HAL
archives-ouvertes.fr

PARA++ : C++ Bindings for Message Passing Libraries : User Guide

Olivier Coulaud, Eric Dillon

▶ To cite this version:

Olivier Coulaud, Eric Dillon. PARA++ : C++ Bindings for Message Passing Libraries : User Guide. [Technical Report] RT-0174, INRIA. 1995, pp.26. inria-00071200

HAL Id: inria-00071200
https://hal.inria.fr/inria-00071200

Submitted on 23 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.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PARA++ : C++ Bindings for Message Passing Libraries : User Guide

Olivier Coulaud , Eric Dillon

parapp@loria.fr

N° 0174

—Juin 1995

PROGRAMMES 1 et 6

ISSN 0249-6399

PARA++ : C++ Bindings for Message Passing Libraries : User Guide

Olivier Coulaud , Eric Dillon
parapp@loria.fr

Programmes 1 et 6 — Architectures parallèles, bases de données, réseaux et systèmes distribués — Calcul scientifique, modélisation et logiciel numérique
Projet RESEDAS , NUMATH

Rapport technique n0174 — |Juin 1995 — 26 pages

Abstract: The aim of Para++ is to provide a user-level C++ interface to message passing libraries, by encapsulating the notions of processes and inter-processes communications into specific C++ objects and streams. Actually, this abstraction level allows to implement Para++ with any kind of message passing library.

Para++'s main idea is to add new C++ io-streams to allows inter-tasks communications. These streams support all generic scalar datatype (`int`, `float`, `double`,...), plus some mathematical datatypes (`Vectors`, `Matrix`,...) in order to exchange data between co-operating tasks. Para++ has been implemented on top of PVM and MPI.

Key-words: Message passing, PVM, MPI, C++, SPMD

(Résumé : tsvp)

Unité de recherche INRIA Lorraine
Technôpole 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 C++ pour l'échange de messages : guide d'utilisation

Résumé : Le but de Para++ est de permettre l'utilisation de bibliothèques de communication par échange de messages à partir d'une interface utilisateur en C++. Par encapsulation des notions de processus et de communications entre processus dans des objets C++, Para++ permet de s'affranchir totalement de la bibliothèque d'échange de messages choisie. De cette manière, Para++ peut être utilisé avec toute bibliothèque de communication. L'idée principale de Para++ est d'ajouter des "flots" C++ pour remplacer les communications entre les tâches. Ces nouveaux flots C++ peuvent être utilisés avec les types de donnée classique (**`int`**, **`float`**, **`double`**,...), et avec quelques autres type pré-définis dans Para++ (**`Vectors`**, **`Matrix`**,...). Para++ a été implanté au dessus de PVM et de MPI.

Mots-clé : échange de messages, PVM, MPI, C++, SPMD

Contents

Introduction **4**

1 Installing Para++ **5**
1.1 Structure of the distribution directory 5
1.2 Installing Para++ 5
1.2.1 Making Para++ 6
1.2.2 Getting started with Para++ 6

2 The Para++ classes and functions **8**
2.1 The Para++ paradigm 8
2.1.1 The task handling with Para++ 8
2.1.2 The communication's handling with Para++ 9
2.2 The Para++ classes 11
2.2.1 The ParaProcess class 11
2.2.2 The ParaStreamOut class 12
2.2.3 The ParaStreamIn class 14
2.3 The Para++ functions and types 14
2.3.1 The Para++ functions 14
2.3.2 The Para++ types 15

3 Some examples **18**
3.1 Computing the value of π 18
3.2 Bitonic sort 20

Appendix **26**

Introduction

The biggest issue with Message Passing Parallelism is the number of message passing libraries available (P4, PVM, MPI, NX, EUI,...). Among them, two libraries are emerging: PVM (Parallel Virtual Machine) [1] which has become a de facto standard, and MPI (Message Passing Interface) [3] which tends to be the future standard. The use of these interfaces are somewhat different, that's why we propose Para++ based on an idea of Stream Message Passing (SMP) proposed by R. Pozo in [4].
The aim of Para++ is to provide C++ bindings to use a message passing library (currently PVM or MPI), without the user had to worry about the syntax of PVM or MPI. Para++0.9 is based on the SPMD parallel programming method. This library was defined and built for an domain decomposition method to solve the laplace operator with spectral methods.

This project has been developed in INRIA[1] and supported by the Charles Hermite Center[2].

[1] National Research Institute in Software Engineering and Automatism - FRANCE
[2] High performance computing and Modelization center in Lorraine FRANCE

Chapter 1

Installing Para++

1.1 Structure of the distribution directory

The Para++ distribution can be found at the following URL address:

`http://www.loria.fr/~coulaud/parapp.html`

The Para++ code distribution contains the following files and directories:

Config/ The makefiles to build the Para++ libraries. This directory contains the different makefiles to fit every architecture.

doc/ This file and a reference card to use Para++.

include/ The C++ include files for making and using the Para++ libraries. This directory must be included when using Para++.

src/ All the C++ sources to build the Para++ libraries.

examples/ Some basic examples and tutorials to start with Para++.

utils/ Some files to help the building of Para++.

1.2 Installing Para++

This section describes how to install the Para++ libraries, either for your own personal use or for the use of everyone at your site. (In the following sections, when we talk about "the Para++ libraries", we mean the Para++ libraries over PVM3.3.7 and over MPI (mpich over p4).

1.2.1 Making Para++

Before making the libraries, you must set some variables to indicate to Para++ the location of pvm3.x.x and of an implementation of MPI. The variables MPI_ROOT and PVM_ROOT must be set according to the local path installation of the corresponding libraries. The variables PVM_ARCH and MPI_ARCH must be set to take into account the architecture on which `Para++` will be compiled.

Then, to build Para++, position yourself in the top-level Para++ directory (here we refer to this directory as Para++, but you may also have it as Para++-1.1 or something similar) and type:

```
make all
```

The parameter `all` means that everything must be compiled: the MPI++ library, the PVM++ library and the set of examples. You can choose to compile some parts only, by setting the parameter to `lib` (compile the two libraries), `mpi` (or `pvm`) to compile only one of the libraries, `examples` to compile the examples.

The Para++ distribution includes a small script that tries to determine the target architure. This script returns one of the following types:

- sun4 : all sun4 workstations
- hp : all HP workstations.
- alpha : the DEC-alpha workstations.
- sgi64 : the silicon graphics workstations (PowerOnyx IRIX 6.0.1).

This answer is used to build the corresponding directory (in the Para++/lib/ directory).

At last, to save disk space, numerous objects files can be removed with

```
make clean
```

and the system can be restored to its original with

```
make realclean
```

(All *.o and *.a are removed).

Remark Para++ has been tested on hp and sun4 with g++ v2.6.3. and CC (ATT 3.x)

1.2.2 Getting started with Para++

The best way to begin with Para++ is to have a look at a basic example included in the distribution. (several examples are given in the 'Para++/examples' directory)

The example used here is `dummy.cc`. Its aim is to send an integer value from the master process to a slave process.

The source code is given below.

```
#include "Para++.hh"
```

```
main(int argc, char **argv)
{
    int n;
    int ntask;
    ParaProcess p("dummy", argc, argv);
//
    p.init(ntask);              // With MPI, the number of tasks
                                // is chosen with the 'mpirun' command.
                                // ntask is set to the right number.
    if (p.master())
        {
//  the master task sends a dummy n to the slave number 1
         n = 1654;
         pout(p.tasks(1)) << n; pout.flush();
        }
    else
        {
// the slave receives a dummy n from the master task
         pin(p.tasks(0)) >> n;
         cout << n << endl;
        }
    p.end();
}
```

To make it, type `make PARARCH=`../utils/parach` examples` in the `Para++/examples` directory.

Then, as we choose to use MPI as underlying library, you must start the executable with the corresponding MPI launching command, that might be something like that:

`mpirun -np 2 dummy`

(You might have to do things before depending on the MPI implementation you use, refer to the documentation of it.)

To use it with PVM, change Makefile in the `examples/` directory, start the virtual machine and type dummy. (Remember that the `dummy` file must be accessible form PVM, in the ${HOME}/pvm3/bin directory for example see [2] for more details.)

Chapter 2

The Para++ classes and functions

This chapter presents the interface used to handle any Message Passing library. This version currently supports PVM and MPI only, but should be extended to other Message Passing libraries.

2.1 The Para++ paradigm

The aim of Para++ is to encapsulate all the message passing primitives in a common interface, that make it easier to use.

To do so, we define two classes named `ParaProcess` and `ParaStream` that handle respectively the tasks and the messages of an application. These classes are detailed below.

2.1.1 The task handling with Para++

An application based on message passing implies that several tasks are running concurrently on different sites. According to the underlying library used (currently PVM or MPI), the tasks have to be replicated on the different processors, each task receives an identifier, and the set of these identifiers will represent the whole application.

Para++ helps the user to launch and to handle the tasks with the class `ParaProcess`. By instantiating a `ParaProcess` object with the appropriate parameters, he will be able to access every identifier of tasks, to replicate the number of tasks he needs, and so on.
The following tiny program shows the basic way to use the class `ParaProcess` :

```
#include "Para++.hh"
```

```
main(int argc, char ** argv)
{
    int ntask;
    ParaProcess p("tiny", argc, argv); // We define the name of the executables
//
    ntask = 2;                         // There will be 2 tasks
//
    p.init(ntask);                     // we launch the second task

    if (p.master())
        cout << "I'm the master" << endl;
    else
        cout << "I'm the slave" << endl;
    p.end();
}
```

This example starts by specifying the underlying device used (here PVM) by using the `#define` primitive. Then the application instantiates a `ParaProcess` object. The parameter passed to the constructor means that the executable's filename is `tiny` (this string is used for replication of processes).

The first method called of the "`p`" object is "`init(2)`", which means that the code must be replicated once so that the application consists of two concurrent tasks. The application then goes on executing the same code.

As you may have noticed it, the programming model of Para++ is currently forced to SPMD[1], the following release of Para++ may have an extension to MPMD[2].

A `ParaProcess` object contains information about the whole application, including the number of concurrent tasks, the list of identifiers (which is the `tasks` vector), and provides a way to synchronise all the tasks by a `barrier` method. All those informations are shared by all the tasks. The reader may refer to the next section for more informations.

2.1.2 The communication's handling with Para++

After having defined the concurrent tasks, Para++ provide a way to let them exchange messages. To do so, Para++ defines two classes inheriting `ParaStream` named `ParaStreamOut` and `ParaStreamIn`, and instantiates two objects `pin` and `pout`. Both those objects can be used like `cout` and `cin` provided by C++.
Here's a little example that shows how to use these two objects:

```
#include "Para++.hh"
```

[1]stands for Single Program Multiple Data
[2]stands for Multiple Programs Multiple Data

```
main(int argc, char **argv)
{
    int n;
    int ntask;
    ParaProcess p("dummy", argc, argv);
    ntask =2;
    p.init(ntask);              // To launch 1 more task

    if (p.master())
        {
// the master task sends a dummy n to the slave number 1
            n = 1654;
            pout(p.tasks(1)) << n; pout.flush();
        }
    else
        {
// the slave receives a dummy n from the master task
            pin(p.tasks(0)) >> n;
            cout << n << endl;
        }
    p.end();
}
```

First of all, we instantiate the `ParaProcess` variable with the name of the executable. We call the `init` method on this object to replicate the code on a child process.

Then we use the `pout` and the `pin` objects like the `cout` and `cin` provided by C++. On the master side, the

`pout(p.tasks(1)) << n; pout.flush();`

sends the value of `n` to the task 1 referenced by its Tid (Task Identifier).
Note that the master task is always named as the task number `0`.
On the slave side, the

`pin(p.tasks(0)) >> n;`

means that the value of n will be received from the task `0`.
In this release, the user cannot choose whether he wants blocking or non blocking `pout` and `pin`. They are forced to 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 done. The reader may refer to the next section for more informations.

2.2 The Para++ classes

The Para++ distribution is build on 3 main classes:

ParaProcess which handles everything about the tasks.

ParaStreamOut which handles everything about sending operations.

ParaStreamIn which handles everything about receiving operations.

The sections below describe those classes. To use all the Para++ classes, the user must first specify the underlying library he uses, by defining one of symbols

```
_PVM_DEV_
_MPI_DEV_
```

This can be done by adding the option: `-D_PVM_DEV_` or `-D_MPI_DEV_` on the "CC" line. Then the program must include the "Para++.hh" file. By default, if no underlying library is defined, the compilation will be aborted.

2.2.1 The ParaProcess class

The aim of this class is to handle all about the tasks. It is responsible for the initialisation of the underlying library (PVM or MPI), the replication of tasks (with PVM), and gives some useful informations and methods on all tasks.

Instantiation of ParaProcess objects

A `ParaProcess` object is designed to handle everything about the cooperating tasks. In consequence, one and only one object of `ParaProcess` class must be instantiated within the source code. This instantiation must be done at the very beginning of the "main" function.

Several parameters must be passed at the instantiation, whatever the underlying library used (PVM or MPI).

The user must pass the name of the executable file, and the `argc` and `argv` arguments. The name of the executable file will only be used within PVM to spawn all the tasks. Both `argc` and `argv` will only be used to fit MPI initialisation.

For example, if the program is called "dummy" the instantiation within Para++ could be:

```
#include "Para++.hh"

main(int argc, char **argv)
{
   ParaProcess p("dummy", argc, argv);
   ...
}
```

Warning: be sure that the referenced executable is accessible to pvm3 (for example, in the ${HOME}/pvm3/bin directory). Refer to PVM User Guide [2] for more informations.

The methods and instances of ParaProcess objects

The `ParaProcess` class contains several methods to handle the tasks.

- `int init(int& n)`: initiates the launching of n-1 slaves tasks. All tasks must call this method. It synchronises all tasks before returning to the caller.
- `int barrier()`: blocks until all tasks have synchronised. All tasks must call this methods to synchronise.
- `int end()`: finalise the Para++ processes. This methods must be called by all the tasks before exiting.
- `int master()`: returns "`PARA_SUCCESS`" if the calling tasks is the master task.

Examples of all these methods can be found in the `Para++/examples/` directory.

A `ParaProcess` object also provides some useful methods like:

- `int me()`: contains the rank number of the calling task.
- `Vector<int> tasks()`: contains the task identifier of each cooperating task. It is the tid number with PVM, and the rank in MPI_COMM_WORLD with MPI.
- `int tasks(int n)`: contains the identifier of task n. task(0) is always the identifier of the master process.
- `int TaskNumber()`: contains the number of cooperating tasks.

All those informations are available locally on all task, after the `init` method is called.

2.2.2 The ParaStreamOut class

The `ParaStreamOut` class is provided to handle every outgoing communication. The user do not have to instantiate a `ParaStreamOut` object, since Para++ does it. This object is named `pout` (the name has been chosen to remind the `cout` stream operator of C++).

Each time the user may want to send the value of a variable to another task, he may use `pout`. But for every outgoing message, one or more receivers must be specified. The user can specify it in two ways. First, he can overload the `pout` object with some parameters, as describe below, and simply use the `pout.flush()` command.

- `pout() << n`: the value of n will be broadcast to all the tasks.
- `pout(int id) << n`: the value of n will be send to the task referenced by its task identifier `id`.

- pout(int id, int tag) << n: the value of n will be send to the task referenced by its task identifier id. The message is tagged with the value tag. This tag will be used at the receiver side to match this very call.

Second, he can choose to build his message first without specifying the receiver by successive "<<" operations and flush the message afterwards to the receiver with the flush(...) method. Suppose that a message has been built with a command like

```
...
pout << n << i << "dummy" << V;
...
```

The user can "flush" the buffer with the following parameters:

- pout.flush() : broadcasts the message to all tasks.
- pout.flush(int id) : sends the message to the task referenced by its task identifier id.
- pout.flush(int id, int tag) : sends the message to the task referenced by its task identifier id. The message is tagged with the value tag. This tag will be used at the receiver side to match this very call.
- pout.flush(Vector<int> V) : sends the message to all the task referenced in an entry of the vector V.
- pout.flush(Vector<int> V, int tag) : sends the message to all the task referenced in an entry of the vector V. The message is tagged with the value tag. This tag will be used at the receiver side to match this very call.

The pout.flush(...) method represents an asynchronous send: it will return to the caller as soon as it is sure that the message will be delivered (i.e. it does not wait for the receiver to actually get it with a matching pin call.).

In Para++, pout (and of course pin) can handle the following types: int, short, long, float, double, char strings, Vector<T>, matrix<T>, where T can be one of the scalar types (int, float, double).

Several examples can be found in the Para++/examples directory.

For the PVM interface, the XDR encoding is used by default but you can change it as follows

```
pout.data_code() = PvmDataRaw;
```

Remark When we use the pout() operator we destroy all the value in the ParaBuffer without sending them.

2.2.3 The ParaStreamIn class

The `ParaStreamIn` class is provided to handle every incoming communication. The user does not have to instantiate a `ParaStreamIn` object, since Para++ does it. This object is named `pin` (the name has been chosen to remind the `cin` stream operator of C++).

Each time the user may want to receive the value of a variable from another task, he may use `pin`. But for every incoming message, the sender must be specified. To do so, the user can overload the `pin` object in two ways:

- `pin() >> n;` : receives the value of `n` from any task.
- `pin(int id) >> n;` : receives the value of `n` from the task referenced by its task identifier `id`. `pin(ANY_TASK)` has the same effect as `pin()`.
- `pin(int id, int tag) >> n;` : receives the value of `n` from the task referenced by its task identifier `id` with a message tagged with `tag`. This receive operation matches only a message tagged with this `tag`. With `pin(ANY_TASK, tag)` you can receive from any task, waiting for a particular tag.

The values are received in the same order as they were sent.
Whenever, the receive object must by instanciated (a `Vector` or a `Matrix` for example), the `pin` operation does it.

The `pin` object represents a "blocking" receive: `pin` will block while waiting for a matching message (i.e. from the right sender and, with the right tag).

2.3 The Para++ functions and types

The Para++ distribution provides a set of functions and types that can be used within the scope of Para++. These functions consist essentialy of global operations between tasks. The types defined within Para++ consist of Vector, Distributed Vector and Matrix, and are designed for high performance computing.

2.3.1 The Para++ functions

This release of Para++ provides only two global operations called `value_reduction` and `sync()`.
The aim of the former is to perform an operation on a value distributed localy on all task, and to make the result available to each task. The aim of the latter is to synchronize all tasks on this function.

Synopsis:

```
int value_reduction(ParaOp operation, T& value)
```

Parameters:

operation - Function which defines the operation performed on the global data. See below for predefined operations. Users may be allowed to define their own functions in the next releases of Para++.

value - The operand of the global operation passed by reference. The type `T` can be choosen among {int, float, double}.

All the tasks call the `value_reduction` function with the local variable used by the operation. After the call, all tasks have the result on their own local variable.
The Para++ library contains the following predefined global functions, that can be used as the parameter `operation` above:

- `ParaMin`: computes the minimum value.
- `ParaMax`: computes the maximum value.
- `ParaSum`: computes the sum of all values.
- `ParaProd`: computes the product of all values.

Example:

```
error = value_reduction(ParaSum, n);
```

Errors:
The following error conditions can be returned by `value_reduction` :

- `ParaBadFunc`: unknown function used.
- `ParaBadParam`: giving an invalid argument value or type.

Synopsis:

```
int sync()
```

Parameters: None.

A call to `sync()` blocks the tasks until all tasks have called the function.

2.3.2 The Para++ types

The Para++ distribution provides several useful C++ classes in the scope of high performance computing. These classes are the following:

- `Vector` allow the user to handle vectors of int, float, or double and to send them via the `pout` and `pin` objects.
- `Matrix` allow the user to handle matrix of int, float, or double and to send them via the `pout` and `pin` objects.

The Vector class

The `Vector` class makes it possible to use vectors within the Para++ application. The main features of `Vectors` are following:

- The initialisation of the vector with a specific value.
- The access and modification of each entry of the vector.
- The common operations between a vector and another vector, and between a vector and a scalar.
- Some useful functions like min, max, norm_inf, norm2.

The constructors of `Vector` are following:

- `Vector<T>(int n)`: Used to instanciate a vector of n entries of type T.
- `Vector<T>(int n, T value)`: Used to instanciate a vector of n entries, and set all the entries to the initial value `value`.
- `Vector<T>(Vector <T>& V)`: It is used to duplicate the vector V.

The methods of `Vector` are following:

- `int numElts()`: Returns the number of entries in the vector.
- `T* Elts()`: Returns a pointer to the array where are stored the values of the vector.
- `T* Elts(int n)` Returns a pointer to the entry number n of the vector.
- `T max()`: Returns the max value of the vector's entries.
- `T min()`: Returns the min value of the vector's entries.
- `T norm_inf()`: Returns the infinite norm of the vector's entries.
- `T norm_2()`: Returns the euclidian norm of the vector's entries.

The classical arithmetic operations on vector-vector or scalar-vector like +,-,+=,-=,* (scalar product) could be used.

An operator has been defined to access an entry of a `Vector`: so if you want to reference the i-th entry of V, you can write:

```
V(i) = 23657;
```

Note that the list of identifiers of tasks defined within the `ParaProcess` class, is a Vector<int>.

The DistVector class

The **DistVector** class makes it possible to use global operations on vectors which are on different processes. In fact this class is a specification of the Vector class where we overlap the global operations like scalar product, min, max , ...

The Matrix class

The Matrix class makes it possible to use the matrix within a Para++ application. The encode of the **Matrix** are done to use them with FORTRAN program. In fact we consider that a matrix is store column by column.

The constructors of Matrix are following:

- `Matrix<T>(int n_rows,int n_cols)` : to instanciate a matrix of `n_rows` rows and `n_cols` columns of type T.
- `Matrix<T>(int n_rows,int n_cols,T val)` : to instanciate a matrix of `n_rows` rows and `n_cols` columns of type T, and set all the values to `val`.
- `Matrix<T>(const Matrix<T>& m)` : to duplicate matrix `m`.

The methods of Matrix are following:

- `int numRows()` : returns the number of rows of the Matrix.
- `int numCols()` : returns the number of columns of the Matrix.
- `T* Elts()`: Returns a pointer to the array where are stored the values of the matrix.
- `T max()`: Returns the max value of the matrix's entries.
- `T min()`: Returns the min value of the matrix's entries.
- `T norm_inf()`: Returns the infinite norm of the matrix's entries.
- `T norm_2()`: Returns the euclidian norm of the matrix's entries.

The classical arithmetic operations on matrix could be used like +,-,+=,-=. Moreover we can access to an element $a_{i,j}$ of the matrix A by `A(i,j)` or `A(i+j*A.numRows())`.

Chapter 3

Some examples

3.1 Computing the value of π

This first exemple presents the computing of π. Since we know that

$$\pi = \int_0^1 \frac{4}{1+x^2} = \frac{1}{N}\sum_{i=0}^{N} \frac{4}{1+(\frac{i}{N})^2}$$

we will integrate the function

$$f(x) = \frac{4}{1+x^2}$$

. This is done by the following program.

```
//-----------------------------------------------------------------
//                        __
//  Aim : Computing the || value
//
//                  / 1
//                  |       4          __
//                  |   ----------  = ||
//                  |    1 + x**2
//                  / 0
//
//        par la méthode des trapèzes.
//
//  Authors  : Olivier Coulaud, Eric Dillon
//  Version : 0.9 de Para++.
//-------------------------------------------------------------------
```

```
#include "Para++.hh"
#include <iostream.h>
#include <math.h>
//
// the function to integrate
//
inline double f( double x)
{
    return(double(4.0) / (double(1.0) + x*x));
}

main(int argc, char **argv)
{
    int i;
    double pi= double(0.0), h, x;
    int nprocs,n;
    double Pi=3.1415926535897932385;
    int MSGDATA=667;
//
    ParaProcess p("pi", argc, argv);
//
    if(p.master())
        {
//
         cout << "Enter the number of steps, n = " ;
         cin >> n;
         cout << "Enter the number of tasks <= n " ;
         cin >> nprocs;
         if ( n == 0 || nprocs == 0 )
             {
                p.end();
                exit(2);
             }
        }
    if (p.init(nprocs) != SUCCESS)
     cout << "init errorr" << endl;
    if (p.master())
        {
         pout() << n; pout.flush(p.tasks(), MSGDATA);
        }
    else
        {
```

```
         pin(-1, MSGDATA) >> n;
        }
    int nombre = int(n/nprocs);
    h = double(1.0) /double(n);
    int me = p.me(),taille_max =  me == nprocs-1 ? n : (me+1)*nombre;
    x =  h * double(me*nombre);
    for (i = me*nombre; i < taille_max; i++)
        {
         pi += (f(x)+f(x+=h));
        }
    pi *= h * double(0.5);
    cout << "Local value of pi " << pi << endl;
    value_reduction(ParaSum,pi);
    if (p.master())
        {
         cout << " pi          = " << Pi << endl
              << " pi computed = " << pi << endl
              << " Abs error   = " << pi - Pi<< endl;
        }
    p.end();
}
```

We have choosen to use PVM as underlying library, but it can be easily be changed to MPI: look the `pi.cc` file in `Para++/examples`.
The master task choose the range of the integration and number of tasks to be used. It sends the range to every tasks.Each task computes a local value of π and the result is given by the `value_reduction` call. The result is only printed by the master task.

3.2 Bitonic sort

This example implements the bitonic sort algorithm with Para++. Each entry of the array to sort is associated with a task. all tasks exchange their values till the array is sorted.

```
//
//  Bitonic Sort with Para++
//
//  $Date: 1995/06/15 16:43:03 $
//
//  The program has not been written to be efficient but only to be
//  an example of Para++ use.
//
//  O. Coulaud, E. Dillon
```

```
//////////////////////////////

#include "Para++.hh"

#define ARRAY_SIZE 4
#define SWAPTAG 2

int inverse(int rank, int j)
{
    int n_rank, jeme;
    // inverse j-th bit of integer "rank"
    jeme = rank & (1<<j);

    if (jeme == 0)
        n_rank = rank + (1<<j);
    else
        n_rank = rank - (1<<j);

    return(n_rank);
}

int min(int n1, int n2)
{
    if ( n1 <= n2 )
        return(n1);
    else
        return(n2);
}

int max(int n1, int n2)
{
    if ( n1 >= n2)
        return(n1);
    else
        return(n2);
}


int bit(int rank, int j)
{
    return ((rank & (1<<j)) != 0);
}
```

```
main(int argc , char** argv)
{

    int my_tid, i, j, part, numt, info, tmp, iproc;
    int k, value;
    int tids[16];
    int one_tid, atid, atag, alen;
    char *argu[1];
    char temp;
    int part_tid;
    int nbtask;
    // Para++ initialisation
    ParaProcess p("bitonic", argc, argv);

    // each entry of the array to sort is a task
    // here is the entry of this particular task.
    // each task has a partner defined by the bitonic algorithm
    // here is the entry of the partner.
    int my_entry, part_entry;

    // The array to be sorted
    static int tab[ARRAY_SIZE] = { 5, 8, 15, 1};

    // ParaProcess launching
    nbtask = ARRAY_SIZE;
    cout << "Initializing ..." << endl;
    p.init(nbtask);

    cout << " Now sorting ..." << endl;
    iproc = p.me();

    // Am I the master ?
    if (p.master())
        {
            // I do not have to send to values of the array to each task
            // since they already know them.
            my_entry = tab[iproc];
            for (i=1; i<=2; i++)
                for (j=i-1; j>=0; j--)
                    {
```

```
                    // find my partner, by changing j-th bit of rank
                    // number.
                    part = inverse(iproc, j);

                    // send my entry to my partner
                    pout(p.tasks(part), SWAPTAG) << my_entry;
                    pout.flush();
                    // ... and get his
                    pin(p.tasks(part), SWAPTAG) >> part_entry;

                    if (bit(iproc, j) == bit(iproc,i))
                        my_entry = min(my_entry, part_entry);
                    else
                        my_entry = max(my_entry, part_entry);
                }

        // The master gathers all the entries.
        tab[0] = my_entry;
        for(i=0; i<ARRAY_SIZE-1; i++)
            {
                pin() >> k >> value;
                tab[k] = value;
            }

        cout << "Resulting array :" << endl;
        for (i=0; i<ARRAY_SIZE; i++)
            cout << tab[i] << " " ;
        cout << endl;

    }
else
    {
        my_entry = tab[iproc];
        for (i=1; i<=2; i++)
            for (j=i-1; j>=0; j--)
                {
                    // finding out the partner's rank
                    part = inverse(iproc, j);

                    // send my entry to the partner....
                    pout(p.tasks(part), SWAPTAG) << my_entry;
                    pout.flush();
```

```
                        // ... and get his
                        pin(p.tasks(part), SWAPTAG) >> part_entry;

                        if (bit(iproc, j) == bit(iproc,i))
                            my_entry = min(my_entry, part_entry);
                        else
                            my_entry = max(my_entry, part_entry);
                    }

            // Each task sends its value to the master
            pout(p.tasks(0)) << iproc << my_entry;
            pout.flush();
        }

    p.end();
}
```

Bibliography

[1] PVM : Parallel Virtual Machine : a users' guide and tutorial for networked parallel computing. Al. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. Mit Press, 1994.

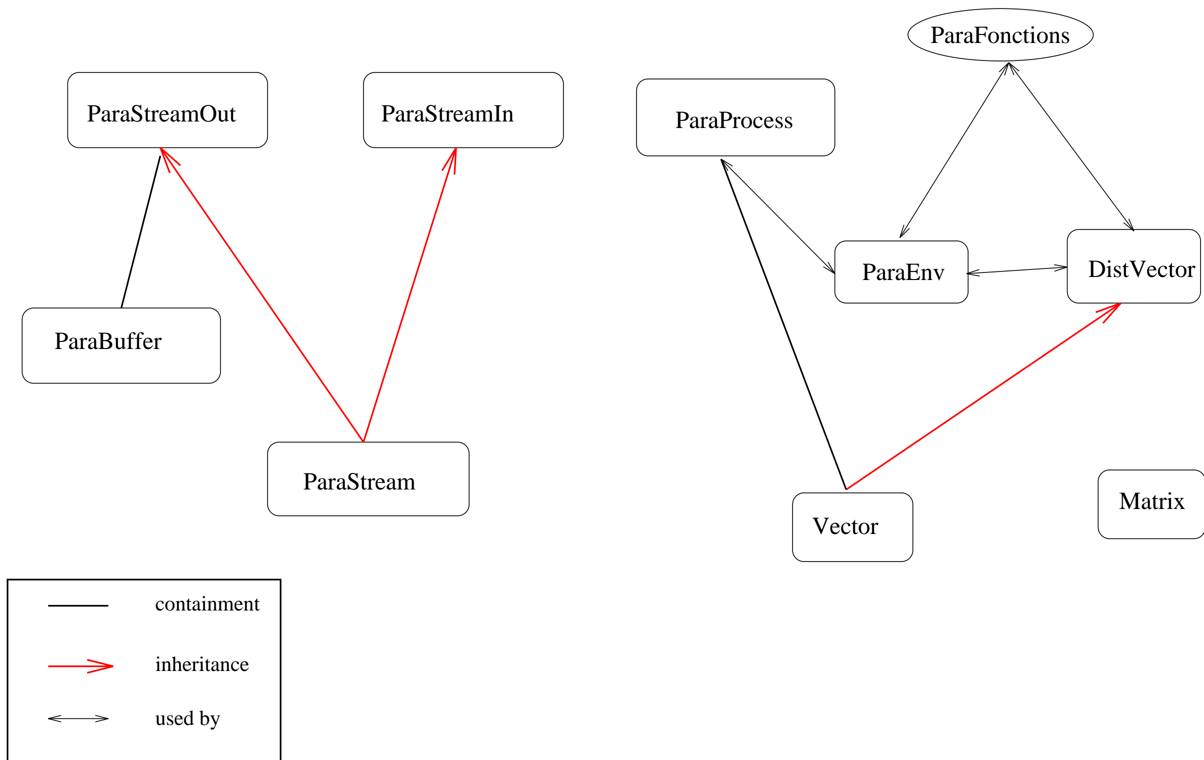
[2] PVM3 User's guide and reference manual. Al. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam.

[3] Using MPI : portable parallel programming with the message- passing interface. W. Gropp, E. Lusk, A. Skjellum. Mit Press, 1994.

[4] "A Streams-Based Interface in C++ for Programming Heterogeneous Systems", R. Pozo, CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing, September 7-8, 1992, Saint Hilaire du Touvet, France.

Appendix

Overview of Para++ Class Hierarchy



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, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
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