HAL
open science

On the Compilation of Data-Parallel Languages on a Distributed Memory Multithreaded Environment with Thread Migration

Christian Pérez, Raymond Namyst

To cite this version:

Christian Pérez, Raymond Namyst. On the Compilation of Data-Parallel Languages on a Distributed Memory Multithreaded Environment with Thread Migration. [Research Report] RR-3207, LIP RR-1997-20, INRIA, LIP. 1997. inria-00073482

HAL Id: inria-00073482

https://inria.hal.science/inria-00073482v1

Submitted on 24 May 2006

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the compilation of data-parallel languages on a distributed memory multithreaded environment with thread migration

Christian Perez

Raymond Namyst

N° 3207

Juillet 1997

THÈME 1

Rapport de recherche

ISSN 0249-6399

On the compilation of data-parallel languages on a distributed memory multithreaded environment with thread migration

Christian Perez
Raymond Namyst

Thème 1 — Réseaux et systèmes
Projet ReMap

Rapport de recherche n° 3207 — Juillet 1997 — 15 pages

Abstract: This paper focuses on the use of distributed memory multithreaded environments in data parallel programs and has two main goals. The first is to show that data parallel programs can support features like communication overlapping, load balancing without global data parallel object redistribution and the efficient use of clusters of uniprocessor and/or symmetric multiprocessors (SMPs). Our extended model introduces *virtual processes*. Virtual processes are implemented with mobile threads. The second goal is to determine the interactions between data parallel programs and a model of distributed memory multithreaded environments, with respect to intra-node communications and especially to thread migration. This paper also discuss this multithreaded environment with respect to the different models of threads and shows that the HPF and the C* data parallel compilation models easily integrate the proposed model.

Key-words: Data-parallel languages, Compilation, Distributed memory multithreaded environment, Thread migration.

(Résumé : tsvp)

This work has been supported by the INRIA project *ReMaP* and by he French CNRS Coordinated Research Project on Parallelism *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

De la compilation de langages data-parallèles sur un environnement multithread à mémoire distribuée incluant la migration de thread

Résumé : Ce papier s'intéresse à l'utilisation des environnements multithread (processus légers) à mémoire distribuée dans les programmes data parallèles. Il vise deux objectifs principaux. Le premier est de montrer que les programmes data parallèles peuvent intégrer des caractéristiques comme le recouvrement des communications par les calculs, l'équilibrage de charge sans redistribution globale des objets data parallèles et l'exploitation efficace des groupes de machines uniprocesseurs et/ou multiprocesseurs symétriques (SMP). Notre modèle étendu introduit les *processus virtuels*. Les processus virtuels sont implémentés à l'aide de threads mobiles. Le second but du papier est de déterminer les interactions entre les programmes data parallèles et un modèle d'environnement multithread à mémoire distribuée vis à vis des communications intra nœud et particulièrement vis à vis de la migration de threads. Ce papier situe aussi cet environnement de thread par rapport aux autres modèles de thread et montre que les modèles de compilation data parallèles de HPF et C* s'intègrent facilement dans le modèle proposée.

Mots-clé : Langages data-parallèles, Compilation, Environnement multithread à mémoire distribué, Migration de threads.

1 Introduction

Data parallelism [9] provides a structured model for parallel programming. One challenge is to efficiently execute data parallel programs on distributed memory machines. This goal seems to have been reached for regular programs with some data parallel languages like High Performance Fortran [14]. Now, the current challenge is to efficiently support semi-irregular applications with such languages.

Recently, the integration of threads in data parallel runtime libraries has dramatically increased [15, 21]. This is due to both the availability of threads libraries in many operating systems, allowing efficient use of shared memory multiprocessor machines [29], and to the availability of *multithreaded environments* [16, 24] providing programming models that integrate thread capabilities with various remote communication schemes on distributed architectures. This integration has exhibited particularly good properties for overlapping communications by computations.

In this paper, we show that using threads in data parallel programs can bring more benefits, especially for load balancing purposes. To achieve efficient execution of semi-irregular data parallel applications, we propose an extended model of execution for data parallel programs based on *virtual processes*. The idea is to allow compilers to make use of a potentially high number of virtual processes which are dynamically mapped onto physical processors. Because the mapping of virtual processes is dynamic, load balancing of semi-irregular computations can be achieved by moving virtual processes across the nodes of the underlying architecture.

Our implementation of virtual processes is based on the use of the multithreaded environment PM^2, which provides mobile threads on distributed architectures. A preliminary study [28] has shown that embedding HPF abstract processors in mobile threads is feasible and that good performances can be achieved through thread migration for a particular data parallel program. This study has shown us that the distributed memory multithreaded environments can help data parallel runtimes manage virtual process migration and that PM^2 needs to be specialized for data parallel programs to offer good efficiency.

This paper is divided as follows. Section 2 presents the classical data parallel compilation view and shows how it lacks some features. In section 3, the virtual process based model is presented. In particular we show how the HPF and C* compilation models can be modified to support it. Section 4 begins by dealing with the different kinds of threads and the different models of distributed memory multithreaded environments. It finishes by reviewing the previous works that have used threads in the execution of data parallel programs. A particular model of distributed memory multithreaded environment, PM^2, and its functionalities that seem interesting for data parallelism are presented in section 5. This leads us to section 6 which deals with the integration of data parallelism and a distributed memory multithreaded environment. Section 7 presents future work and we conclude in section 8 with some open questions.

2 Extending data parallel compilation model

2.1 Classical distributed memory data parallel compilation view

Classical data parallel language compilers distribute the parallel work onto a virtual machine whose processors are called *virtual processors*. These *virtual processors* do not belong to the data parallel languages and are different, for example, from C* [33] virtual processors. They are an abstraction of the real machine for the compilers. The compilers assume *virtual processors* to be mapped one per node because a node of the distributed memory is assumed to only contain one processor. As *virtual processor* are mapped into a process and some parallel machines can only map one process per node, this model seems adequate. Figure 1 illustrates the current situation. So, compilers generate low level code because they assume one *virtual processor* per physical processor and they assume that all virtual to virtual processor communications are distant and need to go through the interconnection network.

This approach is limited for load balancing issues and for recent technologies. First, clusters of symmetric multiprocessors can not be efficiently exploited. Current compilers do not know how to efficiently use such clusters because their compilation scheme does not deal with several processors in a local shared memory space. Second, some complex modules are added in the compilation and/or into the runtime to overlap communications by computations. Third, and maybe the most important one, load balancing can only be done through redistributions of data parallel objects. Redistributions are expensive and force users to have a good knowledge of the machine and to spend time writing complex machine dependent code [28].

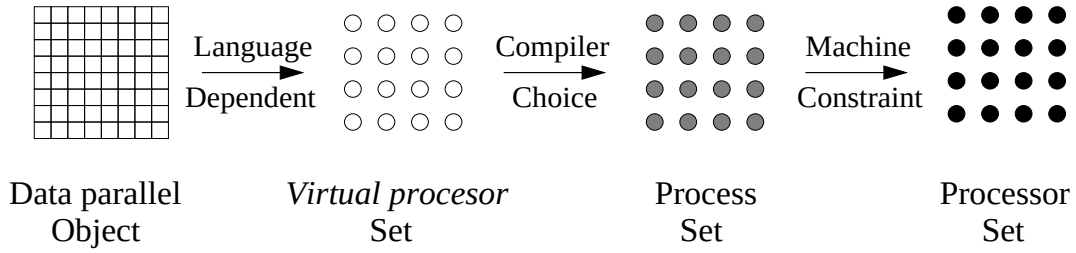

Figure 1: *Data parallel languages distribute data parallel objects onto a virtual machine. Classical data parallel compilers map one* virtual processor *into one process. Then, one process is assumed to be mapped to one processor. Processors are assumed to only be able to communicate through message exchange.*

2.2 Mapping many *virtual processors* into a node

The three problems explained above can be solved by mapping several *virtual processors* per node and by allowing *virtual processors* to move from one node to another.

Virtualization of the underlying architecture The number of processors of a node only affects the computational power and the computation of the machine's load. Indeed, the time to compute p independent tasks (all assumed to have u units of work) on a machine with n processors that can individually proceed α units of work per unit of time is $\alpha \times u \times \lceil \frac{p}{n} \rceil$ while the load of the machine is $\frac{p}{n}$. So, the underlying architecture is virtualized because several *virtual processors* can be mapped to a node which can have one or several processors. The criteria becomes only the overall efficiency without taking care of the overheads introduce.

Overlapping communications by computations Mapping several *virtual processors* to a node introduces a possible overlapping of communications by computations. If a *virtual processor* is blocked on a message receive operation, others *virtual processors* can go on. It is argued in [30] that overlapping communications by computations has a limited utility in data-parallel programs. The achieved speedup is bound by a factor 2 and is generally less. The kind of overlapping generated here, i.e. pipelining, is the worst kind because it may slow down the application due to the cost of sending/receiving messages. So, the benefit of this approach is that it automatically provides overlapping and supports aggregating techniques when needed. For example, the real communications can be done by only one (special) *virtual processor*.

Load balancing the application by migrating processes A dynamic mapping of *virtual processors* to processors can also be used in order to improve the balance of the application's load. When some nodes become overloaded or underloaded, load balancing techniques can be applied to balance the load by moving *virtual processors*. The advantages are a localization of the "redistribution" phases and a distinction between the data parallel object distribution, whose aim is to create parallelism, and the load balancing phase whose goal is to achieve efficiency. The localization means that only the processors that have to send or receive some *virtual processors* are going to have some extra work to do. And yet, this extra work does not prevent other *virtual processors* of such processors going on with their computation. Finally, complex load balancing policies can be embedded in the runtime. The user has only to choose the best one for his/her program/machine. So, a heavy burden can be removed from the users' shoulders.

3 Compiling data parallel programs onto virtual processes

The previous section has shown that mapping several *virtual processors* per processor virtualizes the machine, creates communication overlapping and especially enables load balancing strategies to be applied for data parallel programs. This section defines virtual processes which have a better suited definition than the *virtual processors* and shows how data parallel compilation can be extended to support them.

3.1 Extending data parallel compilation model with virtual processes

Current compilers generate code for *virtual processors* because they assume that the corresponding process will be alone on a processor. But, it seems interesting to have several such *virtual processors* per processor. In order not to confuse between the classic view and this view, we are going to define virtual processes.

We define a virtual process as a part of the distributed domain and the computation related to it, if we refer to the *owner computes rule* that most data parallel compilers use. It embeds the logical functionalities of what classical compilers mapped to a *virtual processor*, i.e. to a process. So, a virtual process can be seen as the code generated at the process level by classical data parallel compilers but with the assumption that many virtual processes can be mapped to a processor via a process. Figure 2 illustrates this model.

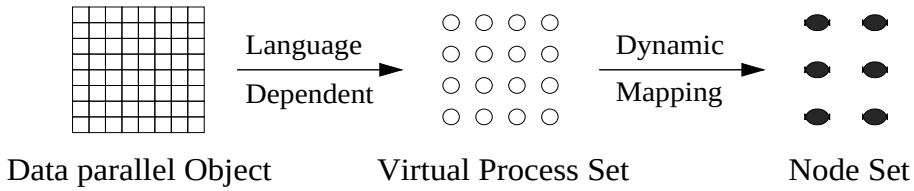

Figure 2: *Data parallel objects are distributed onto virtual processes. Then, several virtual processes are mapped onto a node. Nodes can contain one processor or several shared memory processors.*

The data parallel compilation model has to be extended because now compilers do not generate processor level code but virtual process level code. The difference is that communications between two virtual processes can be local to a node or can be between two different nodes. This approach modifies the compilation model that deals with communications. However, it does not invalidate the current compilers but only enables new optimizations in the communications between virtual processes on the same node. Now, let's examine what this approach means for the two main data parallel compilation models, the HPF and C* models.

The HPF model [14] defines a three level mapping of data. The first level is an alignment level. Arrays are aligned relative to one another or to a `template`, which is an abstract reference array. Then, arrays are distributed onto abstract processors. The last level is declared to be an optional implementation-dependent directive ([14], p. 20). It consists in mapping abstract processors onto physical processors. So, the abstract processors of the HPF model become our virtual processes. While classical data parallel compilers map one abstract processor per node, our approach maps several virtual processes per node. As also shown in [28], the virtual process approach fits very well in the last level of the HPF model.

The C* model, without machine dependent functionalities, is simpler because the programmer can not specify the distribution of his/her shapes. By definition [33], each C* virtual processor is an independent processor. In MIMD machines, the shapes are distributed in block by some compilers to increase grain. Then, each block emulates the C* virtual processors in virtualization loops. We decide that the virtualizations loops are themselves executed by some virtual processes that have to be mapped on a set of nodes.

So, the virtual process model extends very well the HPF and the C* compilation model because it only extends the implementation low level part of these models.

3.2 Implementing virtual processes

One solution is to embed a virtual process in a process and map several processes per processors. But, this approach suffers two main problems. Firstly, some hardwares and/or operating systems like the CRAY T3D or the IBM SP2 only permit mapping one process per node. Secondly, mapping several processes to a node is not an efficient solution for the following reasons. First, the SPMD code is replicated even thoug memory is a limiting resource for data parallel programs. Second, communications between local processes are expensive even if they are on the same processor. In some architectures, this drawback can be solved by using shared memory, but it is not portable. Thirdly, process context switches are expensive.

So, it is some times feasible to implement a virtual process into a process but it is not portable on all kind of machines and it is not very efficient. In fact, we would like to have something light that can run in a process. Lightweight processes appear to provide a good solution. So, the next step is to define lightweight processes and distributed memory multithreaded environment models.

4 Lightweight processes for parallel programming

In this section, we will first recall the main characteristics of the thread concept and why it is well suited to parallel programming. Then, we present some of the most representative environments that are integrating threads and communications on distributed architectures.

4.1 Threads

Lightweight processes are born from the idea of separating characteristics related to *execution flows* from those related to system resources (code, memory, opened files table, etc.) inside system processes. Thus, a thread can be considered to be not much more than an execution stack and an instruction pointer. Several threads can run inside the same system process, sharing its address space, code and other system resources.

These characteristics make operations on threads (context switch, creation, etc.) run at least an order of magnitude faster than those related to system processes. That's the main reason which makes them very interesting in parallel applications. Another reason is bound to their ability to efficiently overlap I/O latencies by computations since inside a system process, a blocked thread (issuing an I/O operation) can often be quickly replaced by a ready-to-execute thread by the scheduler. The quality of these two main properties — efficiency and I/O recovery — depends upon at which level the thread scheduler is implemented.

When the scheduling is implemented at the system level, then threads are called *kernel level* threads and are scheduled by the system scheduler itself. This allows it to correctly trap blocking calls and yield the execution to another thread in the process, or to make multithreaded applications benefit from the real underlying parallelism by transparently balancing the threads among the available processors. However, operations of kernel level threads yet incur some overhead due to crossing system boundaries for each call.

When the scheduling is implemented at the *user level*, the system scheduler is not aware of the existence of threads. As a result, user level threads of an application can not be scheduled over several processors by the system nor be blocked individually on a system call. Despite of these drawbacks, user level threads are interesting because their management is very efficient and because the functionalities of the runtime are easily customizable by an application programmer.

Threads exhibit good properties for both efficiency and ease-of-use. That's why they are used in many application fields such as file servers, compiler runtimes [23], graphical interfaces or even arcade games. In this respect, it is not surprising that the people involved in the design of parallel applications have tried to integrate threads in distributed computing environments. We develop this point in the following section.

4.2 Threads and distribution

Due to the "fine grained" computation resource a thread represents, both in terms of efficiency and scalability, its use in computation intensive parallel applications was recently engaged. In particular, the challenge was to integrate them with communication mechanisms in order to benefit from their ability to recover I/O operations (i.e. communications) on distributed architectures such as networks of workstations. There are two main ways of realizing such an integration.

The first one consists in doing this integration at the application level, by layering both on top of a communication library and a thread library. Although this approach may appear straightforward, several problems may be encountered:

Semantic problem Most of current communication libraries (PVM [17], MPI [22], etc) are not designed to be used in a multithreaded context (i.e. they are not *thread-aware*). For example, some primitives allowing a process to wait for a message are often provided, but the result of several threads calling such a primitive is not defined (i.e. which thread will get the message ?).

Technical problem In addition to the previous semantic problem, one has to consider the usual case where the communication library is not reentrant (not *thread-safe*). As a result, synchronization mechanisms (such as mutexes) are needed to ensure exclusive access to the communication library and "polling algorithms" have to replace direct calls to blocking primitives.

Engineering problem Finally, by solving the two previous problems at the application level, embedding a lot of "specific synchronization code" within regular code may prevent some parts of the application from being easily reused.

With respect to these important drawbacks, this first approach is now seldom used for the benefit of the second one: the integration of threads and communications within a generic *multithreaded* environment. The main idea is to provide a well defined programming model where the thread concept is "global" to the whole underlying architecture (and not just hidden into system processes) and where the technical problems are solved outside the application level. Recently, a strong effort has been made in this research field. As a result, many multithreaded environments are now available on a wide range of distributed architectures [16, 18, 3, 13, 6, 25]. Although these environments are very different, the main mechanisms they provide are often a combination of four basic programming models which are the *data-driven execution* model, the *remote thread operation* model, the *message passing* model and the *remote procedure call* model.

The *data-driven execution* model is featured by environments such as Cilk [3] or TPVM [13]. In Cilk, threads only execute non-blocking code and dependencies between threads are supported by a *continuation* mechanism that starts a new thread only when all its dependencies are satisfied. However, some applications may be difficult to write in this model. That's why the TPVM approach also permits message passing.

The *remote thread operation* model is featured by environments such as Chant [18] or RThreads [10]. This approach extends the semantic of a subset of common threads operations (creation, joining, cancelation, etc) to a distributed context. Although this approach may make the design of distributed applications very straightforward, it is limited to some "well behaved" mechanisms (threads creation or joining) in order to avoid severe efficiency problems tied to network latencies. For instance, synchronization mechanisms such as semaphores are rarely "transparently" extended to a distributed context, because the P and V operations on a remote semaphore would become very expensive compared to their "local" versions.

The *message passing* model is featured by environments such as TPVM, Chant or DTMS [7]. In this approach, threads are able to exchange messages (usually asynchronously) with each other. Therefore, unique *global names* are assigned to threads, so that they can be remotely referenced by each other. Message passing is extensively used in distributed systems, and one may expect multithreaded environments based on this paradigm being also widely used nowadays. However, the implementation of this model does incur some overhead to applications that do not need activities to cooperate in very tied ways [24].

The last major model — the *remote procedure call* model — does not have the previous drawback. It is the basis of environments such as Nexus [16], Athapascan-0 [6], DTS [5] or PM^2 [25]. The basic idea is to provide a mechanism that allows the creation of remote threads to execute some given functions. The relationship between a thread issuing a (lightweight) remote procedure call and the thread spawned to undertake the call is exclusively a client/server one. Thus, the RPC mechanism can be considered as a "decomposition operator" that allows a computation to be split into several sub-computations, as opposed to the message passing mechanism that rather represents a mean of communication between cooperative threads.

4.3 Threads in data-parallel program executions

Due to the wide variety of functionalities provided by the previously mentioned multithreaded environments, it is not surprising that their use in distributed applications is growing more and more. This section relates the different works that have used threads for the execution of data parallel programs. The various works are sorted by the benefits they expect from threads.

Overlapping of communications by computations The first historical interest of having several threads per processor was to overlap communications by computations. When a computational flow (or thread) is blocked on a message reception, another local flow can go on. Much research [1, 2, 31, 27] has explored this field. However, [12] wonders about the success of this approach.

Super linear speedup Another advantage of the multithreading approach is that it may generate super linear speedups. The reason stems from cache effects. The memory accesses of the program are changed because a large domain has been split in smaller domains. Here, each thread allocates its parallel variables. So, it does not compute on a localized part of a larger array like the classical use of shared memory. Cache accelerations are reported in [2, 27, 28]. Cache effects have not been a reason of using threads but can become a good one. The current difficult point is to predict those cache effects.

Load balancing The mapping of threads to processors can also be used to improve the application load balance. Some research focuses on statically finding a good mapping [11, 35, 8]. It take place in the world of task scheduling and only works for (statically) predictable task graphs. [26] dynamically maps such threads to balance the load with a work stealing technique. But, it only works for a particular kind of

HPF nested loops. In [28], we have studied the feasibility of our approach. The paper focuses on HPF and shows that good performance seems to be possible with thread migration.

The previous benefit can only be met with some specific multithreaded environments. In particular, functionalities such as thread migration are seldom available together with high performance thread operations in classical environments. The next section presents PM^2, an environment providing these features that we have used in our implementations.

5 The PM^2 multithreaded environment

PM^2 (Parallel Multi-threaded Machine) [25] is a distributed multithreaded environment which was originally designed to support massively parallel irregular applications. These applications may be decomposed unpredictablely in a high number of concurrent tasks (each of them having a different execution time). As will be described in the following section, the programming model proposed by PM^2 to facilitate the design of such applications on distributed machines is centered around the idea of "*virtualizing*" the underlying architecture by using threads. This point makes PM^2 particularly well suited for the support of data-parallel programs through the concept of *virtual processes*.

5.1 Virtualizing the underlying architecture

The PM^2 programming model intends to provide an easy way of efficiently managing a large set of concurrent activities on a distributed architecture. Due to all the properties their model exhibits (see section 4.1), *threads* were naturally chosen to support the execution of these activities. In this respect, they play the role of virtual processors that may be requested dynamically.

Considering the fact that PM^2 intends to be a generic environment that may be used in very different application fields, it does not contain any fixed load balancing policy that would try to (automatically) ensure an efficient execution regardless of the application's specificities. In fact, it provides an interface allowing the explicit management of virtual processors across the architecture and it is up to the application programmer to implement the best suited "global scheduler" on top of this interface.

Efficiently managing a high number of running activities on a distributed architecture features many problems. In particular, one has to provide mechanisms allowing these activities to communicate with each other. A communication model based on *message passing* between threads was not considered for several reasons. The main one derives from the necessity to set up a global naming policy, so that threads can be reached from any node. In a context where thousands of threads may coexist, this may complicate the job of the programmer by forcing him/her to manage many threads names at the application level. In addition, this solution may complicate the implementation of load balancing mechanisms such as *thread migration* because the global naming would need to be strongly modified (and would become far less efficient).

For these reasons, the PM^2 programming model provides some *decomposition operators* that allow the spliting of computations into several parts and makes all communications between threads implicit. This way, relationships between threads remain simple and known by the system, so that operations such as thread migration can be implemented very efficiently. PM^2 provides two decomposition operators: the *Lightweight Remote Procedure Call* (LRPC) and *Thread Cloning*.

5.1.1 Lightweight Remote Procedure Call

A LRPC consists of forking a remote thread to execute a specified service. It can be either synchronous, asynchronous or synchronous with deferred waiting. Lightweight remote procedure calls need the following parameters: a *mode* (synchronous or asynchronous), a *service* number (identifying the service requested), a *location* (identifying the target node), a *priority* (of the remote forked thread), a pointer to the LRPC's *arguments* and a pointer to the LRPC's *results* (except in asynchronous mode).

A synchronous LRPC works like a classical procedure call, except that the code is executed by a remote thread (in another process). This mechanism is worth being used by a machine which is overloaded or when code needs to be run on a particular architecture. However, a thread executing a synchronous LRPC is blocked until it gets the result back. So a synchronous LRPC does not generate any additional parallelism, since it simply temporary moves an execution flow to another location.

An asynchronous LRPC initiates remote independent execution efficiently, because it only sends one message with no return of implicit results. In this respect, it's very similar to the *Remote Service Request* (RSR)

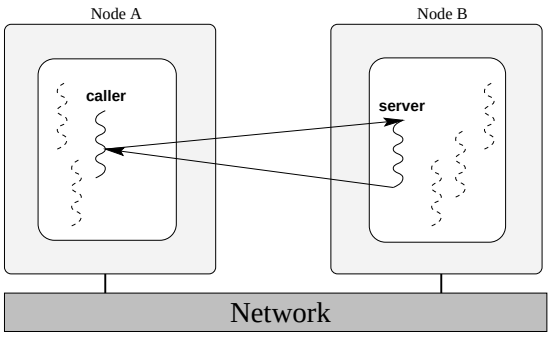

Figure 3: *The Lightweight procedure call mechanism*

mechanism available with the Nexus runtime system [16]. Of course, if needed, it would always be possible to return results later by doing another asynchronous LRPC, but it's not the right way to go since a third variant of LRPC is provided: the "deferred waiting" LRPC. This last mechanism allows one to decompose a LRPC in two steps: The LRP Call (asynchronous creation of remote thread) and the LRP Wait (waiting for results).

5.1.2 Thread Cloning

Thread cloning is a decomposition operator allowing "clones" to be generated from an original thread. After the execution of what we call a *splitting* instruction, all the resulting cloned threads have an identical execution context (i.e. instruction pointer and stack) inherited from the original thread. However, a *rank number* can be used by each clone to distinguish it from the others and perform different computations.

The splitting instruction can be compared to the UNIX fork mechanism, except that it applies to threads rather than to system processes and that it manages thread creations in a more controlled and secure way. In particular, a second instruction – the *merging* instruction – must be called by each clone at the end of the cloning operation. When all clones have called the merging instruction, only one thread is kept alive by the system to continue its execution (figure 4).

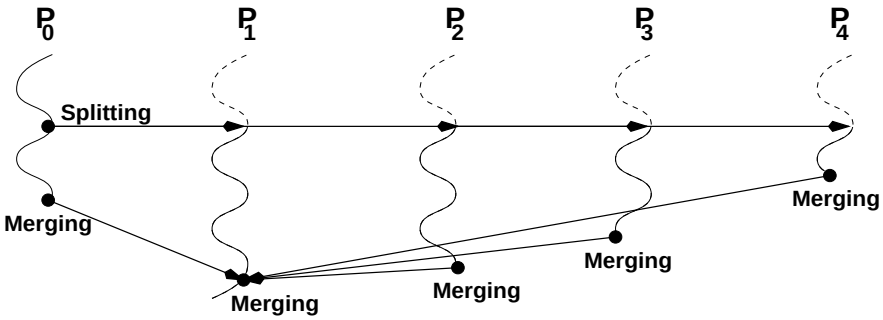

Figure 4: *The cloning operation. Here, the operation was initiated by thread p_0. After each thread has called the* merging *instruction, only one thread (p_1 in this case) continues its execution.*

5.2 Thread Migration

PM^2 provides a thread migration mechanism which allows one to move threads between nodes during their execution. This mechanism, triggered by a call to the `pm2_migrate` function, is done in three steps. First, the thread is frozen, and both its descriptor and the useful part of its stack are packed in a buffer; then, the buffer is sent to the destination module; and last, the thread's descriptor and stack are unpacked on the destination module, the stack is relocated in a new address space, and the thread is unfrozen (Figure 5).

Note that only the thread context is moved (stack, local data, etc) and that, after migration, the thread has access to a new global environment (global variables, etc). Moreover, due to the current implementation, thread migration is only possible between identical processes (same code on same architecture).

An application may not want to let all its threads be "migratable" at every time. For example, a thread issuing some complex operations on a global data structure should not be moved during this time. Such

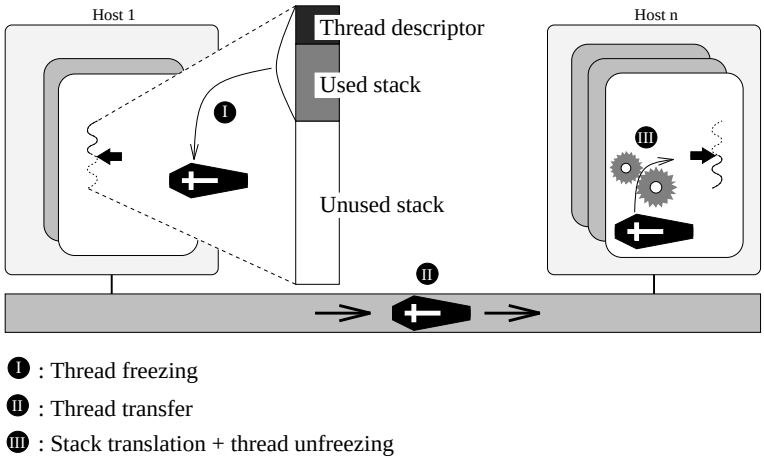

Figure 5: *The PM^2 migration mechanism*

a property has to be known by the runtime, especially when using application-independent load balancing libraries. Therefore, PM^2 attaches to each thread a "migratable" state that can be set (or unset) on demand.

5.3 Implementation

As with many other environments, PM^2 is a tradeoff between functionalities, portability and efficiency. Because PM^2 was designed to experiment with various load balancing strategies, priority was given to functionalities.

While LRPC and cloning primitives are the basis for computation distribution, flexibility of computation with PM^2 mainly depends on lightweight process functionalities. Existing threads packages are numerous. Some provide lots of functionality but lack efficiency or portability whereas others offer only minimal threads support but are very efficient. Some operating systems support lightweight processes (Solaris [32]) but each implementation does not provide the same functionality, so that it is difficult to build a portable platform upon these systems without losing interesting characteristics. Moreover, some specific features like scheduling algorithms or thread migration mechanisms may not be possible on top of *kernel level* thread libraries.

In order to exactly match our needs, we have implemented our own preemptive threads package, called MARCEL. It provides a large subset of the standard POSIX interface primitives with additional features, such as stack extensibility or thread migration support. For communications, the PVM message passing library was chosen. Since it is only used for implementation purposes, its interface is not directly usable.

6 Implementing virtual processes as threads with PM^2

Implementing a virtual process as a thread is straightforward except for efficient communications and thread migration issues. This section examines how communications between threads can be optimized and the compiler and thread environment support needed to achieve thread migration in data parallel programs.

6.1 Communications between threads

The simulation of message passing with asynchronous LRPC is more a benefit than a drawback. Indeed, the remote service can check if the destination thread is present and forward the message if the thread is no longer there. So, threads can migrate without worrying about messages because of a forwarding mechanism. Moreover, the remote service can directly empty the network into the right location.

If source and destination threads are not in the same memory address space, messages have to be generated to communicate. But, what do we do if they are on the same node ? One solution is to still use messages. Another solution is to allow threads to directly access the memory of other local threads.

The message passing approach has the benefit of being easier to implement. Indeed, wherever the destination thread is, a thread always generates a message. The call to the real communication layer can be avoided but in all cases memory copies and buffer management create overheads.

Direct memory access moves the complexity from the runtime to the compiler. The compiler has to manage the dependencies in a finer way. The communications that embed data exchange and implicit synchronizations

are replaced only by synchronizations resulting in a decrease in runtime overhead since we remove memory copies and buffer management.

6.2 Load balancing issues

One goal that we would like to achieve is to balance the load through thread migration. So, we are going to specify the characteristics of the compiler and the multithreaded environment needed to achieve thread migration.

The load of a process is the sum of the load of all its virtual processes. So, load balancing can be achieved by moving such entities from overloaded nodes to underloaded nodes. There are two main choices. Either we use a spy thread in each process or the compiler integrates some calls to the load balancing module in the code of computing threads. The role of the spy thread is to compute the local load, to exchange information with other spies and to preempt and migrate threads when needed. In the second case, the computing threads themselves take care of balancing the load through the calls to the load balancing module.

Without compiler assistance, preemptive thread migration is very complex because Fortran and C compilers may create many pointers when optimizing. With the current compilers, we do not know the pointers generated by compilers and thus we can not correct them after a migration. Therefore, only a thread can really safely call the migration function because we can generate code where such pointers do not need to be corrected. The problem is that threads can only migrate at well defined points.

Another challenge is to deal with the data allocated in the heap by threads. Currently PM^2 does not deal with such data but allows users to migrate it. In PM^2, user functions can be called before sending the migration message and after having rebuilt the thread. We use this functionality to pack to and to unpack from the migration message all the data allocated by the thread in the heap. It is also used to correctly set some pointers to allocated memory. It is not currently clear whether it is the compiler or the thread environment which should deal with such data and pointers.

6.3 PM^2 specialization

The fact that threads execute data parallel programs allowed us to specialize PM^2. Synchronizations and preemptions can be specialized for such programs in order to be more efficient.

Synchronization optimizations Local thread synchronizations are implemented as a two phase operation. Let n be the number of threads to synchronize. In the first phase, $n-1$ threads are going to be blocked, while in the second phase the nth thread releases the $n-1$ blocked threads. The releasing function can be specialized because we have set the priority of all computing threads to the same value. Thus, the releasing function is now one list insertion whereas before it was $n-1$ thread insertions. However, the synchronization still has a cost in $O(n)$ because of the first phase. The optimization only reduces the value of the multiplicative constant from 4.8 μs to 3.1 μs per thread on a Pentium 133 Mhz Linux machine. This represents a 35 % improvement. On this machine, the PM^2 thread context switch takes roughly 1.5 μs.

Preemption specialization For data parallel programs, the full preemption is not well suited. With it, threads progress with at most a time slice of difference. That can lead to a bad use of the overlapping mechanism because all the threads can reach the communication calls within a small amount of time. With n identical threads, this time is at most n times the time slice. The overlapping time is not as long as it could be. To maximize the overlapping time, a thread must not be preempted until it blocks. Nevertheless, the network has to be regularly polled to receive messages. The reduced preemption mechanism we have implemented only preempts a working thread to give the processor to the communication thread. Once the communication thread blocks on a message reception call, the last preempted thread gets the processor back.

6.4 Preliminary results

Our hand-written test program is the Gaussian elimination with a hand written runtime and a specific load balancing policy. This program was chosen because the `BLOCK` distribution is not well balanced and because we know the `CYCLIC` distribution is a good distribution. Thus, we can see how thread migration can balance the load. Figure 6 extracted from [28] reports the time for the Adaptor `CYCLIC` and `BLOCK` versions and the time for the multithreaded program with and without the load balancing module. We can see the cache effects, the

improvement generated by thread migration and the overhead generated by too many threads. The experiments have been done with monoprocessor Alpha machines connected with a gigaswitch network.

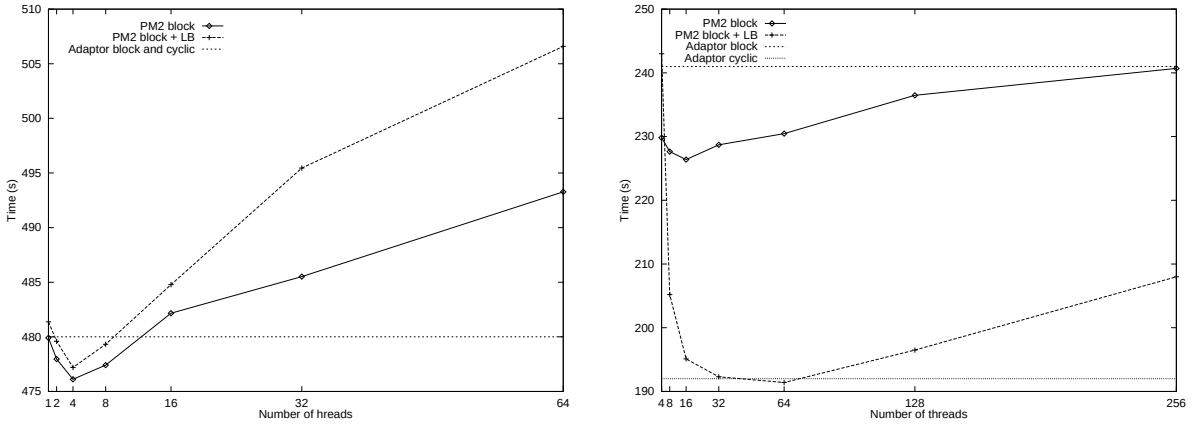

Figure 6: *Time in seconds for a 2048×2048 matrix with one processor (left) and four processors (right).*

7 Future Work

7.1 Evaluation of multithreaded data parallel compilers

We have developed two multithreaded distributed memory compilers by modifying the runtimes of two public domain distributed compilers: Adaptor [4] which is a HPF compiler and UNH-C* [19] which is a C* compiler. Adaptor is developed by T. Brandes at the GMD and UNH-C* by P. Hatcher at the University of New Hampshire. The multithreaded compilers are currently running and are in evaluation phase. This step is just a preliminary one. Compilers will need to be modified to generate better multithreaded code in order not to replicate scalar code, to optimize intra node communications and to offer a better support for thread migration.

7.2 Fissions and fusions

One major problem with virtual process level load balancing is to decide the number of virtual processes. With a large number, we should be able to find good load balanced mapping but with a large overhead. On the other hand, a small number of threads does not suffer a large overhead at the price of a bad load balanced execution [28]. An elegant solution would be to use fission/fusion operators. With them, we will be able to adapt the number of virtual processes. The fission operator splits one thread into several to decrease the granularity. This could be achieved with the thread cloning function. The fusion increases the granularity by merging several threads into one. All participating threads, minus the one, are destroyed and thus the overhead is decreased. The remaining deals with the data.

7.3 Efficient communications

The current implementation of PM^2 is layered on top of PVM for historical reasons. However, using PVM incurs some significant overhead because data is copied at several steps and because we have to use high-level polling algorithms to handle message arrival because PVM is neither thread-aware nor thread-safe.

This situation is about to change since we are currently working on the design of a minimal thread-aware communication layer that intends to be easily ported on different communication protocols such as Ethernet, ATM or Myrinet. The design of a prototype that will use SBP (*Streamlined Buffer Protocol* [20]) for Ethernet and Fast-ethernet networks as well as BIP [34] (*Basic Interface for Parallelism*) for Myrinet networks is under development. The general architecture of this layer is similar to the one used in Nexus [16] and further work may concern the fusion of these two libraries into a common one.

8 Conclusion

This paper describes an extension to the data parallel language compilation model to deal with three execution features: communication overlapping, load balancing without global data parallel object redistribution, and efficient use of clusters of uniprocessor and/or SMPs. The model is based on a view of the machine as a set of communicating nodes, and on the virtual process concept. We have shown that the HPF and C* compilation models can be easily extended. We propose to implement a virtual process into a thread.

After having defined threads and the different solutions proposed for distributed memory multithreaded environment, we focus on a particular model of distributed memory multithreaded environment, represented by PM^2, which fixes the context. Then, we study the interactions between such an environment that brings many functionalities like LRPC, thread migration, thread cloning and data parallel programs. In particular, we study their utilization for load balancing purposes. We also describe two specializations of PM^2 to show how a general environment can more efficiently support data parallel programs.

Future works will concern intensive experimentation of our two existing multithreaded data parallel compiler prototypes as well as the evolution of the PM^2 implementation in order to efficiently exploit high bandwidth networks. All the described functionalities have been implemented except the fission/fusion, although PM^2 supports thread cloning. Preliminary experiments seem to indicate a good behavior except for scalar code, because scalar code is replicated and may generate too much overhead.

Although this direction seems promising, many questions and problems appear. For example, what is the interface between user code and the load balancing module ? What must be done at user level and what can be done by the compiler and/or the runtime ? What features belong to the runtime and what can be handled by the multithreaded environment ? Cooperation seems a partial preliminary answer.

Acknowledgment

We would like to thank Luc Bougé and Jean-François Méhaut for their active contribution to this work. Special thanks to Philip J. Hatcher for the long discussions about C*.

References

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

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

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOP'95 proceedings ACM SIGPLAN*, Santa Barbara, 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. Bubeck and W. Rosenstiel. Verteiltes Rechnen mit DTS (Distributed Thread System). In Marc Aguilar, editor, *Proc. '94 SIPAR-Workshop on Parallel and Distributed Computing*, pages 65–68, Suisse, October 1994. Fribourg.

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

[7] J.N. Colin. *DTMS : Un environnement pour la programmation distribuée à grain indéterminé.* PhD thesis, Université de Mons-Hainaut, 1995.

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

[9] A. Darte and G.-R. Perrin, editors. *The Data-Parallel Programming Model: A Semantic Perspective*, volume 1132 of *LNCS Tutorial*. Springer Verlag, June 1986.

[10] B. Dreier and M. Zahn. RThreads — a Uniform Interface for Parallel and Distributed Programming. In *Proceedings of the Second International Conference on Massively Parallel Computing Systems (MPCS'96)*, pages 557–561, Ischia, Italy, May 1996.

[11] H. El-Rewini, H.H. Ali, and T. Lewis. Task scheduling in multiprocessor systems. *Computer*, December 1995.

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

[13] A. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Proc. of IEEE High Performance Distributed Computing*, pages 211–218, Washington, 1995. D.C.

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

[15] I. Foster and K.M. Chandy. Fortran M: A language for modular parallel programming. *Jounal on Parallel and Distributed Computing*, (25(1)), 1994.

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

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, 1994.

[18] M. Haines, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Proc. of Supercomputing'94*, pages 350–359, Washington, November 1994.

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

[20] P.J. Hatcher, R.D. Russell, S. Kumaran, and M.J. Quinn. Implementing data-parallel programs on commodity clusters. In *Spring School of Data Parallelism*, Les Menuires, France, March 1996.

[21] P. Mehrotra and M. Haines. An overview of the Opus language and runtime system. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers*, New York, November 1994.

[22] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, March 1994. available from netlib2.cs.utk.edu.

[23] F. Mueller, EW. Giering, and TP. Baker. Implementing ada 9x features using posix threads: Design issues. *TRI-Ada*, September 1993.

[24] R. Namyst. *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* PhD thesis, Université des Sciences et Technologies de Lille, Janvier 1997. Available at `http://www.lifl.fr/~namyst/these.html`.

[25] R. Namyst and J.F. Mehaut. PM^2: Parallel Multithreaded Machine. a computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)*, pages 279–285. Elsevier Science Publishers, Sep 1995.

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

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

[28] C. Perez. Load balancing HPF programs by migrating virtual processors. In *Second International Workshop on High-Level Programming Models and Supportive Environments, HIPS'97.* IEEE Computer Society Press, April 1997.

[29] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, and M. Weeks. SunOs 5.0 Multithreaded Architecture. In *Proceedings of the Winter 1991 USENIX Conference*, pages 65–79, 1991.

[30] Michael J. Quinn and Phil J. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *Journal of Parallel and Distributed Computing*, March 1996.

[31] 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.

[32] SunSoft. *SunOs5.2 Guide to Multi-Thread Programming*, 1993.

[33] Thinking Machines Corp. *C* programming guide*, November 1990. Version Number 6.0.

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

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

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