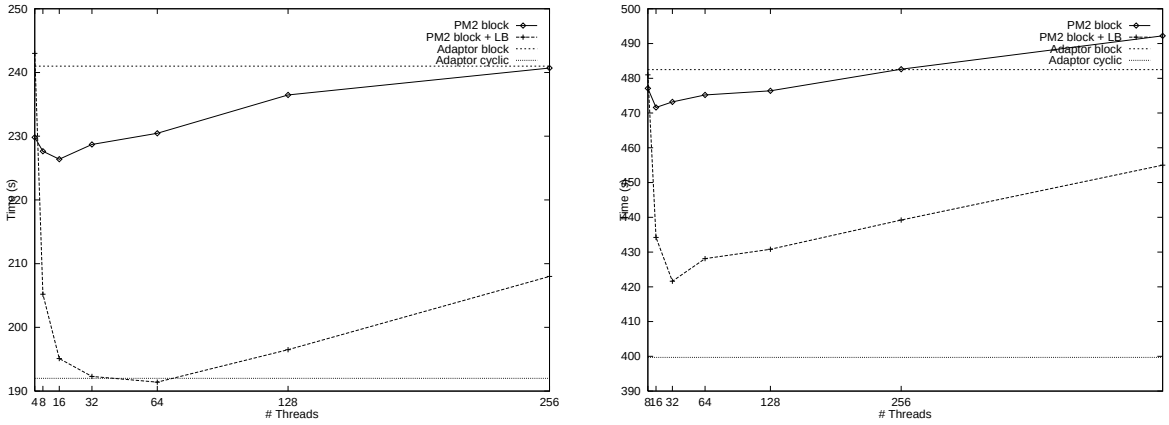


Figure 13: *Time in seconds for a 2048×2048 matrix on 4 processors (left) and for a 3072×3072 matrix on 8 processors (right). The time for Adaptor* `cyclic` *and* `block` *are plotted as well as the time for the thread program without and with the load balancing module.*

7 Conclusion

We have presented in this paper a method that exploits the HPF definition in order to introduce a load balancing mechanism in to the run time. The basic idea is to use the virtual processors of HPF as the unit of migration. So, independently of the alignment and the distribution, we can balance the load of the application by controlling the distribution of virtual processors onto physical processors. We have compared the advantages and the drawbacks of 3 possible implementations. We have validated the feasibility of this approach with a preliminary implementation which implements a HPF virtual processor in a thread and thus migrates threads. Experimentations were done with a well-know program that is the Gaussian elimination with partial pivoting. Measures seems to confirm that this is an interesting direction.

Further work is needed to test this approach on a large range of applications. The integration of HPF and migrating virtual processors has to be developed as well as the load balancing module. However, some questions appears. For example, what is the good number of threads for a given core of computation ? Should the number of thread remains constant or should it vary ?

Acknowledgments

I would like to thank Luc Bougé for his guidance and suggestions. I would like also to thank Thomas Brandes for his explanations and helpful comments. I thank also J.-M. Geib, J.-F. Mehaut and R. Namyst from LIFL for their hospitality and their help for PM2. Last thanks to the LIFL which granted me access to their Alpha farm.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. Technical Report A-278-CRI, CRI, ENSM Paris, November 1995.

[2] F. André. A Multi-Threads Runtime For The Pandore Data-Parallel Compiler. Technical Report 986, IRISA, February 1996. Available at URL http://www.irisa.fr/EXTERNE/ bibli/pi/pi96.html.

[3] 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.

[4] T. Brandes. Adaptor (HPF compilation system), developed at GMD-SCAI. Available at URL http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html.

[5] T. Brandes and F. Desprez. Implementing pipelined computation and communication in an HPF compiler. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1123 in Lecture Notes In Computer Science, pages 459–462, Lyon, France, August 1996. Springer.

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

[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] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1994. Version 1.1.

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

[10] I. Foster and K. M. Chandy. Fortran M : A Language for Modular Programming. *Journal of Parallel and Distributed Computer*, 1994 (to appear). Available at URL http://www.mcs.anl.gov/fortran-m/papers/.

[11] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Task-parallel Runtime System. In *1st Int. Workshop on Parallel Processing*, 1994.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, to appear. Available at URL http://www.mcs.anl.gov/nexus/.

[13] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant : A Talking Threads Package. Technical Report 94-25, ICASE, NASA Lamgley Research Center, April 1994.

[14] J. Holm, A. Lain, and P. Banerjee. Compilation of Scientific Programs into Multithreaded and Message Driven Computation. In *Proceeding of the 1994 Scalable High Performance Computing Conference*, pages 518–525, Knoxville, TN, May 1994.

[15] IMAG Laboratoire de Modélisation et Calcul (LMC). Projet APACHE. Available at URL http://www-apache.imag.fr/apache/index.html.

[16] A. Lain. *Compiler and Run-Time Support for Irregular Computations.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[17] Philippe Marquet. Ferme d'Alpha du LIFL. Available at URL http://www.lifl.fr/~marquet/ ferme-admin/install/install.html.

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

[19] R. Namyst and J.-F. Mehaut. PM2 parallel multithreaded machine : A multithreaded environment on top of PVM. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95*, pages 179–184, Lyon, France, September 1995. LIP, ENS Lyon, Hermès.

[20] California Institute of Technology. The CC++ Programming Language. Available at URL http://compbio.caltech.edu/CCplusplus.html.

[21] C. Perez. Utilisation des processus légers pour l'exécution de programmes à parallélisme de données : étude expérimentale. Technical Report 96-09, LIP, ENS de Lyon, April 1996.

[22] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers. Technical Report UMICAS-TR-94-57, University of Maryland, 1994. Available at URL http://hyena.cs.umd.edu/pub/papers/irreg-support.ps.Z.

[23] A. Sohn, M. Sato, N. Yoo, and J.-L. Gaudiot. Effects of multithreading on data and workload distribution for distributed memory multiprocessors. In *Proceeding of the 10th IEEE International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[24] G. Stellner and J. Pruyne. Resource management and checkpointing for PVM. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95*, pages 131–136, Lyon, France, September 1995. LIP, ENS Lyon, Hermès.

[25] Thinking Machine Corporation, Cambridge MA. *C* programming guide*, 1990.

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

[27] R. von Hanxleden, K. Kennedy, and J. Saltz. Value-Based Distributions and Alignments in Fortran D. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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)
ISSN 0249-6399