Para++: C++ bindings for Message Passing

Olivier Coulaud, Eric Dillon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Para++: C++ bindings for Message Passing

O. Coulaud and E. Dillon

N ° 3116

Février 1997

THÈMES 1 et 4

ISSN 0249-6399

Para++: C++ bindings for Message Passing

O. Coulaud and E. Dillon

Thèmes 1 et 4 — Réseaux et systèmes — Simulation et optimisation
de systèmes complexes
Projet Numath, Resedas

Rapport de recherche n ° 3116 — Février 1997 — 18 pages

Abstract: This paper describes a high level C++ library for message passing applications. Our interface is build on top of PVM and MPI. The goal of this interface is to allow a quicker design of parallel applications without any important drop of performances. We introduce two levels of tasks and use C++ streams for communications. We also present a performance study over both PVM and MPI to show the overhead of our implementation. Finally, we detail two applications based on the heat equation to explain how our library can be used for SPMD and MPMD applications.

Key-words: C++ bindings, message passing, parallel computing

(Résumé : tsvp)

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Para++ : une interface pour l'échange de messages

Résumé : Ce papier décrit une interface C++ de haut niveau pour la programmation par échange de messages. Cette interface est implantée au dessus de PVM et MPI. L'idée de cette interface est de permettre le développement rapide d'applications parallèles sans introduire de chute de performances notable. Nous définissons une hiérarchie à deux niveaux entre tâches et utilisons le mécanisme de "stream" C++ pour les communications. Nous présentons aussi ici une étude de performances au dessus des deux implantations PVM et MPI pour mesurer le surcoût introduit par `Para++`. Enfin, nous détaillons deux applications basées sur la résolution de l'équation de la chaleur pour expliquer concrètement comment notre bibliothèque C++ peut être utilisée à la fois dans le cadre d'application SPMD et MPMD.

Mots-clé : Interface C++, échange de messages, calcul parallèle

1 Introduction

As a consensus by now, parallel computers and network of workstations appear as a good opportunity to develop high performance systems and to solve large problems in many application areas. Problems however remain when it comes to efficiently using these novel architectures : the design of programming languages and software tools is essential.

Indeed, efficiency not only consists in finding the fastest architecture to run a particular application, but also consists in reducing the design time of a parallel application and making it easier for new users to turn to parallel programming.

Many programming methods have already been explored to use parallel architectures. A first approach is given by the use of parallelization tools: feeding these tools with sequential codes, parallel codes are generated to get more efficiency on parallel computers. This approach however remains dedicated to specific sequential codes.

Parallel languages offer other opportunities for parallel programming by providing more abstract models. Indeed, they offer various features to allow the design of parallel codes. In such cases, abstraction is usually reached by hiding explicit communications and synchronizations in a particular programming paradigm. PCN [7], Compositional C++[3], High Performance Fortran (HPF) are part of this approach. Unfortunately, how efficiently this approach can be implemented remains an open question.

The last approach is represented by low level programming languages. These languages are based on explicit mechanisms for data distribution, communications and synchronizations. Among them a special set of models have been developed, with coarse grain processes, which can be basically represented by both PVM [10] and MPI [9] libraries. The reason why such libraries have become really popular is that they were primarily developed to harness the unused power of networks of workstations. Since such networks are by far the most common type of parallel systems widely available today, a lot of users have tried them before actually using them on parallel architectures.

Anyway, even if they have become popular, they suffer from several drawbacks: explicit message passing and decomposition, and above all a need for tuning to get even reasonable performances.

Finally, several extensions to sequential languages have been proposed. Based on Fortran, lots of parallel extensions have appeared. The most popular are Fortran D[8], Fortran M[6], HPF.

Looking at C++ language it gets really harder to design an efficient parallel extension. Few extensions already have been designed, all based on different concepts. Some of them have emerged, namely CC++[3], pC++[1] or μC++[2]. More formally, a US consortium called High Performance C++ (HPC++[1]) also tries to define a standard model for parallel programming based on C++. This model is aimed at enabling the writing of portable parallel applications and at providing a target language for vendors allowing compiler optimizations. Moreover, this language is intended to support both data parallelism and control parallelism. In Europe, the Europa Working Group on ||C++[2] roughly works on the same topics.

[1] see `http://www.extreme.indiana.edu/hpc++`
[2] see `http://www.lpac.ac.uk/europa`

An alternative to these languages is the use of sequential languages like Fortran77, C or C++ extended with the required features of parallel applications: i.e. task management, synchronization, communication between tasks. On one hand, the advantage of this approach is that it only requires the use of a message passing library to reach parallelism. On the other hand, based on this distributed memory paradigm, the data locality remains explicit, allowing thereby easy tuning. Finally, in practice the portability of such a language only depends on the availability of message passing libraries on all target architectures.

At this time, only two libraries need to be considered :

- MPI [9] is the emerging standard but all features will only be available in the MPI-2 version (e.g. dynamic and one-sided operations).
- PVM [10] has become the *de facto* standard thanks to its wide portability, and its use within both industrial and academic laboratories.

Finally, the choice of the sequential language to be extended should be guided by the level of abstraction, portability and ease for programming, which directly leads us to C++.

`Para++` stands for "C++ binding for message passing". It is not only a parallel extension of C++ or a layer over PVM and MPI. It is a way to simplify the design step of parallel applications by adding a task hierarchy and considering communications as "C++ streams", just as any other C++ I/O.

Broadly speaking, `Para++` offers two main advantages :

- the definition of a tasks hierarchy. Whereas all tasks are identical within PVM or MPI, "Master" and "Slave" tasks have been explicitly introduced within Para++. They hold different features and properties to fit the SPMD and a "Multiple SPMD" model. By the way, it allows a more simplified and structured view of parallel applications.
- the use of a stream-based interface for inter-tasks communications. Whereas all C++ inputs/outputs are handled through so-called C++ "streams", Para++ provides new dedicated streams called "`pout`" and "`pin`" (remaining `cout` and `cin`, for parallel-out and parallel-in) to handle inter-tasks communications. Thanks to this notation, it allows a very light-weight syntax for all communications.

However, `Para++` differs from other proposals for C++ programming with message passing. For example, whereas `oompi` [11] is aimed at providing a specific C++ interface dedicated to MPI, `Para++` rather provides a higher level of abstraction for general message passing use. Moreover, `Para++` also provides portability: many users are currently wondering whether they should choose PVM or MPI for the design of their parallel application. `Para++` has been implemented on both of them, so that the code can be developed independently from any of them, and run it on the most efficient one according to the available architecture.

Finally, `Para++` has been developed so that only a low overhead is introduced to the top of MPI or PVM library calls, so that efficiency can still be guaranteed.

In the first part of this paper we will see both models of tasks supported by our C++ bindings. In the next part, communications will be described. Then, we will give a survey of `Para++`'s overhead

over both PVM and MPI libraries. Finally we will present a numerical application developed using `Para++`.

2 Task Model of Para++

Para++ was first developed to design parallel applications according to the SPMD programming paradigm [4]. Further, we extended our model to allow a MPMD-like model of programming.

In the following sections, we will first focus on the SPMD feature of Para++ before further explaining the MPMD-like extension we added.

2.1 SPMD model within Para++

The basic idea of Para++ was to allow the programming of SPMD applications using C++ language and message passing. However, both main message passing libraries (namely PVM and MPI) do not provide any C++ interface[3] that would allow for an easy syntax for message passing. With Para++, we provide an abstract view of message passing primitives, through a set of C++ classes, abstract enough to allow their implementation on top of any message passing library. As a consequence, it allows the use of message passing operations within C++ language through a simplified syntax, and a higher level of abstraction. Indeed, we provide objects to represent SPMD tasks, as well as objects for communicating with each other. Let's focus on the tasks representation first, communication objects are discussed in section 3.

The root object in Para++ is called `ParaProcess`. This object is the representation for a SPMD task. As a consequence, it has to be instantiated by each task involved in a SPMD application. This object holds informations and methods to allow all tasks to run the same executable code, accordingly to the SPMD programming paradigm. Finally, since each task involved in a SPMD application must hold such a `ParaProcess` object, a whole SPMD application might be represented by this `ParaProcess` object, where all tasks run the same C++ code : that is the reason why we call SPMD tasks, "`ParaProcess` tasks" within Para++.

To get started with Para++'s syntax, here is an example of a Para++ SPMD application. The skeleton of such an application is given in figure 1.

This example only shows the use of `ParaProcess` objects. On this example, you might see that the programming of a SPMD application remains very simple.

The Para++ program must begin with the instantiation of a `ParaProcess` object, before calling an `init(..)` method on it. This call is responsible for the spawning of all tasks involved in the application : in this example, n tasks have to run for the SPMD computation.

As you can see on this example above, all tasks involved in the SPMD application run exactly the same code : the `ParaProcess` object hides the "starting step" of the application from the user. More clearly, you do not have to make explicit calls to `pvm_spawn` anymore, as required with PVM.

Finally, the `init(..)` method is a key method for Para++ programs, since it is responsible for the correct launching of the whole application : it has to be called at the very beginning of the

[3] C++ bindings are likely to be included in the next version of MPI, but remains very close to the C bindings.

```
#include "Para++.hh"
main(int argc, char ** argv)

    ParaProcess p("pi", argc, argv);
    ...
    p.init(n);
//   end of SPMD initialisation
    ...
    p.end();
```

Figure 1: Skeleton of a SPMD application with Para++

program. Moreover, it performs a synchronization across all tasks of the application, to ensure that they will all begin computation at the same time. In the same way, the `end(..)` method, is to be called at the end of the application, to ensure that everything ends well before leaving.

So, the `ParaProcess` objects hide everything about the beginning and the ending of SPMD applications. As a consequence, it not only simplifies the syntax but also ensure that everything is ready for actually beginning computations. Moreover, this notation is really satisfying with the SPMD programming model since all tasks seem to run exactly the same code. This remark becomes a true advantage for beginners in message passing programming, since two operations only are required to have the whole set of co-operating tasks ready for computation: a `ParaProcess` instantiation and an `init(..)` call.

2.2 MPMD extension to Para++

In the previous section, the SPMD support was presented. It is mainly based on the definition of `ParaProcess` objects used to represent a set of SPMD tasks.

To allow both MPMD and dynamic features in Para++, we introduced a hierarchy in tasks, giving the so called `ParaProcess` tasks, the ability to dynamically spawn other tasks or set of tasks.

The first aim of this extension was to enable Para++ tasks to run different executable codes.

When using PVM (and now MPI-2) each task is able to create a new process by calling a `spawn` function. Within Para++, we wanted to clarify this point, by allowing only selected tasks to create new tasks. So, we introduced new objects called `ParaSlave` to represent tasks that are not allowed to create new tasks. More precisely, this means that two kinds of tasks may exist within a Para++ application:

- top level tasks, instantiating `ParaProcess` objects. These tasks are allowed to dynamically create new tasks.

- second level tasks, or *slave* tasks, instantiating `ParaSlave` objects. Moreover, these `ParaSlave` tasks must be the result of a process creation from a top level task.

Process creation is done within the `ParaProcess` objects thanks to a dedicated method called `startSlave`. This method allows the creation of a set of tasks running the same executable code. Notice that this method is not available on `ParaSlave` objects.

Finally, we can summarize the tasks hierarchy of a Para++ application by figure 2

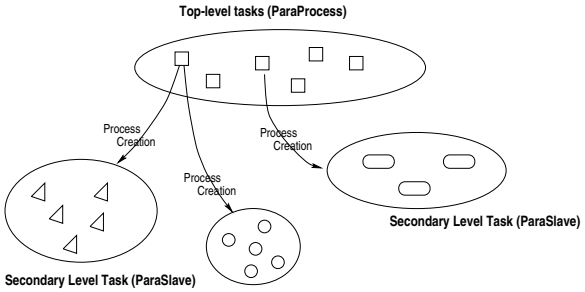

Figure 2: Tasks hierarchy of a Para++ application

In this figure we can see both level of Para++ tasks. All tasks of the first level are started together during the `init(...)` method of the `ParaProcess` object, whereas all "slave" tasks (secondary level) are started by the former ones.

Finally, this scheme leads us to an extended MPMD model in Para++. What we have here is more something of a "M-SPMD" (Multiple - SPMD model) since all tasks are grouped in sets : all tasks of a single set run the same executable code.

Moreover, this M-SPMD model allows us to identify Para++ tasks more easily. Each set of tasks is attached a `ParaContext` object, that is instantiated within each task of a set. After that, each task has a rank within this "context", so that each task can be identified by its rank and its context. This identification scheme is detailed in section 3.

As a conclusion, the tasks model provided within Para++ is aimed to simplify the structure of message passing applications. Tasks are grouped by executable code and hierarchied by the object they instantiate (`ParaProcess` vs `ParaSlave`).

3 Para++'s communication interface

Para++'s communication interface is based on the introduction of new C++ streams that can be used like `cout` or `cin`. The use of these streams is linked to the notion of contexts, since a task is represented by its instance in a context. Because we have two level of tasks (see Fig. 2) we also have two kinds of communication as described in figure 3. When two tasks in a same local context want to communicate we will speak of `intra-context` communications while if we have two

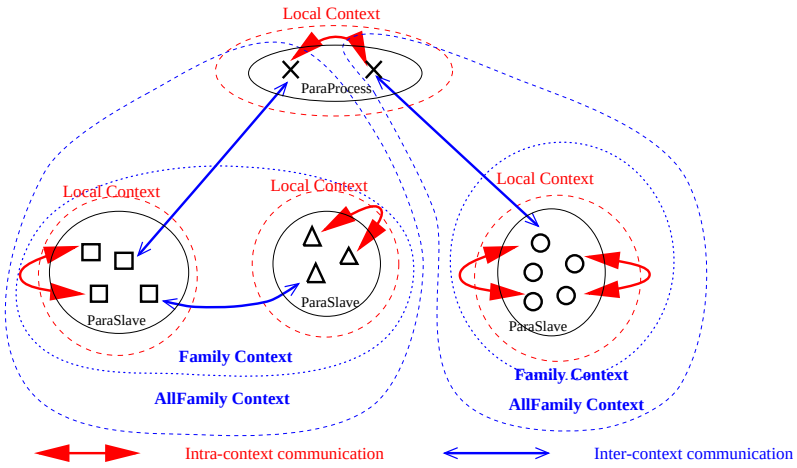

Figure 3: Communication class

tasks in the same family context but in two different contexts we will speak of `inter-context` communications.

So, `Para++` provides two new specific streams to exchange data with other tasks:

- **pout**: to asynchronously "send" values to another task.
- **pin**: to receive values from another task, as a blocking operation.

Both those objects can be used like `cout` and `cin` provided by C++ (see [5]).

In the following subsections, we will first present the intra-context communications (where the communicating tasks belong to the same context). After this, we will present the extension to handle inter-context communications and finally we describe a non-blocking receive.

3.1 Intra-context communications

The `pout` and `pin` streams implement an asynchronous send and a blocking receive: i.e. the `pout` will completed as soon as the message is ensured to be delivered, and the `pin` will block until a matching `pout` is issued.

Here's a little example that shows how to use these two objects within the same context

```
#include "Para++.hh"
main(int argc, char **argv)
{
  int n, ntask = 2;
  ParaProcess p("dummy", argc, argv);
  p.init(ntask);  // To start 1 more task
```

```
  float x;
  Vector<double> V(10) ;
    ...
  if (p.master())
      {
    ...
// task number 0 sends a
//      message to task number 1
        pout(1) <<  n << x << V
                << flush();
    ...
      }
  else
      {
    ...
// task number 1 receives
//  a message from task number 0
          pin(0) >> n >> x >> V ;
    ...
      }
  p.end();
}
```

The sending of a message takes a three steps process :

step 1
First we initiate the communication with the "`()`" operator. Here, this way we specify the receiver by its instance.

step 2
Then we pack values by successive calls to the << operator, in order to have the whole message contiguously stored in a buffer.

step 3
Finally, we send the message by using the `flush` method.

So, the line in the above example :

```
pout(1) << n << x << V <<flush;
```

sends the value of integer `n`, of float `x` and of Vector of double `V` to task number 1. We can also split the line in order to follow the above construction of the sending message as follows

```
pout(1) ;
pout << n << x << V ;
pout.flush() ;
```

On the other side, the receiving of a message is implemented in a two steps process :

step 1
First we initiate the communication with the "`()`" operator. Here, this way, we specify the sender by its instance and its context.

step 2
Then we unpack values by successive call of >> operator.

Finally, the instruction

```
pin(0) >> n >> x >> V ;
```

means that the values of `n`, of `x`, and of Vector `V` will be received from task `0`.

Remark 1 *The task which rank is 0 is always considered as "master" task of a local context.*

Finally, `pout` and `pin` are not only dedicated to point to point communications. The parameter passed to the `pout` operator identifies the target tasks. The parameter passed to the `flush()` method might be used to "tag" messages. All possibilities are summarized in table below :

	flush()	flush(tag)
`pout()`	untagged broadcast	tagged broadcast
`pout(P)`	untagged send to P	tagged send to P
`pout(Dests)`	untagged multicast	tagged multicast

Table 1: Communications in the intra-Context

In the above table `P` is the identifier of a task and it is an integer between 0 and the number of tasks minus one, `Dests` is a vector of task identifiers.

3.2 Inter-context communications

Due to the dynamic aspect of process creation, inter-context communications require an initialization step.

In practice, to have all "context" interconnected and available, the user must call the "getForeignContext" method on each `ParaProcess`/`ParaSlave` object within each task. Of course, these calls must only be done when all tasks are correctly spawned, i.e. after all "`startSlave`" and "`init()` calls.

After that, inter-context communications are possible. The only difference between inter-context communication and intra-context communication is in the first step of the construction of sending message. In fact, we have to add to the instance of the target task its context in the `()` operator as follows

```
pout(2,CtxtSolver) ;
```

To obtain the foreign context `CtxtSolver` we use the `slaveContext` method as follows

```
ParaContext CtxtSolver = p.slaveContext(0) ;
```

where p is a `ParaSlave` object, `ParaContext` the context class and the `slaveContext(0)` method returns the context of the first slave group of tasks created by a `startSlave` method.

We describe in the following table all the possibilities of communications through contexts

	flush()	flush(tag)
`pout(ctxt)`	untagged broadcast to all tasks in the context ctxt	tagged broadcast to all tasks in the context ctxt
`pout(P, ctxt)`	untagged send to P located in the context ctxt	tagged send to P located in the context ctxt
`pout(Dests, ctxt)`	untagged multicast to Dests in the context ctxt	tagged multicast to Dests in the context ctxt

Table 2: Communications in inter-Context

In the above table `P` is the identifier of a task and it is an integer between 0 and the number of tasks minus one, `Dests` is a vector of task identifiers and `ctxt` is a `ParaContext` object.

Remark 2 *Two tasks in different contexts can communicate if and only if they belong to the same AllFamily context.*

3.3 Non Blocking receive

The `pin` object implements a "blocking" receive: `pin` will block while waiting for a matching message (i.e. from the right sender and with the right tag). For non blocking receive we have introduced the `arrived(...)` method. This method probes the arrival of a message and returns the tag and the sender. When the message is arrived, it must be actually received by a `pin()` call as seen in the previous section.

The example below shows how the `pin` object can be used to handle non-blocking receives.

```
...
while ( !pin.arrived(t1,tag))
  {
     ...
  }
pin(t1,tag) >> n;
   ...
```

In this case, the program can execute actions while waiting for a message from task `t1` tagged with `tag`. When the message has arrived we can receive it with the `pin` object as describe above.

4 Performance study

In this section we try to quantify the overhead introduced by Para++ interface over MPI or PVM libraries.

To do so we will present performance measurements on various architectures supported by Para++. We will mainly focus on Para++'s latency and bandwidth compared to PVM's and MPI's.

To get those results we used the *ping-pong* algorithm. Basically, it consists of two tasks : the first one sends messages to the second one, whereas the later simply echoes the message. We compute the time spend by the message from end to end, by dividing the round trip time by two. Thanks to this algorithm, we are able to compute both *latency*[4] and the maximum bandwidth (or throughput)[5]. We implemented this algorithm with MPI, PVM and Para++ and compared the results.

In the following sections, we will first focus on the use of Para++ on top of PVM, before presenting the same experiments on top of various implementations of MPI.

4.1 Performances of Para++ on top of PVM

To evaluate the overhead of Para++ on top of PVM, we run the ping-pong algorithm on two Sun UltraSparc 1/170, interconnected by a standard 10 Mbits/s Ethernet network.

The results of the experiments carried out with PVM are presented on table 4.1.

`Implementation`	Latency (ms)	Bandwidth (Mbits/s)
`Para++ over PVM`	0.393	9.0
`PVM (3.3.10)`	0.390	9.1

Table 3: Performances comparisons with PVM

Clearly, on these results, we cannot see any obvious overhead when using Para++ on top of PVM. Basically, we achieved such performances by inlining all Para++ functions.

In fact, the syntax of Para++ can directly be mapped on PVM's: the `pout()` call represents the `pvm_init()` call, the successive "<<" operators are mapped to PVM's packing functions, whereas the final `flush` is the `pvm_send` operation. The same holds on the receiving side.

4.2 Performances of Para++ on top of MPI implementations

To evaluate the overhead introduced by using Para++ on top of MPI implementations, we carried out the same experiment on various MPI implementations and architectures.

We mainly used two configurations for experiments :

Case One : two Sun UltraSparc 1/170, interconnected by a standard 10Mbits/s Ethernet network, using LAM 6.1 implementation of MPI, with all optimisations flags set (`-c2c, -O, -nger`).

Case Two : two R8000/90Mhz processors of a SGI PowerChallenge machine, using SGI's implementation of MPI.

The results we got are presented in table 4.

[4] defined by the time spend to send a zero-sized message

[5] defined by the maximum number of bits that can be sent by second

Implementation	Latency (ms)	Bandwidth (Mbits/s)
`Para++ over LAM61`	0.374	7.1
`LAM61`	0.268	9.2
`Para++ over SGI MPI`	0.074	225
`SGI MPI`	0.030	501

Table 4: Performances comparisons with MPI

These results are slightly different from PVM's. Regarding the latency, Para++ only introduces a small overhead, but Para++ definitely involves a drop of the maximum bandwidth. These results are emphasized with the experiment on the PowerChallenge, since the MPI communications are implemented through shared memories.

Conversely with PVM, each Para++ call is not exactly mapped to a MPI call. Firstly, when implementing the ping-pong algorithm with MPI, we only need the `MPI_Send` and `MPI_Recv` functions, whereas when using Para++ we indirectly use the `MPI_Pack` and `MPI_Unpack` functions. So in this case, we introduce one more buffer-copy with Para++.

However, thanks to inlining, the latency is not too affected, but regarding the bandwidth, Para++ may really lack performances.

5 Applications with Para++

We present now two academic applications : the first one using the SPMD model and the second one with our M-SPMD model to see how we can use communications through different contexts.

5.1 The heat equation

For a SPMD example, we consider the problem of the heat equation in the unit square with Dirichlet conditions

$$\left\{ \begin{array}{lcll} \dfrac{\partial u}{\partial t} - \dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} & = & f(x,t) & \text{in } \Omega = [0,1]\text{x}[0,1], \\ u(x,t) & = & 0 & \text{on } \partial\Omega, \\ u(x,0) & = & u_0(x). & \end{array} \right. \tag{1}$$

The heat equation is discretised in time by the explicit Euler scheme and in space by the classical center finite difference scheme at 5 points on a uniform cartesian grid. In two dimension of space the scheme writes

$$\left\{ \begin{array}{l} \text{for i = 0 to NG-1 do} \\ \quad \text{for j = 0 to NG-1 do} \\ \qquad U_{i,j}^{n+1} = U_{i,j}^n + \frac{\delta t}{h^2}(U_{i+1,j}^n + U_{i,j+1}^n - 4U_{i,j}^n + U_{i-1,j}^n + U_{i,j-1}^n) \end{array} \right. \tag{2}$$

where δt the time step, $h = 1/NG$ is the space step, NG the number of points in x-direction and in y-direction, $U^n_{i,j} = u(x_i, y_j, n\delta t)$ the value of the solution at point $(x_i = ih, y_j = jh)$ and at time $n\delta t$, (x_i, y_j) the grid of collocation points.

We decompose the collocation grid in PxQ subgrids of dimension NxM with $N = NG/P$ and $M = NG/Q$. Now we map the grid divided in block on a grid of (P, Q) processors. Moreover to use formula (2) on each subgrid we add ghost point as follows

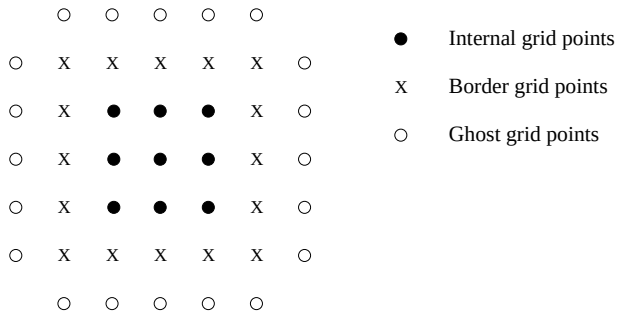

Figure 4: A subgrid

In fact the ghost points are duplicated on each processors so the dimension of the subgrid is now $(N + 2)$x$(M + 2)$.

Before using the scheme (2), we update the value of U on ghost points and then we apply the above formula. To update the value we first consider Blocking communication.

step 1
a) Send the values to North, South, East, West.
b) Update the ghost point.

step 2
Apply formula (2) on border and internal points.

Algorithm 1

The skeleton of the program is given by figure 1 and the key point of algorithm 1 is the following part

```
if( VoisinEast >= 0 )
  {
    pout(VoisinEast) << U(Index_E) << flush(102);
    pin(VoisinEast,103) >> buff2 ;
    for ( j=0 ; j< buff2.numElts() ; j++ )
      { U(Index_E(j)+1) = buff2(j) ; }
  }
if( VoisinWest >= 0 )
  {
```

```
    pin(VoisinWest,102) >>  buff2 ;
    pout(VoisinWest) << U(Index_W) << flush(103);
    for ( j=0 ; j< buff2.numElts() ; j++ )
      {U(Index_W(j)-1) = buff2(j);}
}
                    Update West and East ghost points.
```

If we have a neighboring at the east we send the value of U located on the border points to it with tag `102`. Then we wait from the east the value of ghost points with tag `103`. At last, we put these values in U.

For "non-blocking" communication we modify the loop in time as follows

step 1 Send the values to North, South, East, West.

step 2 Apply formula (2) on internal points.

step 3 Update the ghost points

step 4 Apply formula (2) on border points.

Algorithm 2

The main different between algorithm 1 and algorithm 2 is in step 4 where we use our non-blocking receive.

```
while ( cpt != cptMax)
{
  if( pin.arrived() )
   {
     tag = pin.tag() ;
     if( tag == 103 )
      {
        pin(VoisinEast,103) >> buff2 ;
        ADD   = 1 ;
        Index = &Index_E ; buff = &buff2 ;
      }
     if( tag == 102 )
      {
        pin(VoisinWest,102) >>  buff2 ;
        ADD   = -1 ;
        Index = &Index_W; buff = &buff2 ;
      }
     for ( j=0 ; j< buff->numElts() ; j++ )
      {
         U((*Index)(j) + ADD) = (*buff)(j);
      }
     cpt += 1;
```

```
    }
}
```

where `cptMax` is the number max of neighboring of the subgrid.

We look if a message has arrived from anywhere with any tag and if one is present, we get its tag and then dispatch the work to do.

This example of three terms formula could be easy generalized to recurrent formula as

$$U^{n+1} = F(U^n)$$

where F is a function of the elements of U^n.

5.2 A M-SPMD application

We present now an academic application using the M-SPMD model. Our goal is to solve and to visualize the evolution of the solution of the heat equation (1).

This application is split into three specific applications.

- the first one is the master: it will be implemented as a `ParaProcess` task, and will be responsible for the slaves spawning.
- the second one is the solver: it will be implemented as a `ParaSlave` task (or set of tasks). This solver will be spawned by the master and will perform all computations. After each computation step, the data will be passed to the visualization tasks through inter-context communications.
- the third one is the visualizer: it will be implemented as a `ParaSlave` task (or set of tasks), and be spawned by the master. It will get data from the solver using inter-context communications.

The solver application is the one we described in the previous section. The goal of the master application is to spawn the two specific applications and then broadcast informations like the number of collocation points, N, and the number of processors if the other applications are parallel. Afterwards, the solver task computes the solution by the above formula and at each iteration in time, it sends the value of the solution, U, to the post-processing task which is in charge of the visualization of the solution.

Fig. 5 shows the skeleton of the three programs of our application. Let us first describe the master's task. In the above figure in **1** we start four tasks "solver" with parameter "2 2 20 20". All of these tasks are unrolled in the slave group which is numbered `0`. Then in **2**, we spawn four tasks "postraitment", and all of these slave tasks belong to the slave group `1`. Now in **3** each tasks call the `getForeignContext()` method to share all contexts. After those calls two tasks in different contexts can communicate. In **4** we give an example of data exchanges between two different contexts. First we search the context of the other slave group, then each "solver" task sends to its instance in the other context four integers. Before closing with the `end()` method, we synchronize all tasks in the same family with `syncWithAllFamily()` method.

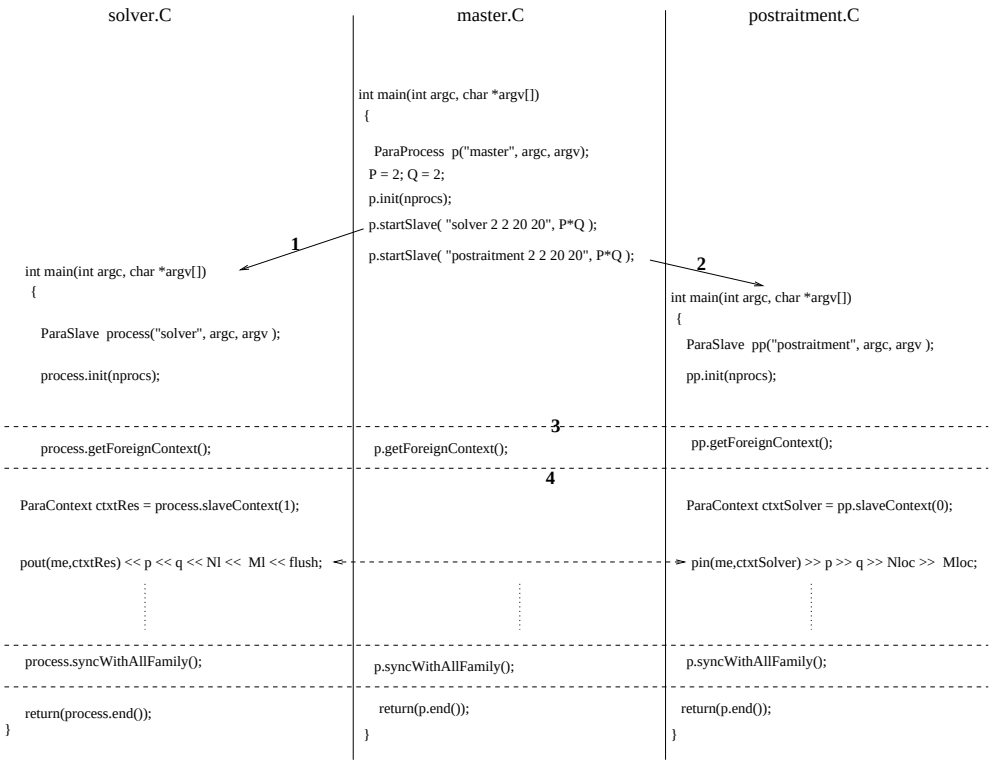

Figure 5: Application

References

[1] F. Bodin, P. Beckman, D. Gannon, S. Naranaya, and S. X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[2] P. Buhr and R. Stroobosscher. μc++ Annotated Reference Manual. Technical Report v4.2, University of Waterloo, jan 1995.

[3] M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report CS-TR-92-13, California Institute of Technology, septembre 1992. http://www.compbio.caltech.edu.

[4] O. Coulaud and E. Dillon. Para++: C++ Bindings for Message Passing Libraries. Technical Report RT-174, INRIA, June 1995. http://www.loria.fr/para++/parapp.html.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1994.

[6] I. Foster, R. Olson, , and S. Tuecke. Programming in Fortran M. Technical report, Argonne National Lab., 1993. ftp://ftp.mcs.anl.gov/pub/fortran-m/reports.

[7] I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Apporach. Technical report, Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439.

[8] G.C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. The Fortran D Language Specification. Technical Report CRPC-TR90079, Rice University, 1990. (revised April 1991).

[9] MPI Forum. MPI: A Message Passing Interface Standard. University of Tennessee, June 1995.

[10] PVM Team. PVM 3 users's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September 1994. Available via NetLib.

[11] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of POOMA'96, Parallel Object-Oriented Methods and Application Conference*, Santa Fe, New Mexico, feb 1996.

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

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399