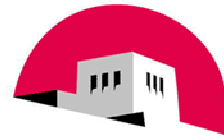




Ecole d'ingénieurs et d'architectes de Fribourg  
Hochschule für Technik und Architektur Freiburg



The University of New Mexico

# Improving POP-C++ for HPC

## Diplomawork 2007

Schrag Manuel

09/06/2007 – 11/09/2007



**Responsible interns :** Pierre Kuonen  
François Kilchoer  
Jean-François Roche  
Guilherme Peretti Pezzi

**Responsible externs :** Barney Maccabe (UNM)  
Rolf Riesen (Sandia Labs)  
Tuan Anh Nguyen (HCMUT)

**Expert :** Peter Kropf (UNI-NE)

**Student :** Manuel Schrag

**Date :** 09/06/2007 to 11/14/2007



## TABLE OF CONTENTS

<b>GENERAL INTRODUCTION.....</b>	<b>5</b>
1. TECHNOLOGIES USED.....	5
1.1. Ubuntu.....	5
1.2. POP-C++.....	5
1.3. MPI.....	5
1.4. Computer cluster / GRID.....	5
2. OBJECTIVES.....	6
3. PRESENTATION OF THE DOCUMENT.....	6
<b>COMPARISON BETWEEN POP-C++ AND MPI.....</b>	<b>7</b>
1. INTRODUCTION {COMMON}.....	7
2. ANALYSIS.....	7
2.1. POP-C++ Runtime {common}.....	7
2.2. POP-C++ remote method call.....	8
2.2.1. Invocation semantics.....	9
2.3. The MPI standard {common}.....	10
2.3.1. MPI-processes.....	10
2.3.2. SPMD.....	11
2.4. MPI message passing (point-to-point).....	11
2.4.1. Buffering.....	11
2.4.2. Blocking point-to-point communication.....	12
2.4.3. Non-blocking point-to-point communication.....	12
2.5. The "Phoenix" cluster.....	12
2.6. Benchmarking data sending.....	13
3. DESIGN.....	14
3.1. Cluster configuration.....	14
3.1.1. POP-C++.....	14
3.1.2. OpenMPI.....	14
3.2. Benchmarking latency and data sending.....	14
3.2.1. MPI test scenario.....	15
3.2.2. POP-C++ test scenario.....	17
4. IMPLEMENTATION.....	18
4.1. MPI.....	18
4.1.1. Asynchronous latency and data sending.....	18
4.1.2. Synchronous latency and data sending.....	19
4.2. POP-C++.....	20
4.2.1. Asynchronous latency and data sending.....	20
4.2.2. Synchronous latency and data sending.....	21
5. RESULTS.....	23
5.1. Prediction.....	23
5.1.1. OpenMPI asynchronous.....	24
5.1.2. OpenMPI synchronous.....	25
5.1.3. POP-C++ asynchronous.....	26
5.1.4. POP-C++ synchronous.....	27



5.1.5.	Conclusion .....	28
5.2.	Obtained results .....	29
5.2.1.	Latency.....	29
5.2.2.	Data sending.....	30
5.2.3.	Measurement vs. prediction .....	34
5.2.4.	Including unconcerned processes/objects .....	36
6.	CONCLUSION .....	36
6.1.	Encountered problems .....	36
6.1.1.	Starting the "SXXparoc" service on computing nodes .....	36
6.1.2.	ssh: Connection refused .....	37
6.1.3.	Cluster access and disturbed measurements.....	37
6.2.	POP-C++ vs. MPI .....	37
<b>COLLECTIVE COMMUNICATION IN POP-C++ .....</b>		<b>39</b>
1.	INTRODUCTION {COMMON}.....	39
2.	ANALYSIS .....	39
2.1.	Collective communication in MPI.....	39
2.2.	Collective communication in POP-C++ .....	41
2.3.	The POP-C++ parser .....	43
3.	DESIGN .....	46
3.1.	Group representation.....	46
3.2.	Collective communication operations.....	47
3.3.	Architecture .....	49
3.4.	Creating a "group parser" .....	52
3.5.	Handling a group of remote objects for N-N communication.....	55
4.	IMPLEMENTATION.....	55
4.1.	Code models .....	55
4.2.	Group parser.....	55
4.2.1.	Including the group parser in the POP-C++ compilation process.....	58
4.3.	Where to find the source code for the group parser? .....	59
4.4.	Limitations and recommendations.....	59
4.4.1.	Installing the distribution .....	59
4.4.2.	Split application in logical parts.....	59
4.4.3.	Remote objects as parameters.....	59
5.	TESTS .....	61
6.	FUTURE WORK.....	61
6.1.	Parse C++ class declarations .....	61
6.2.	Clean source code.....	61
6.3.	Sophisticated algorithms for group method invocations .....	62
7.	CONCLUSION .....	63
<b>GENERAL CONCLUSION .....</b>		<b>64</b>
1.	PERSONAL CONCLUSION.....	64
2.	THANKS.....	64



---

APPENDIX.....	65
1. UNREALIZED DESIGNS .....	65
1.1. Syntax for collective communication .....	65
1.1.1. Create a stub for every collective communication operation.....	65
1.1.2. Differentiate incoming and outgoing methods .....	65
2. CD STRUCTURE .....	66
3. SOURCE CODE.....	67
3.1. Code model for the base POPGroup template.....	67
3.2. Code model for specialized POPGroup templates .....	67
3.3. Exception handling in collective communication .....	69
4. DEFINITIONS.....	69
5. REFERENCES.....	71
6. FIGURES .....	72
7. TABLES.....	73



---

# General introduction

---

In this project, four different tasks figure in our goal specification. The work has to be split in two distinct projects between the two students Manuel Schrag and Frédéric Barras. The goal is to improve the language POP-C++ for **H**igh **P**erformance **C**omputing (HPC). For more details, please refer to section 2 *Objectives*. This project is realized at the *University Of New Mexico (UNM)* in Albuquerque, USA.

## 1. Technologies used

This is a brief description of the different technologies used during the project.

### 1.1.Ubuntu

*Ubuntu* is a distribution of the free operating system Linux and is based on *Debian Linux*. We use the version 6.10, called *Edgy Eft*, on our notebooks. At this time (September 2007), the most recent version is 7.04. This version is actually not compatible with our notebooks. After research on the internet, it seems that it is a common problem that *Ubuntu 7.04* has difficulties with hardware detection.

### 1.2.POP-C++

POP-C++ is a programming language which is derived from C++. It has been developed by the GridGroup at the EIA-FR. Its main objective is to allow programmers to write object oriented applications which are able to run objects on different workstations connected by a network. These objects are distributed automatically during the execution of an application. At the time of this project, POP-C++ is running on workstations with a Unix/Linux operating system.

The POP-C++ environment includes two main parts. First of all, a precompiler which is able to parse and read a POP-C++ code to generate pure C++ code. This code is now ready to be compiled with a standard C++ compiler. Secondly, the runtime: an environment which is necessary to run the parallel distributed objects.

### 1.3.MPI

MPI (Message passing interface) is a standard, describing message passing in parallel computing on a distributed computer system. The programming interface specifies a collection of operations and their semantics.

The implementation of MPI used during this project is *OpentMPI v1.2.3* which implements the MPI-2 standard. It is delivered with C/C++ and Fortrand 77 resp. Fortrand 90 compilers. The programmer can choose to write programs in one of these languages. Throughout this document when talking about MPI, we always make reference to this standard and this implementation.

### 1.4.Computer cluster / GRID

A computer cluster is a group of connected computers working together. They are commonly connected to each other by a fast local area network, allowing them to share tasks though they can be viewed as a single computer. Clusters are used for different tasks (e.g. Load-balancing). We will use a cluster for high performance computing. Programs which are designed to run on such a cluster (e.g. MPI-programs) split



the main problem in different smaller tasks to distribute them on different nodes on the cluster. These tasks are computed in parallel which increases the performance.

The main differences between a cluster and a GRID:

- GRID computing is dedicated to work on computers that can be geographically separated
- Computers on a GRID are quite autonomous and are not dedicated to accomplish only subtasks on this network

## 2. Objectives

This project will help improve the POP-C++ programming language in high performance computing. The first step will be the comparison with the MPI programming interface. This will help us to find eventual weak points of POP-C++. The second step is to add global communication to POP-C++. Global communication makes it possible for remote objects to communicate by broadcasting (point-to-multipoint message passing). The next step will be the development of a convenient method to create arrays of parallel objects using different personalized constructors. Until now, only the default constructor is used. The last step in the project is the validation of the asynchronous object creation in the most recent version of POP-C++.

In our planning, we must include the details about the two first parts. If we manage to finish them before the end of the project, the rest of the time is used for the two last tasks.

## 3. Presentation of the document

This document is subdivided in five main sections:

- a general introduction: Gives an overview of the project. Describes the objective(s) of the project and the technologies used.
- the first part of the project which is the comparison between MPI and POP-C++: This section contains a general analysis of MPI and POP-C++ followed by the design and implementation of the test scenarios used to accomplish the comparison. A subsection called *Results* presents the results of these tests.
- the second part of the project which is adding collective communication to POP-C++: It contains an analysis of how collective communication works in MPI and how it can be embedded in an object oriented programming language. This analysis is followed by the design and implementation of a POP-C++ distribution including collective communication.
- a general conclusion: This section contains my personal conclusion of the project and my thanks to all the persons supporting me during the work.
- an appendix: Contains references, definitions, extracts of source code, the structure of the attached CD and an index of all figures and tables of the document. Another subsection describes unrealized designs which were discussed during the project.

References are noted between brackets ([ ]).

Common parts can are noted {common} in the title of the corresponding section.

---

# Comparison between POP-C++ and MPI

---

## 1. Introduction {common}

In this part of the project, we have to compare POP-C++ to another distributed programming interface called MPI, implemented in OpenMPI (other implementations exist: e.g. MPICH). We have to define a test scenario to measure the differences between POP-C++ and MPI in terms of performance. To do this, we must implement equal test programs in both languages and run them on a cluster under identical conditions. The result will permit us to determine which language is slower in which part of the program and where their weaknesses are. This kind of comparison is called **Benchmarking**.

## 2. Analysis

This chapter is an analysis of the technologies used (POP-C++ and MPI) for the benchmarks on a cluster and issues related to benchmarking on multiple nodes.

### 2.1.POP-C++ Runtime {common}

This is only a short description of the POP-C++ runtime. A complete description is available in, "WSDL with POP-C++" [3].

The POP-C++ Runtime contains the following classes:

- *Interface* : the local representation of a remote object
- *Broker* : makes the translation between network messages and method call on the remote object
- *Combox* : contains the socket for the network communication
- *JobManager* : manages the resources and places objects on remote nodes
- *Buffer* : packs/unpacks the data which have to be sent/received
- *Objects* : are the real remote object, always bound with a buffer, and an interface on the local side.

Every remote object has an interface on the local runtime, which has the same methods with the same signatures. Seen from the local runtime, calling a method on a remote object or on a local object makes no difference. When a method call is received by the interface, it is forwarded through the network to the remote object by using the combox and the buffer. The broker attached to the remote object receives the packet and translates it into a method call for the object in order to send back the result, if necessary.

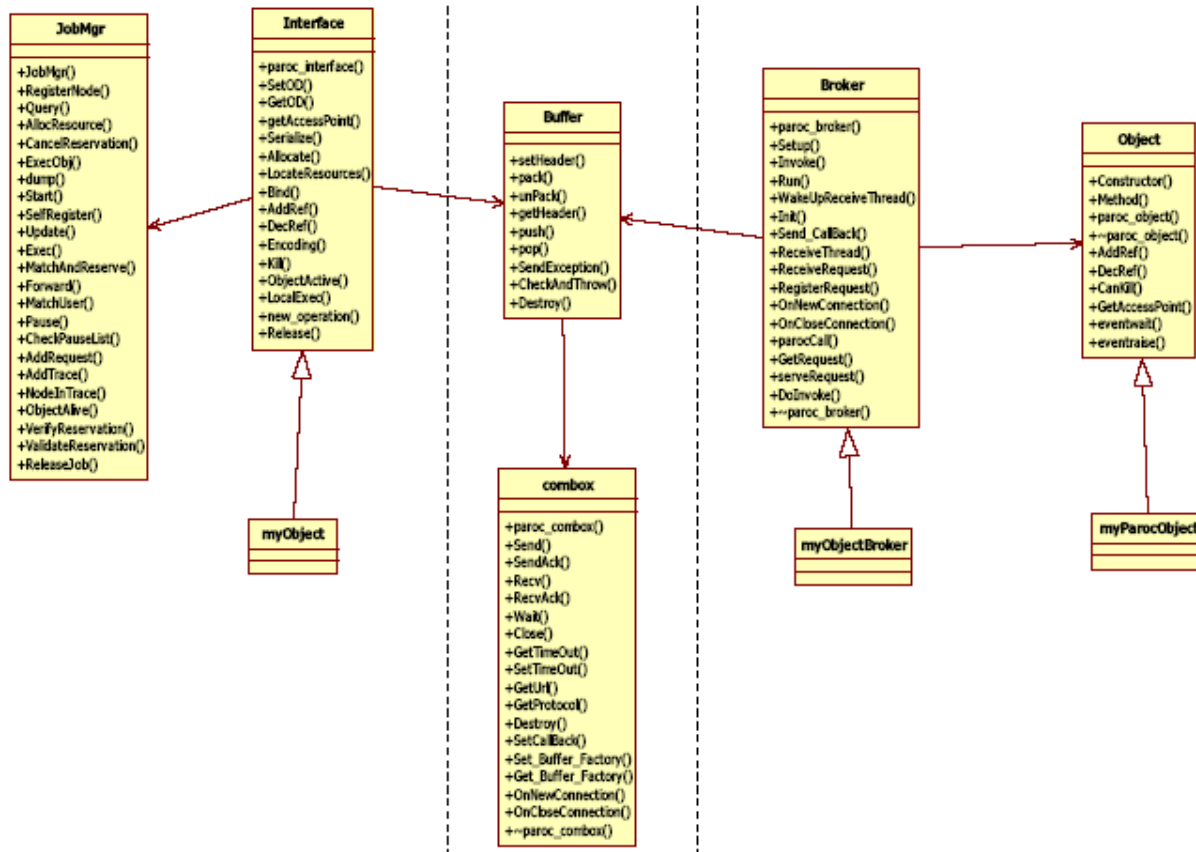


Figure 1 The POP-C++ class diagram

This diagram shows the different classes of the POP-C++ runtime (Figure 1), on the left are the classes used exclusively on the local part, on the right the classes used by the remote part and in the middle, the classes used by both local and remote parts. *MyObjectBroker* and *MyParocObject* are the remote part of our object. *MyObjectBroker* receives and serves requests and calls the corresponding methods on *MyParocObject*.

## 2.2.POP-C++ remote method call

The diagram below shows the simplified call of a method on a remote object (Figure 2). The *Interface*, which represents the remote object on the local machine, sends data via the *combox* through the network. This data includes all the information needed to invoke the remote method such as:

- Which method to invoke
- What parameters (type, size)

On the remote machine, the corresponding *combox* receives the data, which will be processed to filter out the information mentioned before to finally invoke the correct method with the passed parameters.



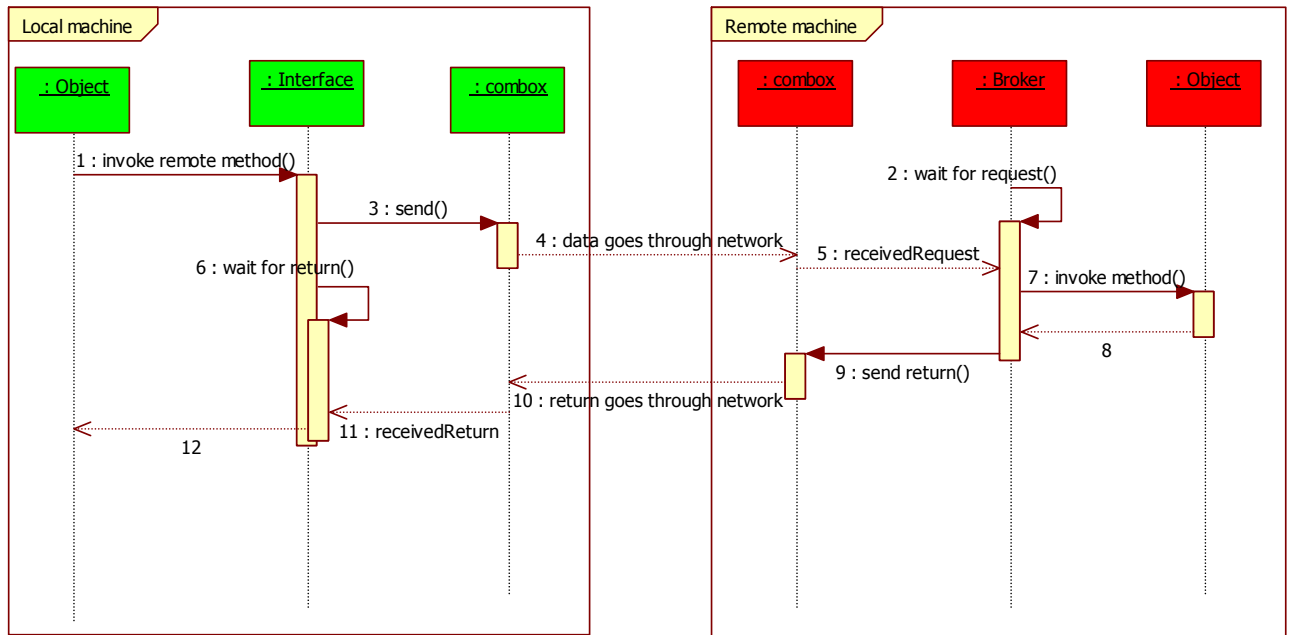


Figure 2 Simplified remote method call in POP-C++

### 2.2.1. Invocation semantics

To properly design a test scenario, it is necessarily to know the different object-sided invocation semantics of a method in POP-C++. The figure below shows the effect on the called object (Figure 3). But on the caller side also, two different semantics are possible.

- A synchronous call waits for the called method on the remote object to return
- An asynchronous call doesn't wait for the called method to return. It continues processing once the call has been posted

For more details, consult the "User and Installation Manual" of POP-C++ [4].

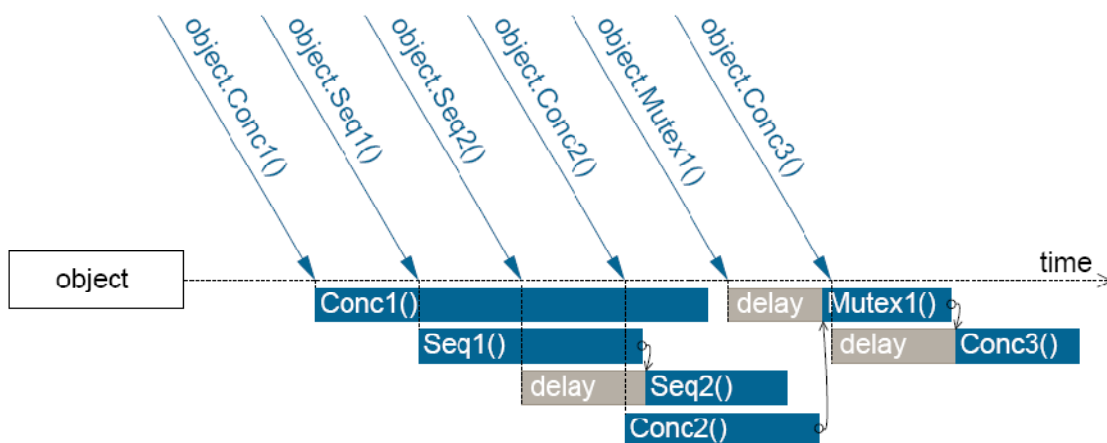


Figure 3 Different object-sided invocation requests [4]

## 2.3. The MPI standard {common}

MPI uses the message passing paradigm to communicate in a distributed memory system (Figure 4). Each memory entity is accessible only via its corresponding processor (CPU). To communicate, processors use a network which connects them to each other. Messages can be sent and received in different manners. For example, it can be passed in a blocking way, which is very safe but slow, or in a non-blocking way, which can increase the performance of a program by eliminating active waiting of the CPU.

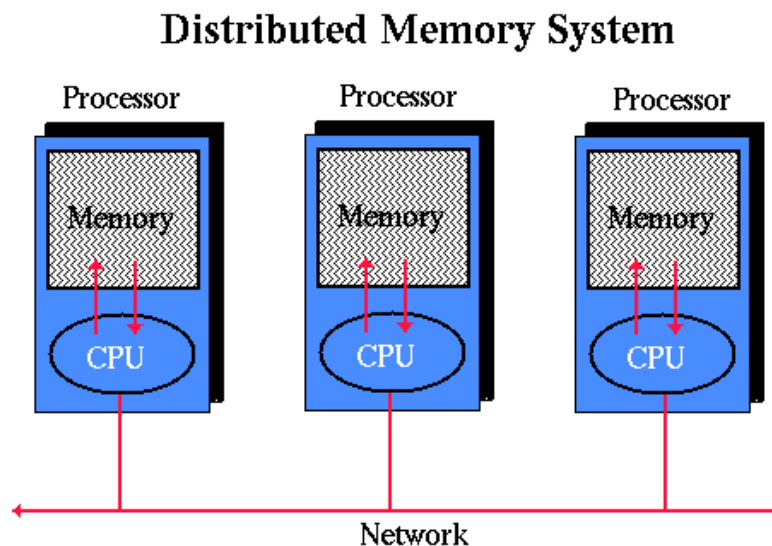


Figure 4 Distributed Memory System [2]

Message passing is used to exchange data between processes. Process A **sends** a memory buffer (data) of its application memory through the network to process B, which **receives** the message and stores it into his application memory.

### 2.3.1. MPI-processes

It is important to understand the difference between a **process** (set of executable instructions) and a **processor** (hardware)! More than one process can execute on the same processor but will nevertheless communicate via message passing. MPI differentiates processes by their **rank**. Every process is identified by a number going from 0 to N-1, where N is the number of processes running.

Every MPI-programmer has to be aware about issues related to concurrent computing. In an MPI-application, there is usually more than one process running at the same time. They communicate between each other in either a blocking or non blocking way. In case of inappropriate use of the message passing routines provided by MPI, unexpected consequences may show up (e.g. Dead-lock or data loss). Different **communication events** can take place during message passing:

- copying a message from application memory to system memory (send buffer)
- arrival of a message

### 2.3.2. SPMD

Programming in MPI applies the SPMD (**S**ingle **P**rogram, **M**ultiple **D**ata) mechanism. This means that the same program is running everywhere. To avoid that every process executes exactly the same instructions, they have to be differentiated in the program by their rank. A model which is very often used is to differentiate only process 0 from the rest of the processes.

```
if rank==0
    send(message)
else
    receive(message)
```

## 2.4. MPI message passing (point-to-point)

This chapter describes the different manners of message passing in a point-to-point communication. Point-to-point means that the message has one sender and only one receiver. For more details, consult reference [2].

Criteria for successful communication are

- The sender specifies what message to pass (buffer, size)
- The buffer size of the receiver has to be large enough to store the message
- The sender has to specify a valid destination rank
- The receiver has to specify a valid source rank
- Sender and receiver have to specify the data type they want to send/receive
- Each message is accompanied by a tag (positive integer) which has to match
- A communicator has to be specified in sends and receives. The communicator defines which collection of processes may communicate with each other.

### 2.4.1. Buffering

As mentioned before, MPI's goal is to transport a piece of memory from one node to another by passing it in a message. However, it is possible that data is not going directly from one application buffer to the other, but has to be copied to/from system buffer before (Figure 5).

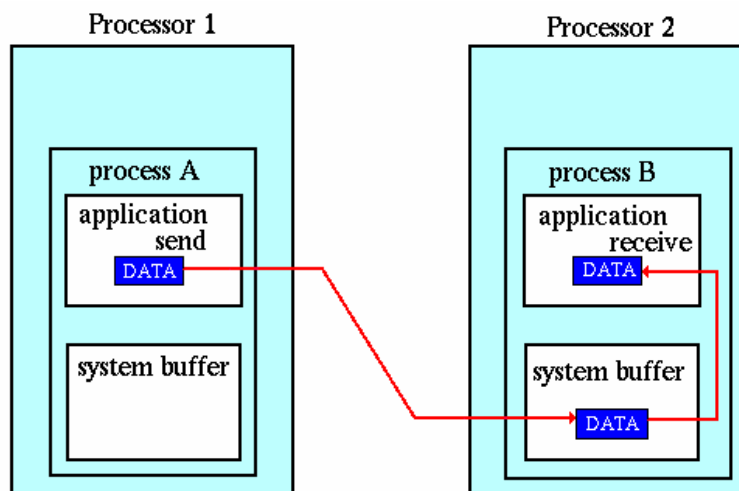


Figure 5 Message passing with buffering at receive [2]

This can happen for example, when *process B* is not yet ready to receive the message which *process A* is trying to send him. In the meantime, the message has to be copied to *process B's* system buffer, and recopied later to the application buffer.

#### 2.4.2. Blocking point-to-point communication

A blocking send/receive is the safest way to communicate.

**For sends:** The program doesn't continue until the data has been successfully sent to the corresponding receive, or copied to the system buffer. If one of the mentioned communication events is fired, it is safe to reuse the memory buffer containing the data.

**For receives:** The program is blocked until the data is completely stored in the application buffer allocated for this receive.

#### 2.4.3. Non-blocking point-to-point communication

A non-blocking routine doesn't wait the firing of a communication event. It returns directly to the caller after having posted the request of the send/receive. Thus it is up to the programmer to ensure that the memory buffer used in the corresponding send/receive is not used until the data is completely transferred from one process to the other or to the system buffer. Non-blocking communication is less safe than blocking communication, but allows processes to reduce active waiting and can improve the performance of a program.

### 2.5. The "Phoenix" cluster

**Phoenix** is a 16 node cluster for parallel processing research in the "Scalable systems labor" at the University of New Mexico (UNM). Each node contains two 2.4 GHz AMD Opteron processors, 2 GB of RAM, gigabit ethernet networking and InfiniBand networking. Only the head node contains disks. The other 15 nodes must be booted from the network. The currently installed MPI-2 implementation is *OpenMPI v.1.2.3* (see *Phoenix overview* [5]).

## 2.6. Benchmarking data sending

This test scenario consists in measuring the time to send a quantity of data from the local application buffer to the remote one (point-to-point). To measure accurate values, the operation of data sending has to be isolated in both technologies (Table 1).

#	MPI	POP-C++
1	Complete initialization procedure (MPI_INIT)	Local object is ready to send data
2	Synchronize processes	The remote object has been created and is ready to receive a request
3	<b>Start timer</b>	<b>Start timer</b>
4	Send raw quantity of data in a blocking way to the receiver	Invoke a method on the remote object by passing a raw quantity of data as a parameter
5	Copy data from senders application buffer in the system buffer	
6	Copy data from senders system buffer to receivers system buffer	
7	Copy data from system buffer to receivers application buffer	
8	The receiver copies the data in his application buffer	Remote object receives the data from a parameter
9	<b>Stop timer</b>	<b>Stop timer</b>

Table 1 Optimal benchmarking for data sending in POP-C++ and MPI

(The highlighted steps #5 to #7 are not always executed, but they might be depending on the message size)

As we can see, the operation is isolated but there is a problem with the timer. We have to start the timer on the side of the initiator and stop it on the side of the receiver. But since there is **no global clock** available in parallel computing, the timer cannot be correctly handled like this. Thus, the timer has to be started and stopped on the same side. We'll have to take this fact into consideration to design a correct test scenario.

Another point to take care about could be the existence of **operating system interruptions**. The process dedicated to send and receive messages might not be the only resource which is accessing the CPU. To minimize those interruptions, the process needs to have a high priority and the fewer other processes are running on the node the better it is. However, for the comparison between MPI and POP-C++, those interruptions can be **neglected**, because they'll happen in both technologies, and might be filtered out in the phase of interpreting the results.

The next chapter is the design of test scenarios to benchmark data sending in MPI and POP-C++ taking into consideration the above discussed points.

### 3. Design

This chapter contains the configuration of the two technologies (POP-C++, OpenMPI) on the cluster and the design of the different test scenarios.

#### 3.1. Cluster configuration

The nodes on the cluster are named as follows:

- Head node: phx\_head
- Compute nodes: phx0 – phx14

##### 3.1.1. POP-C++

To get the most accurate results for latency benchmarks, POP-C++ is configured to run only one job per computing node. The head node is not supposed to run any jobs at all. It is designated to create the parallel objects on the computing nodes and to gather the results and print them on the output.

##### 3.1.2. OpenMPI

Per default, InfiniBand is used on phoenix for communication. Nevertheless it is possible to use TCP over Ethernet by specifying it on the command line when executing the application. But it seems that this feature is not optimally implemented in OpenMPI [6]. This fact has to be considered during the test phase. To guarantee the same circumstances as in POP-C++, only one process will run on each computing node and no processes on the head node.

OpenMPI uses two different protocols depending on the message size. An *Eager* protocol for short messages and a *Rendez-vous* protocol for long messages. The boundary of these protocols is individually configurable on clusters. *Phoenix* has the following configuration:

- **For Ethernet:** TCP eager limit = 65536Bytes (64KB)
- **For InfiniBand:** openib eager limit = 12288Bytes (12KB)

During the tests, it will be interesting to see if there is a jump in that area of the resulting graph.

#### 3.2. Benchmarking latency and data sending

The most common test scenario to benchmark latency and sending data from one application buffer to the other is the **ping-pong** test. It consists in sending data to the receiver and waiting for its acknowledgment. But it would be inaccurate to measure the time for one message (and one acknowledgement) only, because there may be big differences from one result to another. To get a representative result, the scenario consists in passing data in a loop and taking the **average** of the measurements. Another way to evaluate the result could be by taking the **best** measurement.

I consider two different test scenarios in each technology, an asynchronous and a synchronous. The synchronous is the “real” ping-pong application but includes more operations which are not part of the pure sending process. This is why I expect that the asynchronous scenario will deliver more accurate results.

### 3.2.1. MPI test scenario

To understand the test scenarios, some basic MPI routines are essential to know:

<b>MPI_Irecv</b>	Posts a request to allocate a memory space which can serve as a receive buffer. After this, the process continues without waiting for the actual reception of the message. To see if the message is available in the application buffer, the requests status has to be consulted (see <i>Wait</i> routine).
<b>MPI_Isend</b>	Identifies a memory space which will serve as a send buffer. The program continues without waiting for the data to be copied out of the application buffer. To check if the data has left the buffer, the status of the send request has to be consulted (see <i>Wait</i> routine).
<b>MPI_Recv</b>	Blocks until the message is available in the application buffer.
<b>MPI_Send</b>	Blocks until the application buffer containing the data to send, is ready for reuse.
<b>MPI_Rsend</b>	Blocking ready send. Should only be used if the programmer is certain that the matching receive has already been posted.
<b>MPI_Wait</b>	MPI_Wait blocks until a specified non-blocking send or receive operation has completed.

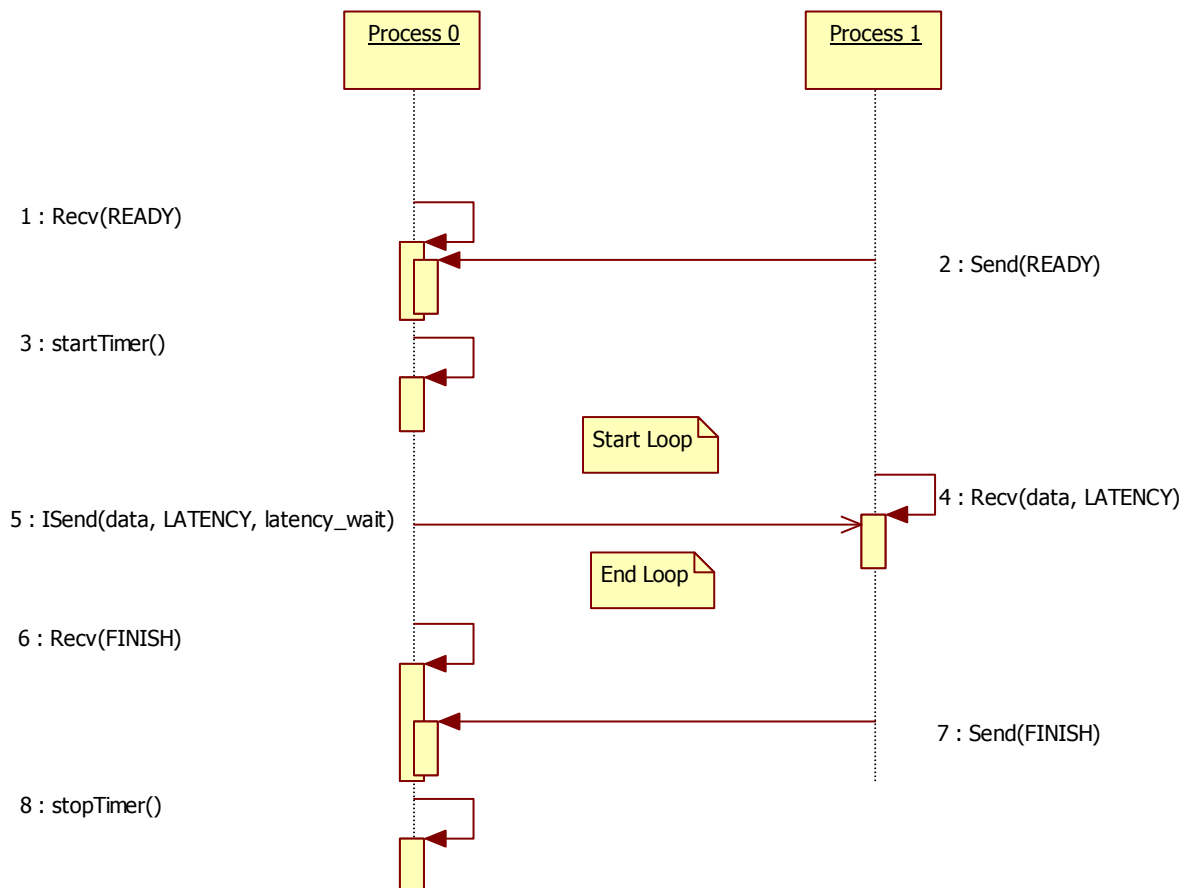


Figure 6 Asynchronous MPI latency and data sending test scenario

To synchronize the processes, a blocking “READY” message is used. Now *Process 0* can start the measurement. Within the loop, it sends unblocking messages to *Process 1* by passing the desired quantity of data. Once *Process 0* has finished the loop, it has to wait for the acknowledgment of *Process*

1 ("FINISH" message) before stopping the timer. This acknowledgment ensures that all the data has been communicated. As we can see, it is not possible to measure the time for one send only, but we can estimate it by averaging the whole loop (Figure 6).

**Remark:** Often, *MPI\_Isend* is accompanied by a call to *MPI\_Wait* to check if the data has been completely copied out of the buffer (by checking the status of the request "latency\_wait"). In our case there is no need for that, because we are not interested in the actual contents of the buffer but only in its size. And moreover, the contents never change.

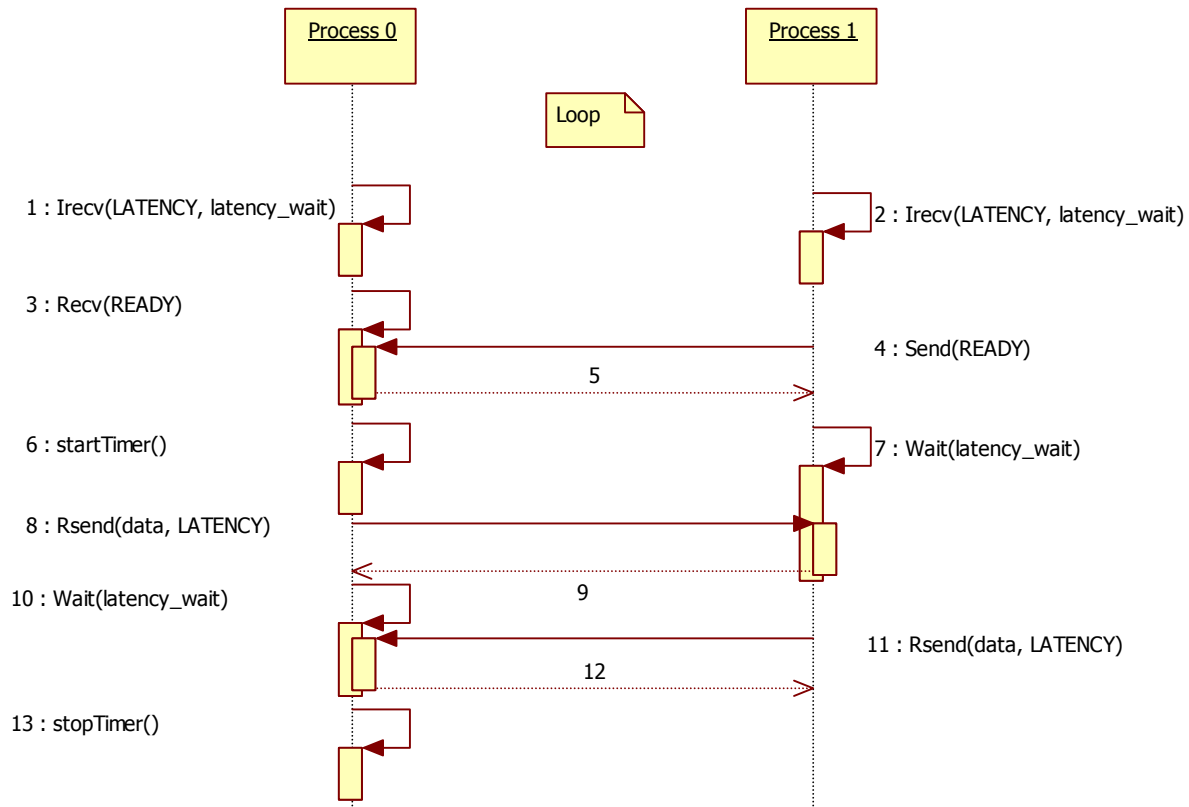


Figure 7 Synchronous MPI latency and data sending test scenario

Both processes post an unblocking *receive* at the beginning of the scenario, so they already have a memory buffer available when the message or acknowledgment arrives. The blocking "READY" message is used to synchronize the processes before starting the timer; then the actual measurement takes place. *Process 0* sends the data first and then waits for the posted receive request and *Process 1* does exactly the opposite (Figure 7). Contrary to the asynchronous test, the time for a single send can be captured. Thus we have possibility to figure out minimum and maximum time for data sending, not only the average of the entire loop.



### 3.2.2. POP-C++ test scenario

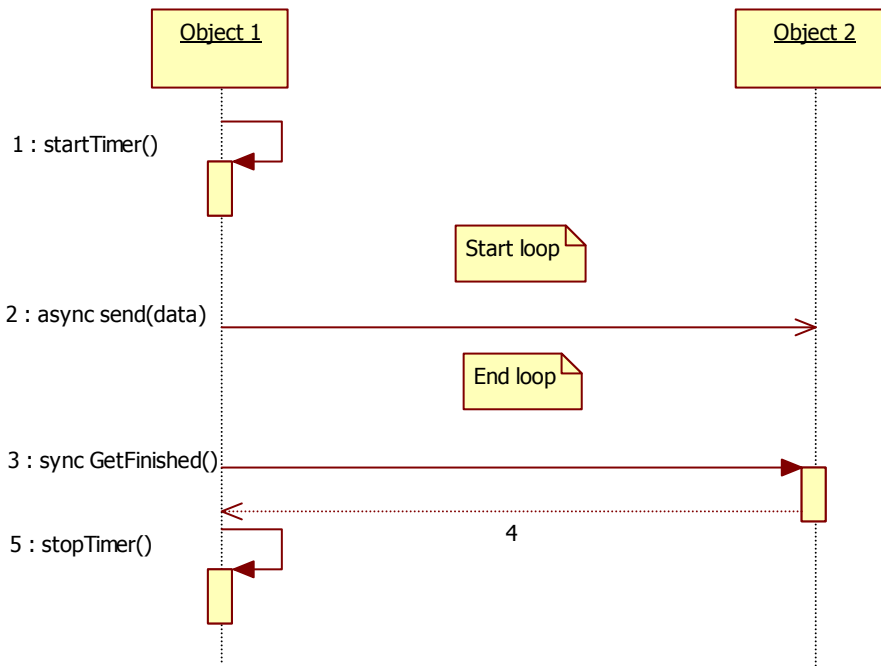


Figure 8 Asynchronous POP-C++ latency and data sending test scenario

The main program creates two parallel objects. *Object 1* starts the timer. Within a loop, it invokes an asynchronous method on *Object 2* by passing data in a parameter. Once the loop has finished, *Object 1* calls a synchronous method on *Object 2*. When this method call returns, all preceding asynchronous calls have been processed and *Object 1* can stop the timer in order to calculate the average result (Figure 8). In this case, it is not possible to measure minimum and maximum values of a single send.

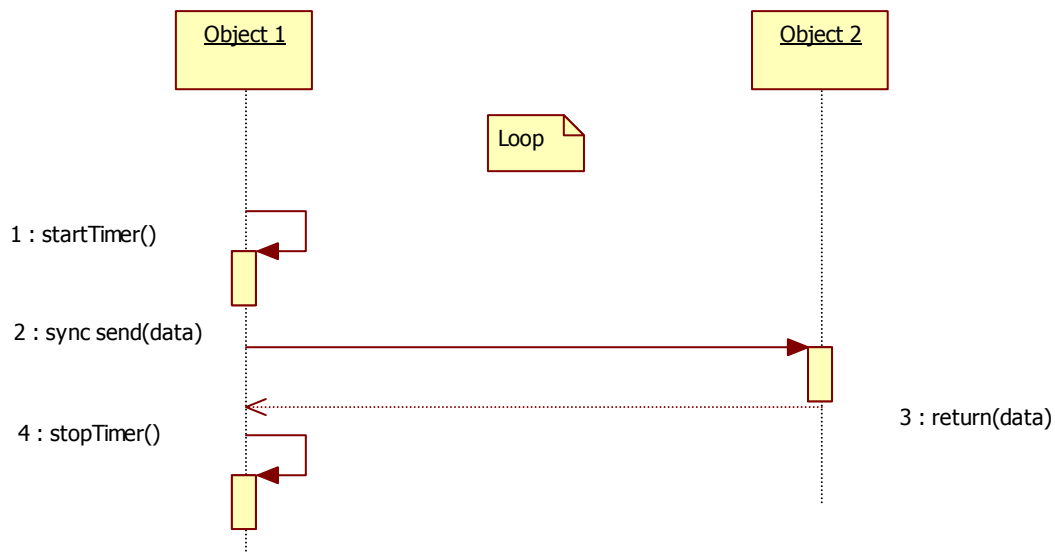


Figure 9 Synchronous POP-C++ latency and data sending test scenario

*Object 1* simply calls a synchronous method on *Object 2* by passing data as a parameter. This method returns the same quantity of data back to the caller, *Object 1* (Figure 9). It is possible to capture minimum, maximum and average values used for the data sending process.

The next chapter presents the implementation of the test scenarios in the current chapter.

## 4. Implementation

Four different test programs have been implemented, two in each technology, and are introduced in this chapter. Every program prints the results to the output:

- Names or IP addresses of the nodes where a process/object is executed
- Size of the passed message in bytes
- Average time for data sending in micro seconds
- Estimated average bandwidth in mega bytes per second (ignoring latency)

In addition to that, the synchronous test programs print:

- Minimum and maximum time for data sending in micro seconds
- Estimated Minimum and maximum bandwidth in mega bytes per second (ignoring latency)

### 4.1.MPI

The MPI test programs are written in the C programming language.

#### 4.1.1. Asynchronous latency and data sending

By default, the program starts by sending a 0byte message from *process 0* to *process 1*. Then *process 0* waits for the acknowledgment of the same length. This is repeated 1000 times to print the average time to the output (see screenshot Figure 10). Then the program does the same for messages of 16, 32, 48, ..., 1024 bytes in length.

Different arguments can be passed on the command line to personalize the test:

- -s: length of the first message (**start length:** default 0)
- -e: maximum length of the last message (**end length:** default 1024)
- -i: how to **increment** the message length (default 16)
- -t: number of **trials** for each message length (default 1000)
- -k: defines that the values in -s, -e, -i are indicated in **kilobytes**

By executing the program with *mpiexec*, the number of processes to run has to be specified. Useful communication (for this test) takes place between only two processes. That's why a minimum of two is required, but it's not limited to that. If more processes are running, the ones which are not involved in the useful point-to-point communication are just consuming a message in every iteration which has been send by *process 0*. This allows us to test how the number of running processes affects performance.

```
schram@phoenix:~/mpi/async_latency
*****
Process1 phx1
Process0 phx0
#
# Results for 10000 trials each of length 0 through 10 in increments of 1
#
# Exchanging data between nodes 0 and 1.
# Length          Latency          Bandwidth
# in bytes        in micro seconds  in MB/s
0                13.39             0.00
1                 5.73             0.17
2                 5.83             0.33
3                 5.83             0.49
4                 5.75             0.66
5                 5.80             0.82
6                 5.78             0.99
7                 5.81             1.15
8                 5.81             1.31
9                 5.75             1.49
10                5.81             1.64
```

Figure 10 Screenshot of asynchronous test program in MPI

### Examples:

```
mpirun -np 2 -mca btl self,tcp -hostfile hostfile ./main -s 0 -e 1024 -i 64 -t 1000
```

Run the test program with 2 processes over tcp. The available computing nodes are listed in a *hostfile*. Provide results for 1000 trials each of length 0 through 1024B in increments of 64B.

```
mpirun -np 8 -hostfile hostfile ./main -s 1 -e 128 -i 16 -t 500 -k
```

Run the test program with 8 processes over InfiniBand. The available computing nodes are listed in a *hostfile*. Provide results for 500 trials each of length 1 through 128KB in increments of 16KB.

#### 4.1.2. Synchronous latency and data sending

By default, the program starts by sending a 0byte message from *process 0* to *process 1*. Then *process 0* waits for the acknowledgment of the same length. This is repeated a 1000 times. During that operation, the minimum, maximum and total value is continuously updated in a variable to print the result (minimum, average and maximum) to the output (see screenshot Figure 11). Then the program does the same for messages of 16, 32, 48, ..., 1024 bytes length.

Different arguments can be passed on the command line to personalize the test:

- **-s:** length of the first message (**start length:** default 0)
- **-e:** maximum length of the last message (**end length:** default 1024)
- **-i:** how to **increment** the message length (default 16)
- **-t:** number of **trials** for each message length (default 1000)
- **-k:** defines that the values in **-s**, **-e**, **-i** are indicated in **kilobytes**

By executing the program with *mpirun*, the number of processes to run has to be specified. Useful communication (for this test) takes place between only two processes. That's why a minimum of two is required, but it's not limited to that. If more processes are running, the ones which are not involved in the useful point-to-point communication are just consuming a message sent by *process 0* in each iteration. This allows us to test how the number of running processes affects performance.

```
schram@phoenix:~/mpi/sync_latency
*****
Process1 phx1
Process0 phx0
#
# Results for 10000 trials each of length 0 through 10 in increments of 1
#
# Exchanging data between nodes 0 and 1.
# Length          Latency          Bandwidth
# in bytes        in micro seconds    in MB/s
#
#   minimum      average      maximum      minimum      average      maximum
#
# 0      41.48      50.38      96.44      0.00      0.00      0.00
# 1      48.40      51.02      86.90      0.01      0.02      0.02
# 2      47.56      50.82      71.53      0.03      0.04      0.04
# 3      47.92      50.96      90.00      0.03      0.06      0.06
# 4      47.56      50.81      63.06      0.06      0.08      0.08
# 5      47.92      50.76      107.53     0.04      0.09      0.10
# 6      48.40      51.13      64.02      0.09      0.11      0.12
# 7      48.40      51.38      83.57      0.08      0.13      0.14
# 8      48.40      51.15      75.58      0.10      0.15      0.16
# 9      48.40      51.23      80.47      0.11      0.17      0.18
# 10     48.52      51.40      64.02      0.15      0.19      0.20
```

Figure 11 Screenshot of synchronous test program in MPI

### Examples:

```
mpiexec -np 2 -mca btl self,tcp -hostfile hostfile ./main -s 0 -e 1024 -i 64 -t 1000
```

Run the test program with 2 processes over tcp. The available computing nodes are listed in a *hostfile*. Provide results for 1000 trials each of length 0 through 1024B in increments of 64B.

```
mpiexec -np 8 -hostfile hostfile ./main -s 1 -e 128 -i 16 -t 500 -k
```

Run the test program with 8 processes over InfiniBand. The available computing nodes are listed in a *hostfile*. Provide results for 500 trials each of length 1 through 128KB in increments of 16KB.

## 4.2.POP-C++

This chapter explains the POP-C++ test program implementations corresponding to the design in section 3.2.2.

### 4.2.1. Asynchronous latency and data sending

This implementation consists in invoking an asynchronous method on a remote object by passing data in a parameter to measure latency. See the signature of this method below:

```
async void pingAC(
    [in, size=len] char *buf, int len
);
```

By default, the program creates two parallel objects. *Object 1* calls the asynchronous method on *Object 2* by passing a buffer of 0bytes length a 1000 times. After that, *Object 1* calls a synchronous method on *Object 2* to make sure that all asynchronous invocations have been treated and stops the timer. Then the main program prints the average value to the output (see screenshot Figure 12) and redoes the same for buffers of 16, 32, 48, ..., 1024 bytes length.

Different arguments can be passed on the command line to personalize the test:

- -s: length of the first message (**start length**: default 0)
- -e: maximum length of the last message (**end length**: default 1024)

- -i: how to **increment** the message length (default 16)
- -n: **number of parallel objects** to create (default 2)
- -t: number of **trials** for each message length (default 1000)
- -k: defines that the values in -s, -e, -i are indicated in **kilobytes**

One has the choice to create more than two parallel objects. Useful communication (for this test) takes place between only two parallel objects. That's why a minimum of two is required, but it's not limited to that. If more parallel objects are running, the ones which are not involved in the useful point-to-point communication are just consuming a method call initiated by *Object 1* in each iteration. This allows us to test how the number of living parallel objects affects performance.

```
schram@phoenix:~/popc/async_latency
# in bytes      in micro seconds      in MB/s
0               45.20                 0.00
1               16.58                 0.06
2               16.57                 0.12
3               16.55                 0.17
4               16.35                 0.23
5               16.65                 0.29
6               16.66                 0.34
7               16.67                 0.40
8               16.66                 0.46
9               16.69                 0.51
10              16.74                 0.57
#...done
[objectmonitor.cc:69]Check parallel objects...0 object alive

*****
[codemgr.cc:13]Now destroy CodeMgr

[schram@phoenix async_latency]$
```

Figure 12 Screenshot of asynchronous test program in POP-C++

### Examples:

```
parocrun objmap ./main -s 0 -e 1024 -i 64 -n 2 -t 1000
```

Run the test program with 2 parallel objects. Provide results for 1000 trials each of length 0 through 1024B in increments of 64B.

```
parocrun objmap ./main -s 1 -e 128 -i 16 -n 8 -t 500 -k
```

Run the test program with 8 parallel objects. Provide results for 500 trials each of length 1 through 128KB in increments of 16KB.

#### 4.2.2. Synchronous latency and data sending

The idea of this test is to call a synchronous method on a remote object by passing data in a parameter to measure latency. See the signature of this method below:

```
sync void pingSC(
    [in, size=len] char *buf, [out, size=len] char *retbuf, int len
);
```

By default, the program creates two parallel objects. *Object 1* calls the synchronous method on *Object 2* by passing a buffer of Obytes length. Since the second parameter is an output parameter, it has to be

transported back to *Object 1*. This is repeated a 1000 times. Then the main program prints the minimum, average and maximum values to the output (see screenshot Figure 13) and redoes the same for buffers of 16, 32, 48, ..., 1024 bytes length.

Different arguments can be passed on the command line to personalize the test:

- `-s`: length of the first message (**start length**: default 0)
- `-e`: maximum length of the last message (**end length**: default 1024)
- `-i`: how to **increment** the message length (default 16)
- `-t`: number of **trials** for each message length (default 1000)
- `-n`: **number of parallel objects** to create (default 2)
- `-k`: defines that the values in `-s`, `-e`, `-i` are indicated in **kilobytes**

One has the choice to create more than two parallel objects. Useful communication (for this test) takes place between only two parallel objects. That's why a minimum of two is required, but it's not limited to that. If more parallel objects are running, the ones which are not involved in the useful point-to-point communication are just consuming a method call initiated by *Object 1* in each iteration. This allows us to test how the number of living parallel objects affects performance.

```
schram@phoenix:~/popc/sync_latency
Pong 192.168.0.3
Ping-pong encoding: raw
Ping-pong protocol: socket
Ping 192.168.0.4
# Exchanging data between nodes 192.168.0.4 and 192.168.0.3.
# Length          Latency          Bandwidth
# in bytes        in micro seconds  in MB/s
#
# 0  minimum  average  maximum  minimum  average  maximum
# 0  44.47    45.10    67.47    0.00     0.00     0.00
# 1  45.42    46.13    57.10    0.02     0.02     0.02
# 2  45.42    46.10    57.94    0.03     0.04     0.04
# 3  45.42    46.09    56.51    0.05     0.06     0.06
# 4  45.42    45.99    57.94    0.07     0.08     0.08
# 5  45.42    46.02    59.60    0.08     0.10     0.10
# 6  45.42    46.03    57.46    0.10     0.12     0.13
# 7  45.42    46.04    56.03    0.12     0.15     0.15
# 8  45.42    46.00    57.58    0.13     0.17     0.17
# 9  45.42    46.03    57.58    0.15     0.19     0.19
# 10 45.42    45.99    56.03    0.17     0.21     0.21
```

Figure 13 Screenshot of synchronous test program in POP-C++

### Examples:

```
parocrun objmap ./main -s 0 -e 1024 -i 64 -n 2 -t 1000
```

Run the test program with 2 parallel objects. Provide results for 1000 trials each of length 0 through 1024B in increments of 64B.

```
parocrun objmap ./main -s 1 -e 128 -i 16 -n 8 -t 500
```

Run the test program with 8 parallel objects. Provide results for 500 trials each of length 1 through 128KB in increments of 64KB.

The results provided by these test programs are presented in the next chapter.



## 5. Results

This chapter includes a prediction of the tests which will be performed and the actual measurements to finally compare theory and reality.

### 5.1. Prediction

To detect and explain abnormal behavior in the data sending measurements, a prediction of latency and used bandwidth is necessary. This prediction is searched by making basic measurements several times. By sending 0Byte messages repetitively, the latency can be captured. To find the average bandwidth, we send messages in the range of 10 – 100Kbytes. The formula  $B = \frac{len}{\Delta t - l}$  (where  $len$  is the size of the data,  $\Delta t$  the time used for the operation and  $l$  the estimated latency) lets us calculate the bandwidth of each step.

The used network is Gigabit Ethernet. This means that the theoretical attainable bandwidth amounts to

$$\frac{1000'000'000 \text{ bits}}{8 \frac{\text{bits}}{\text{B}} * 1024 \frac{\text{B}}{\text{KB}} * 1024 \frac{\text{KB}}{\text{MB}}} \cong 119 \text{ MB/s.}$$

### 5.1.1. OpenMPI asynchronous

By running the test program as follows on the cluster, the average latency can be determined. It is between 5 and 6 $\mu$ s (Figure 14). The first result is not taken into consideration. TCP seems to do need a “warmup” when running over TCP [6].

```
mpiexec -np 2 -hostfile hostfile -mca btl self,tcp ./main -s 0 -e 0 -i 0 -t 10000
```

```
# Exchanging data between nodes 0 and 1.
# Length          Latency
# in bytes        in micro seconds
  0                13.53
  0                 5.67
  0                 5.70
  0                 5.72
  0                 5.74
  0                 5.70
  0                 5.72
  0                 5.72
  0                 5.69
  0                 5.75
  0                 5.68
  0                 5.73
  0                 5.73
  0                 5.68
```

Figure 14 Output to define latency of OpenMPI asynchronous

The same program is executed to find the average Bandwidth, but with different arguments.

```
mpiexec -np 2 -hostfile hostfile -mca btl self,tcp ./main -s 10240 -e 102400 -i 10240 -t 10000
```

Size [Bytes]	$\Delta t$ –latency [ $\mu$ s]	Bandwidth [MB/s]
10240	89.26	114.72104
20480	171.05	119.731073
30720	257.99	119.074383
40960	344.92	118.752174
51200	431.96	118.529493
61440	518.84	118.418009
71680	765.73	93.6100192
81920	871.07	96.4820333
92160	923.83	99.7586136
102400	1018.79	100.511391
<b>Average</b>		<b>109.958823</b>

Table 2 Output to define average bandwidth of OpenMPI asynchronous

The average bandwidth is  $\sim$ 110MB/s (Table 2).



### 5.1.2. OpenMPI synchronous

By running the test program as follows on the cluster, the average latency can be determined. It is between 49 and 50 $\mu$ s (Figure 15).

```
mpiexec -np 2 -hostfile hostfile -mca btl self,tcp ./main -s 0 -e 0 -i 0 -t 1000
```

```
# Exchanging data between nodes 0 and 1.
#   Length           Latency
#   in bytes         in micro seconds
size                minimum      average      maximum
   0                 46.49       49.56       133.51
   0                 39.94       49.45       141.02
   0                 46.97       49.51       174.52
   0                 42.44       49.52       58.53
   0                 39.94       49.56       54.96
   0                 40.05       49.54       52.09
   0                 40.05       49.63       53.05
   0                 42.44       49.51       53.41
   0                 39.94       49.67       53.41
   0                 45.06       49.69       105.50
   0                 42.08       49.65       60.44
```

Figure 15 Output to define latency of OpenMPI synchronous

The same program is executed to find the average Bandwidth, but with different arguments.

```
mpiexec -np 2 -hostfile hostfile -mca btl self,tcp ./main -s 10240 -e 102400 -i 10240 -t 1000
```

Size [Bytes]	$\Delta t$ -latency [ $\mu$ s]	Bandwidth [MB/s]
10240	130.61	74.5215
20480	210.14	92.69904
30720	307.23	99.63674
40960	405.66	100.3946
51200	491.57	103.4573
61440	583.35	104.6981
71680	745.43	95.68816
81920	834.65	97.91313
92160	911.59	100.2349
102400	1008.42	101.0051
<b>Average</b>		<b>97.02487</b>

Table 3 Output to define average bandwidth of OpenMPI synchronous

The average bandwidth is  $\sim$ 97MB/s (Table 3).

### 5.1.3. POP-C++ asynchronous

By running the test program as follows on the cluster, the average latency can be determined. In this case, it is between 8 and 9 $\mu$ s (Figure 16).

```
parocrun objmap ./main -s 0 -e 0 -i 0 -t 1000
```

```
# Exchanging data between nodes 192.168.0.4 and 192.168.0.3.
# Length          Latency          Bandwidth
# in bytes        in micro seconds in MB/s
0                 8.87             0.00
0                 8.83             0.00
0                 8.81             0.00
0                 8.85             0.00
0                 8.85             0.00
0                 8.83             0.00
0                 8.84             0.00
0                 8.83             0.00
0                 8.82             0.00
0                 8.81             0.00
0                 8.17             0.00
0                 8.98             0.00
```

Figure 16 Output to define latency of POP-C++ asynchronous

The same program is executed to find the average Bandwidth, but with different arguments.

```
parocrun ./main -s 10240 -e 102400 -i 10240 -t 1000
```

Size [Bytes]	$\Delta t$ –latency [ $\mu$ s]	Bandwidth [MB/s]
10240	77.93	128.1922884
20480	164.5	123.010391
30720	251.57	121.1308702
40960	338.61	120.2548369
51200	425.67	119.7212739
61440	512.73	119.3496377
71680	599.77	119.1073595
81920	686.84	118.9159372
92160	773.99	118.7736007
102400	860.99	118.6544768
<b>Average</b>		<b>120.7110672</b>

Table 4 Output to define average bandwidth of POP-C++ asynchronous

The average bandwidth is  $\sim 121$ MB/s (Table 4). This is even higher than the theoretical attainable bandwidth. An explanation for that is the fact that the latency measurement takes CPU cycles into account which extend the measured time and thus increases the calculated bandwidth.

### 5.1.4. POP-C++ synchronous

By running the test program as follows on the cluster, the average latency can be determined. It is between 46 and 47 $\mu$ s (Figure 17).

```
parocrun objmap ./main -s 0 -e 0 -i 0 -t 1000
```

```
# Exchanging data between nodes 192.168.0.4 and 192.168.0.3.
#   Length           Latency
#   in bytes        in micro seconds
#
#           minimum   average   maximum
0           45.42     46.17    60.08
0           45.42     46.11    56.03
0           45.42     46.07    53.05
0           45.42     46.12    59.96
0           45.42     46.08    62.58
0           45.42     46.05    55.07
0           45.42     46.09    58.05
0           45.42     46.01    48.04
0           45.42     46.04    63.54
0           45.42     46.05    55.55
0           45.42     46.06    55.55
0           45.42     46.15    54.96
0           45.42     46.11    55.07
```

Figure 17 Output to determine latency of POP-C++ synchronous

The same program is executed to find the average Bandwidth, but with different arguments.

```
parocrun ./main -s 10240 -e 102400 -i 10240 -t 1000
```

Size	$\Delta t$ –latency [ $\mu$ s]	Bandwidth [MB/s]
10240	130.01	78.5517
20480	214.1	94.05281
30720	320.49	95.8293
40960	424.19	96.03076
51200	523.08	97.73044
61440	632.42	97.18905
71680	730.58	98.19985
81920	848.45	96.66533
92160	966.67	95.39286
102400	1066.39	96.40188
<b>Average</b>		<b>94.6044</b>

Table 5 Output to define average bandwidth of POP-C++ synchronous

The average bandwidth is  $\sim$ 95MB/s (Table 5).

### 5.1.5. Conclusion

By using the predicted information about latency and bandwidth, the time used for a certain message size can be calculated using  $t = \frac{len}{B} + l$ , where  $l$  is the latency,  $len$  the message size and  $B$  the bandwidth. This formula corresponds to a linear function  $y = ax + b$  and thus  $l$  is the y-axis intercept and  $\frac{1}{B}$  the slope of the function ( $len$  is the x-axis variable).

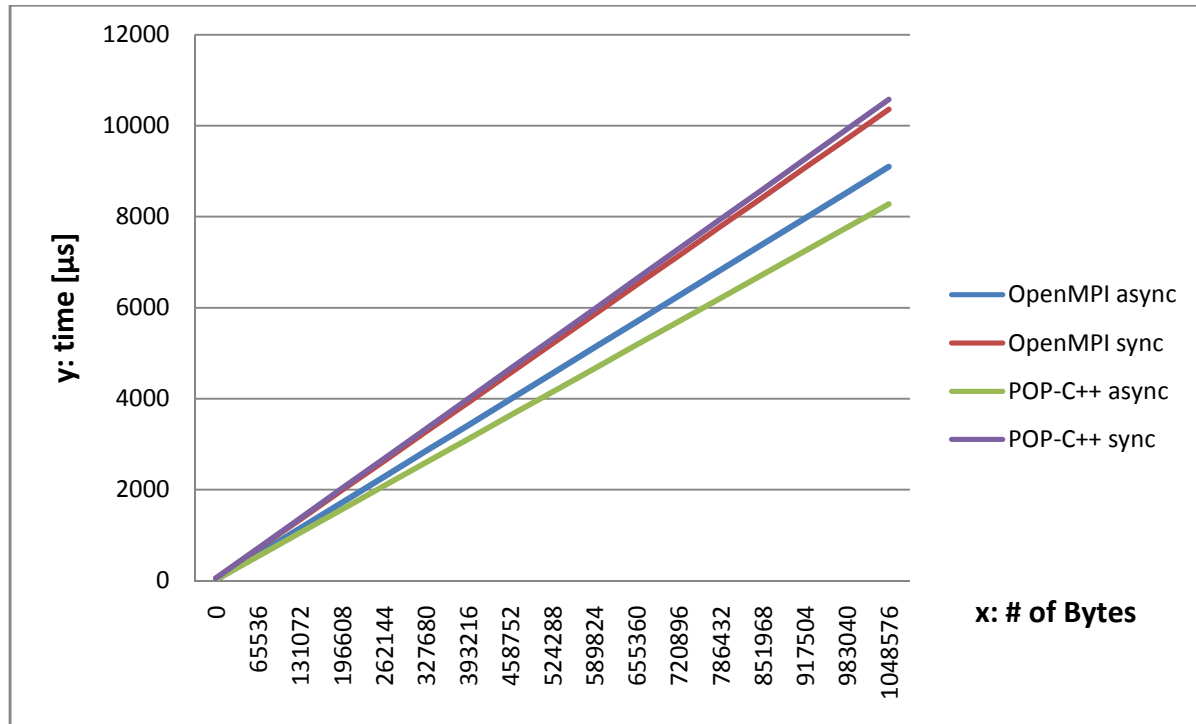


Figure 18 Prediction of the test scenarios

The asynchronous programs in both technologies should provide better results than the synchronous ones (Figure 18).

According to the chart, additional work has to be done in the synchronous versions when the message size increases because the slope of the function is steeper. Actually I cannot see the reason why, because all programs use the same protocol and encoding to transport the data over the network. We will see if the measurements correspond to those predictions.

## 5.2. Obtained results

To keep the charts more readable and lighter, the differences between prediction and measurement are discussed in section 0. The current section compares the different test scenarios ran on the cluster.

### 5.2.1. Latency

Let us check if the estimated latency corresponds to the reality. To do this, the test program sends very small messages.

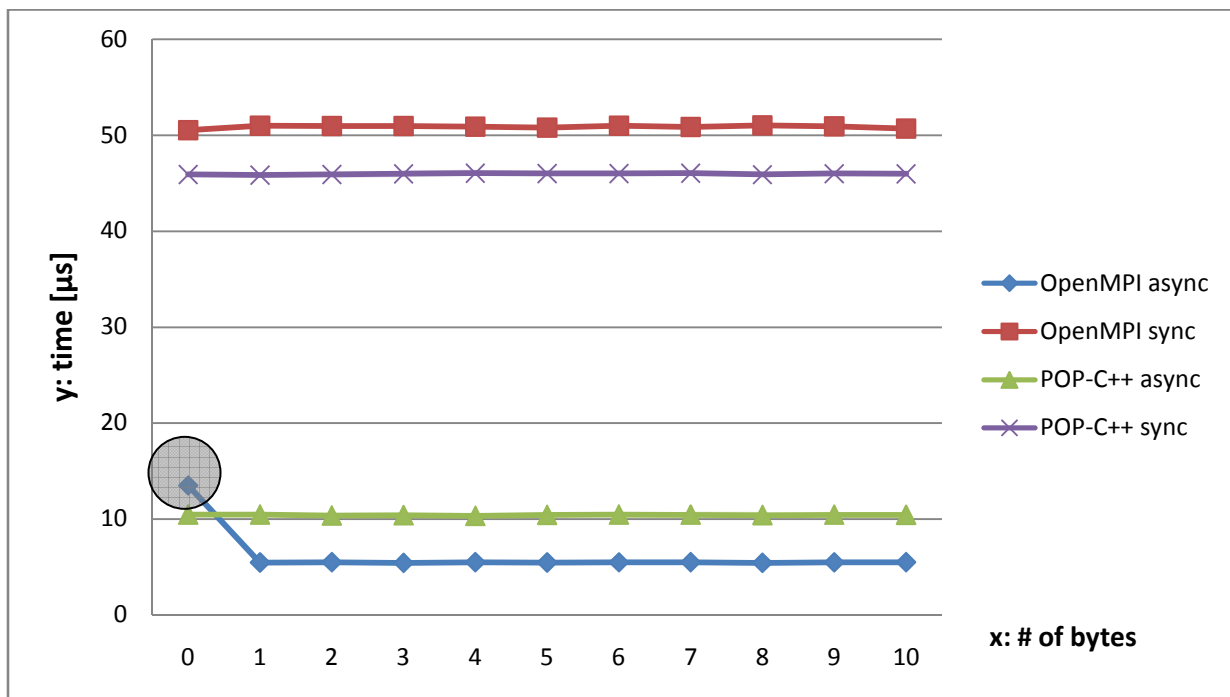


Figure 19 Test in range of single bytes

The circled area is not taken into consideration because of the “warmup” side effect (see section 0). All the other values correspond to the prediction. It is already visible that the asynchronous programs in both technologies provide better results in terms of latency (Figure 19). Explanations about that effect below:

**For POP-C++:** The reason resides in the different signatures (invocation semantics, parameters) of the principal methods used for the measurement (see section 0). A header of each input and output parameter has to be serialized in a buffer belonging to the POP-C++ runtime before sending or receiving data. In the synchronous version, more of those operations have to be done (more parameters) on local and remote objects and each takes an amount of time which is in the range of 1-5 $\mu$ s. Unfortunately I didn't have the possibility to modify the POP-C++ runtime on the cluster to measure the exact time, but could only do the test on my notebook.

**For OpenMPI:** The synchronous program uses calls to *MPI\_Wait* to synchronize with the posted non-blocking receives. Additionally to that, the non-initiating process has to invoke routines to know the message size, source and tag before it is able to return the same message type to the initiator.

### 5.2.2. Data sending

On the chart below (Figure 20), the circled areas are considered as victims of either the “warmup” side effect (see section 0) or of operating system and network related perturbations.

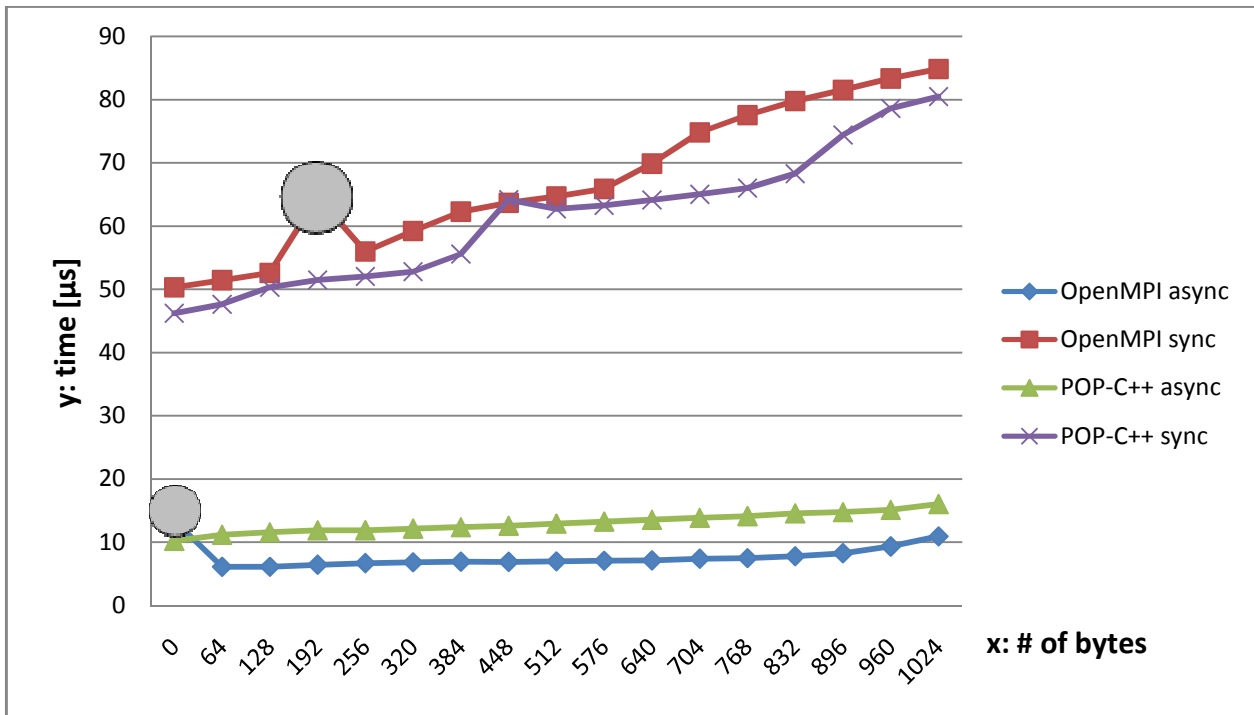


Figure 20 Messages of 0 to 1KB in increments of 64B

**Asynchronous:** Asynchronous programs show a strictly linear behavior, which corresponds to theory. Only the OpenMPI graph seems to become steeper at its end. We will see in the next test, if this trend continues.

**Synchronous:** The synchronous versions are linear as well in a global view. But there are more ups and downs in the graphs. The only explanation I can find is the window size problematic related to the TCP protocol. We will see if the next measurement shows similar behavior.

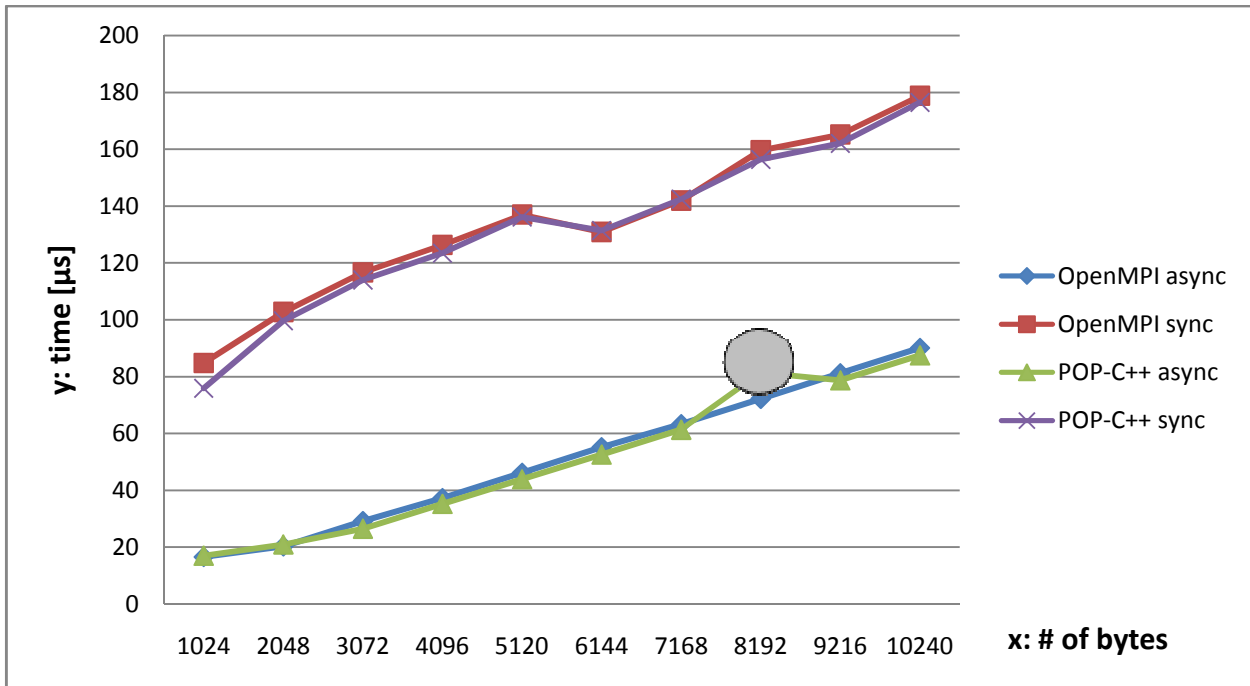


Figure 21 Messages of 1KB to 10KB in increments of 1KB

The circled area is considered as a victim of OS noise or other perturbations.

**Asynchronous:** In fact, the trend observed in the preceding test continues in the current one (Figure 21). For messages <1KB, OpenMPI was faster than POP-C++ and between 1-10KB the opposite is the case. But we can see that the two graphs stay close to each other. The difference is always smaller than 3μs.

**Synchronous:** The areas of abnormal behavior in those two slopes are exactly at the same places on the x-axis and thus independent from the technology. I repeated this test several times to be sure that the result always looks like that. The reason must lie in the underlying transporting protocol (TCP), which has to regulate the data flow. Window size decreases at every reception of a message and once it is too small to receive another message (not enough space in the reception buffer), additional TCP data packets are circulating between sender and receiver to inform each other about the situation. Depending on how the message size matches with the window size available during the communication process the number of those packets can vary. When the same test program runs over InfiniBand, none of these jumps appears. The strange thing is that the asynchronous versions don't show that jump. Bidirectional communication in the synchronous versions could be the source of this behavior. It would be interesting to see, if the test programs provide the same results on another cluster.

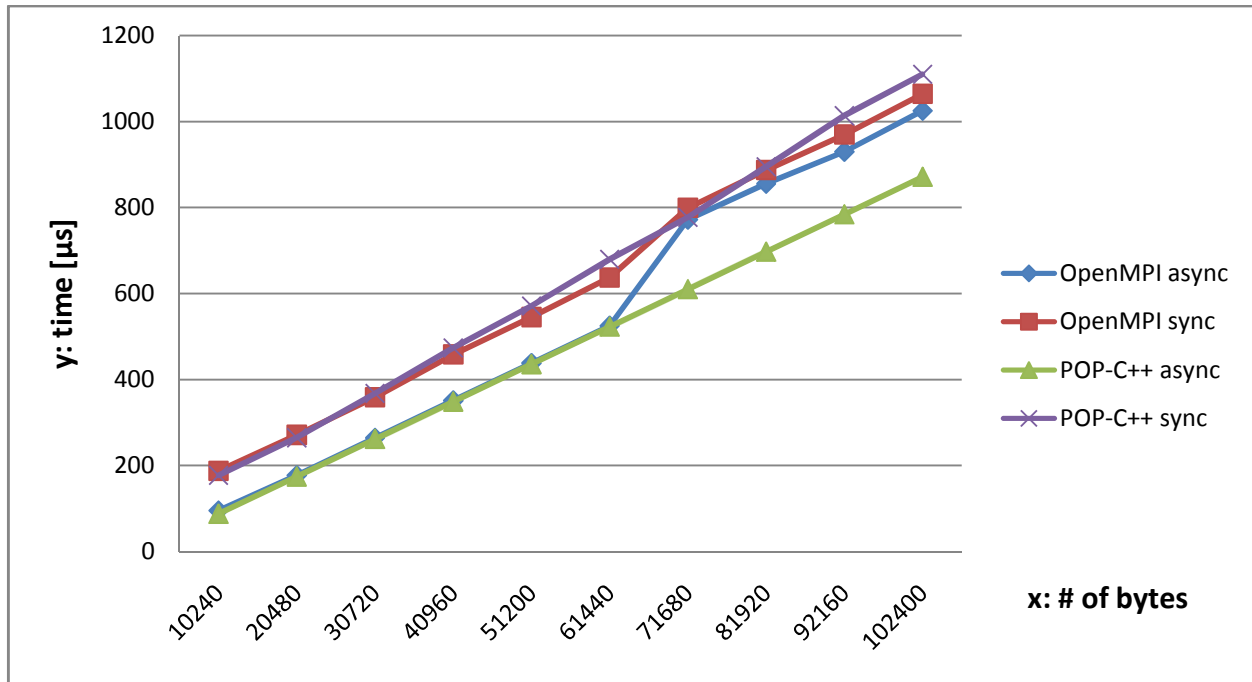


Figure 22 Messages of 10KB to 100KB in increments of 10KB

The bigger the messages are, the more we can see the linear behavior in the chart.

Between 60 and 70KB a jump can be observed for OpenMPI programs (Figure 22). This is due to the *Eager* protocol boundary fixed to 64KB for OpenMPI on the cluster (see section 3.1.2). For bigger messages, OpenMPI uses the *Rendez-vous* protocol which performs a “handshake” before the message is communicated. The jump is bigger in the asynchronous version. That can be explained with the different routines which are used to send the data (ISend, RSend; see section 0).

Test program	Standard deviation [MB/s]
OpenMPI asynchronous	9.8
OpenMPI synchronous	5.9
POP-C ++ asynchronous	0.4
POP-C++ synchronous	4.8

Table 6 Standard deviation for messages of 10KB to 100KB

In the table above (Table 6), the standard deviation of the bandwidth used for all four test programs is listed. It is bigger for OpenMPI programs in this range of messages. This is due to the protocol change at 64KB. POP-C++ async provides almost perfectly linear results.



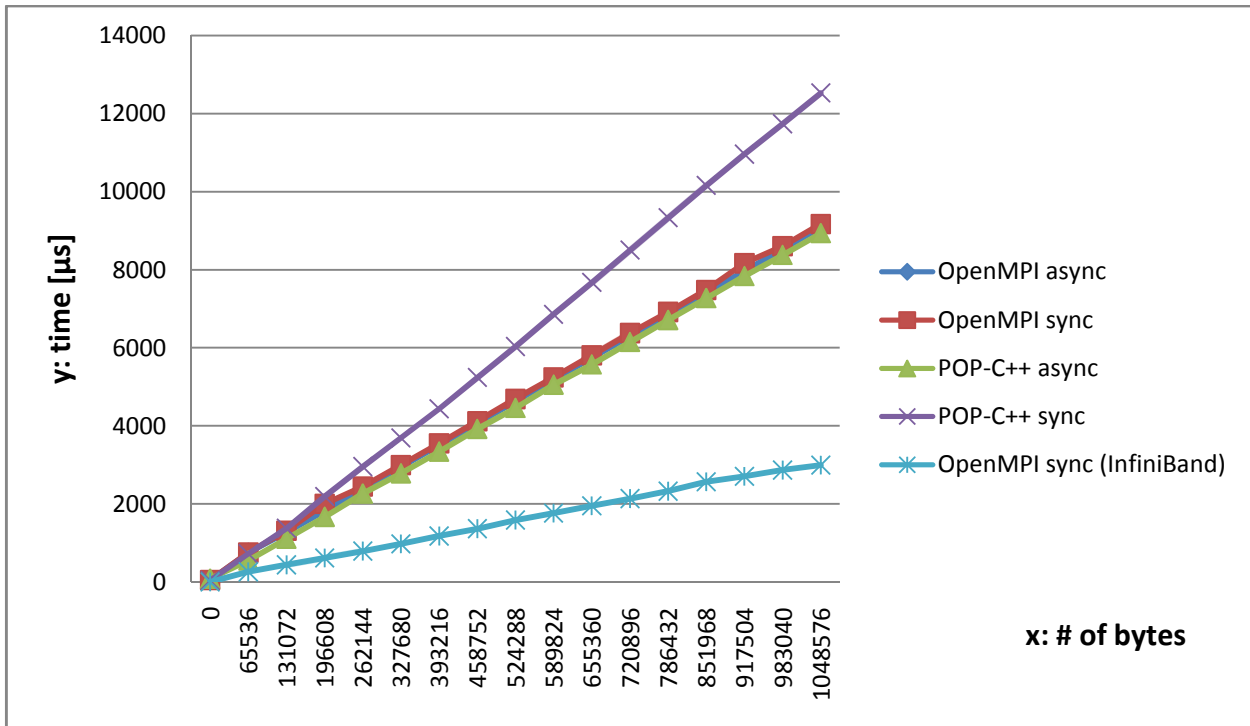


Figure 23 Messages of 0 to 1MB in increments of 64KB

In a more distant point of view, OpenMPI async, OpenMPI sync and POP-C++ async are “playing” on the same level. Unlike the prediction, the synchronous OpenMPI program corresponding graph is shallower.

Only POP-C++ sync loses performance by increasing the message length (Figure 23). This indicates that the mechanism for output parameters in POP-C++ might be not optimally implemented. I can’t find another reason, because the conditions for the communication of the data itself are identical as they are for the asynchronous version. It could be the thread, responsible to return the value, which stays longer in an inactive state depending upon the size of the returned parameter.

I added another line on the chart to show once the performances of OpenMPI when it’s running over InfiniBand. The bandwidth used in this test is more than doubled compared to OpenMPI over TCP.

### 5.2.3. Measurement vs. prediction

#### 5.2.3.1. OpenMPI asynchronous

The results obtained in the measurements globally correspond to the calculated prediction. Differences can be observed when OpenMPI changes the protocol from *Eager* to *Rendez-vous* (Figure 22). At this point a temporary performance decrease of up to 100 $\mu$ s shows up. The chart below shows the global behavior of prediction and measurement (Figure 24).

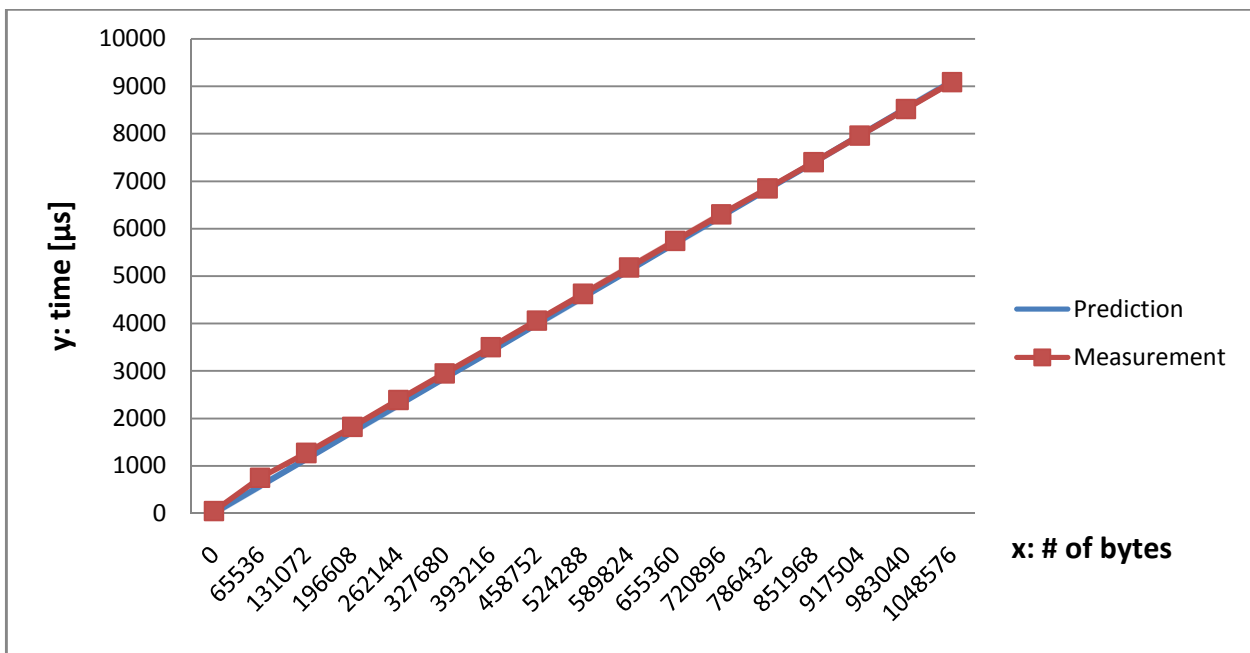


Figure 24 Global differences between prediction and measurement in OpenMPI async

#### 5.2.3.2. OpenMPI synchronous

The prediction was too pessimistic for big messages; they use more bandwidth than expected (Figure 25). However small messages, (<50KB) do not use the expected bandwidth and end up slower than in the prediction. This is also the range where we can observe the most irregularities due to the TCP protocol (Figure 21).

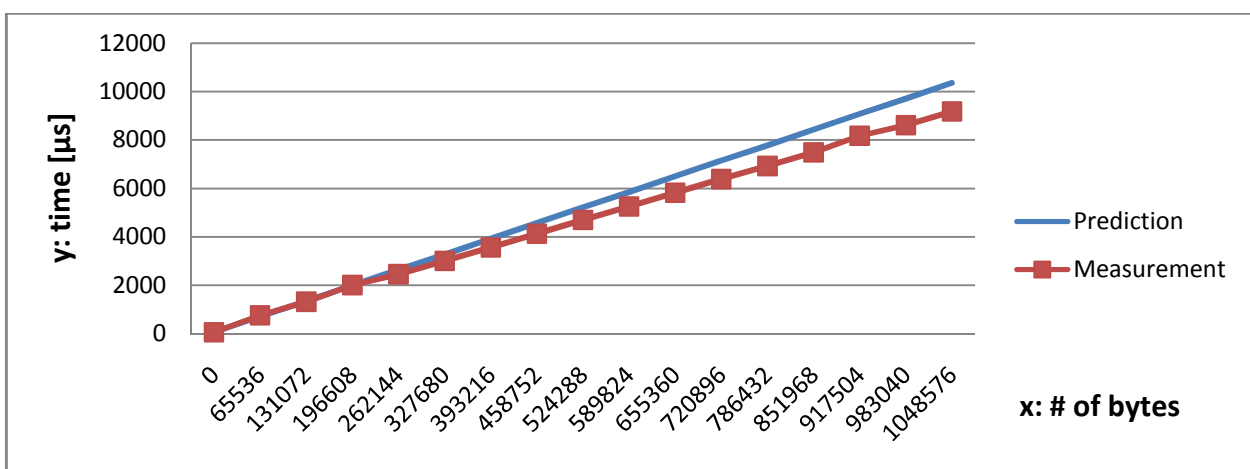


Figure 25 Global differences between prediction and measurement in OpenMPI sync

### 5.2.3.3. POP-C++ asynchronous

The prediction of this test scenario was too optimistic for big messages (Figure 26). Like mentioned in section 0 the estimated latency is too large and thus I used also a too large bandwidth to make the prediction.

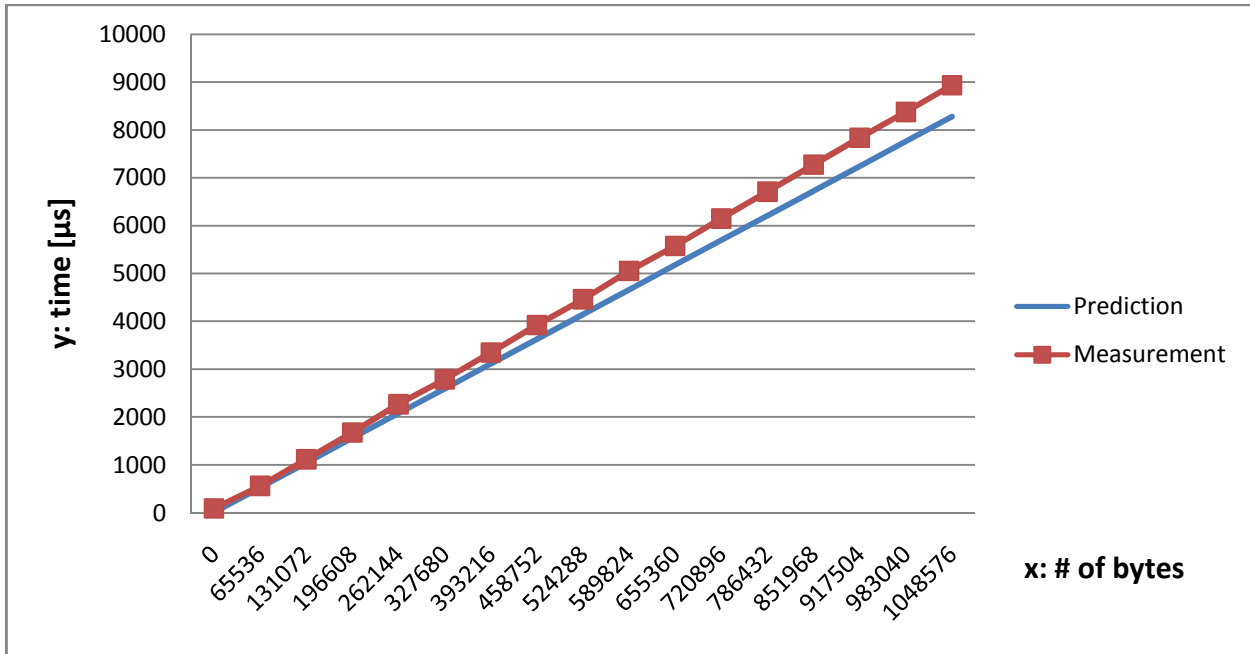


Figure 26 Global differences between prediction and measurement in POP-C++ async

### 5.2.3.4. POP-C++ synchronous

This test scenario shows a similar behavior as the synchronous OpenMPI program for messages smaller than 100KB. But unlike in that test, performances in terms of bandwidth are not increasing with the message size but are decreasing (Figure 27) and thus this scenario loses performances compared to all other scenarios (Figure 28).

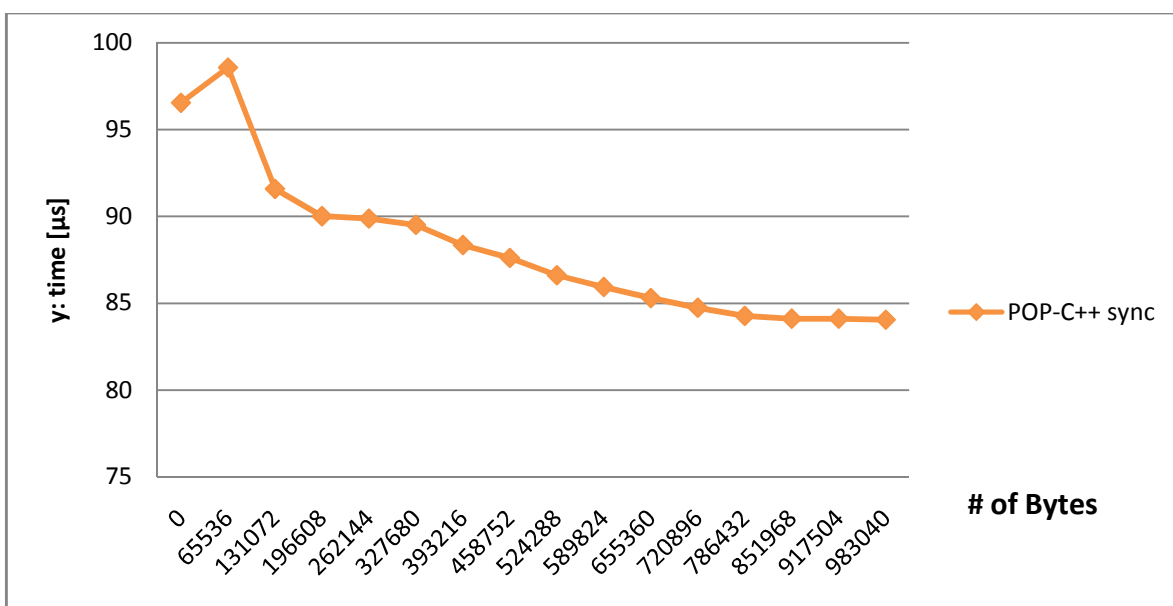


Figure 27 Bandwidth decrease in POP-C++ sync

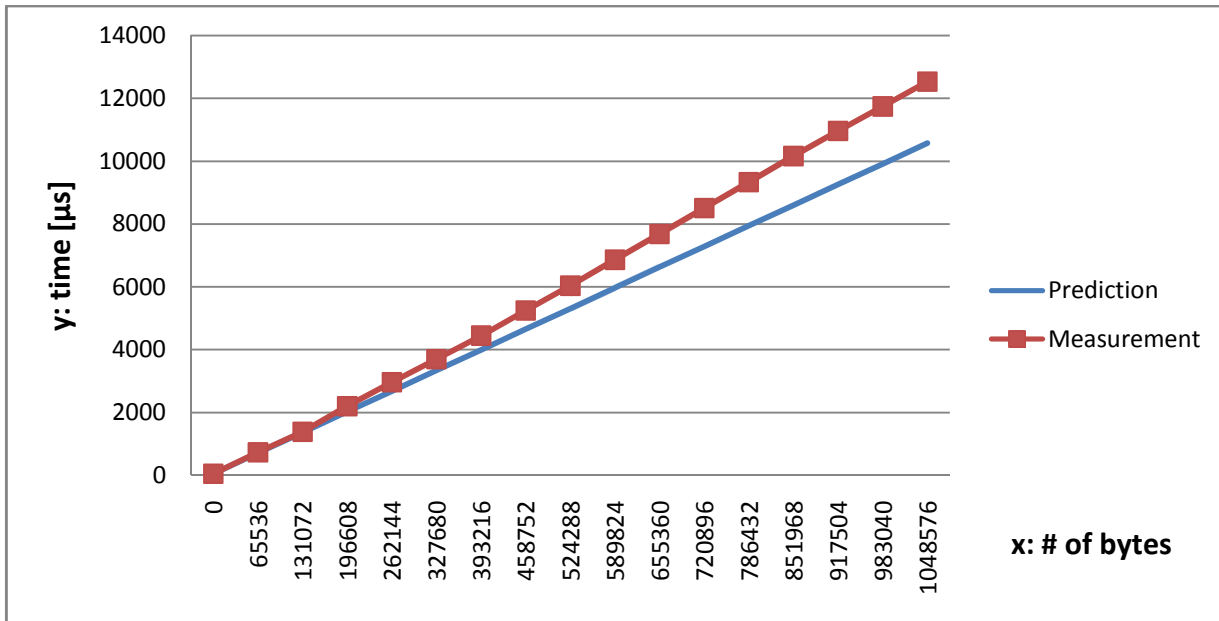


Figure 28 Global differences between prediction and measurement in POP-C++ sync

#### 5.2.4. Including unconcerned processes/objects

The test programs were implemented in a manner to verify if other processes/objects included in the program could disturb the point-to-point communication between the two concerned processes/objects. They only consume a message in each iteration but on the outside of the actual measurement. But as expected, this is not the case for all four test programs. These measurements showed the same performances as the ones with only 2 processes/objects.

## 6. Conclusion

This chapter contains an explanation of the encountered problems, the final result of the tests and a personal conclusion.

### 6.1. Encountered problems

#### 6.1.1. Starting the "SXXparoc" service on computing nodes

When the system administrator of the *Phoenix* cluster tried to startup the *SXXparoc* service on the computing nodes with an automatic startup script, it stayed stuck on every node. He had to start the service on each node manually. On a small cluster like phoenix that is not a big deal. But on bigger clusters with thousands of nodes it would be nearly impossible.

We managed to run the script without blocking by using the `-f` flag on the ssh command line. But another problem still persists. Once the script has finished, the ssh processes for every node still run in the background although they are supposed to disappear.

### 6.1.2. ssh: Connection refused

During the tests on the cluster, a SSH error showed up when the application is trying to create objects without the job manager but by using the object descriptor *od.url*. The problem was that POP-C++ by default uses RSH to use remote execution. To use SSH, the environment variable PAROC\_RSH has to be exported to the SSH location before starting the *SXXparoc* service.

For example with bash shell:

```
export PAROC_RSH=/usr/bin/ssh
```

### 6.1.3. Cluster access and disturbed measurements

The Phoenix cluster is used for benchmarking by other people of the computer science department at the UNM. This is why the time to access it is limited for everyone. When something went wrong during a measurement (Table 7) the time to wait for the next access was 1-3 days.

Size [Bytes]	Minimum [ $\mu$ s]	Average [ $\mu$ s]	Maximum [ $\mu$ s]
10240	177.5	202.04	101204.99
20480	256.9	301.93	104022.03
30720	353.93	358.32	101346.97
40960	451.09	498.99	103460.07
51200	537.04	589.89	102808
61440	629.54	710.83	103517.53
71680	781.54	799.1	20708.44
81920	877.02	886.66	1651.53
92160	954.03	1050.44	103341.1
102400	1049.4	1063.81	1914.02

Table 7 Disturbed measurement on the cluster

## 6.2. POP-C++ vs. MPI

The artificial ping-pong test showed that POP-C++ can reach the same performances as OpenMPI for data sending. As the latency is almost the double for POP-C++, it is slower for messages smaller than 1KB where latency has more weight on the result. As already discussed, the POP-C++ test program seems to include more CPU cycles into the measurement than the MPI test program, which falsifies the result for latency and small messages a lot. When the message size gets bigger, POP-C++ is even faster than OpenMPI. But, as mentioned earlier, OpenMPI is not optimally implemented to use TCP and Ethernet. This is because faster local networks like InfiniBand or Myrinet are more commonly used on HPC-clusters and OpenMPI is more sophisticated for those technologies.

The table below (Table 8) summarizes performance comparisons between POP-C++ and MPI in all ranges of data length to pass from one process/object to another. In my tests, POP-C++ was slower for messages <10KB and shows better performances than OpenMPI for messages >10KB.

Range [Bytes]	Perf. MPI [%]	Perf. POP-C++ [%]
1-1024	100	68.27680637
1024-10240	100	96.0923761
10240-102400	100	107.9309088
131072-1048576	100	109.8135363

Table 8 Performances of POP-C++ vs. MPI

A delicate point in POP-C++ is the use of output parameters. As we could see in the synchronous tests, the performances decreased by increasing the parameter size (Figure 27). The main difference of the two POP-C++ test programs resides in the fact that the called method in the synchronous test returns a value (output parameter). So the reason for the performance decrease has to be in the mechanism to return a value to the caller.

---

# Collective Communication in POP-C++

---

## 1. Introduction {common}

This part of the project consists in adding collective communication to POP-C++. POP-C++ already provides a feature to use collective communication by using MPI underneath (see [4]). What we want to do is adding collective communication independent from MPI. Collective communication means that more than 2 entities are involved in a communication process (point to multipoint, multipoint to multipoint).

My colleague Frédéric Barras and I took the decision that I try to embed collective communication by modifying the POP-C++ parser. He will design and implement a version without modifying the parser, but by adding a library providing collective communication functions.

The analysis of collective communication and the POP-C++ is presented in the next chapter.

## 2. Analysis

This chapter analyzes how to embed collective communication into the existing runtime of POP-C++. For a better understanding, MPI collective communication is analyzed first.

### 2.1. Collective communication in MPI

Understanding point-to-point communication in MPI is recommended before reading this section (see section 2.4 of the first part on page 10).

In parallel computing we can find communication models where more than 2 processes are involved at the same time. The MPI standard specifies a collection of functions which are defined under the term *Collective communication* and which correspond to such communication models. Often, one process is distinguished from the others (by its rank) and is commonly called *root* (for 1-N / N-1 functions). But in some functions, every process does the same (N-N functions).

Method	Description	Visualization
<b>Broadcast</b>	The <i>root</i> process sends identical data to any other process of the same group.	<p>Figure 29 MPI broadcast [7]</p>
<b>Scatter</b>	<i>root</i> sends different data of same size to any other process of the same group.	<p>Figure 30 MPI Scatter [7]</p>
<b>Gather</b>	<i>root</i> gathers the data from all involved processes and places them, sorted by rank, into its reception buffer.	<p>Figure 31 MPI Gather [7]</p>
<b>Reduce</b>	The main idea is to do the same as the <i>Gather</i> operation. But before storing every single received piece into the reception buffer, a Boolean or arithmetic operation is performed to combine the data. And only the result of this operation is stored in the reception buffer of <i>root</i> .	<p>Figure 32 MPI Reduce [7]</p>
<b>AllGather</b>	No <i>root</i> process is present in this function. It corresponds to a multi broadcast where every process sends its data to every other process of the group. The final reception buffer of those will be identical.	<p>Figure 33 MPI Allgather [7]</p>



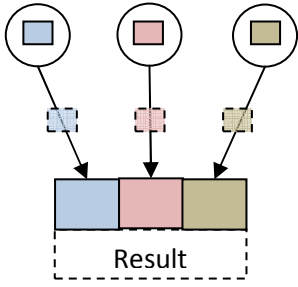
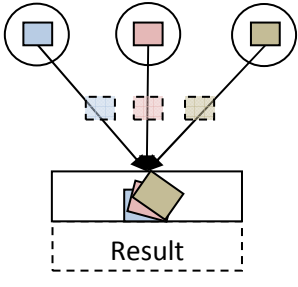
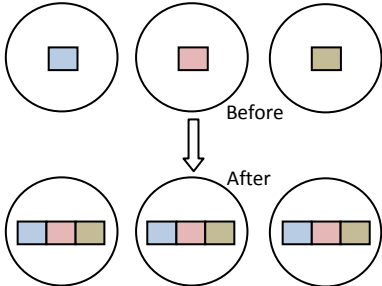
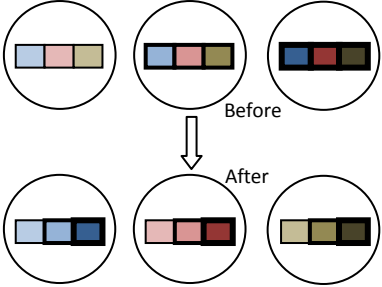
<p><b>All-to-All</b></p>	<p>This is an N-N function and thus doesn't contain a <i>root</i> process. Process <i>i</i> sends the <i>k</i>th block of its send buffer to process <i>k</i>, which stores it in the <i>i</i>th block of its reception buffer.</p>	<p>Figure 34 MPI All-to-All [7]</p>
<p><b>AllReduce</b></p>	<p>N-N function with no <i>root</i> process. Multi broadcast with following reduction operation.</p>	<p>Figure 35 MPI Allreduce [7]</p>

Table 9 Explanation and visualization of MPI collective communication functions

## 2.2. Collective communication in POP-C++

In the current version of POP-C++ it is possible to couple MPI code in a POP-C++ program by using a special template class. The disadvantage in this approach is that we partially lose the object oriented paradigm and introduce the message passing paradigm instead. Table 10 explains how collective communication should be implemented to keep the OO paradigm. Circles represent parallel objects and rectangles represent a data element.

Method	Description	Visualization
<p><b>Broadcast</b></p>	<p>The program invokes a method on the group by passing data in parameters. This data is communicated to all members of the group and is identical for every one of them.</p>	<p>Figure 36 POP-C++ Broadcast</p>
<p><b>Scatter</b></p>	<p>The program invokes a method on the group by passing lists of data elements in parameters. Each parameter is a list of data elements. The <i>i</i>th element of this list is sent to the <i>i</i>th object in the group, thus every one receives potentially different data.</p>	<p>Figure 37 POP-C++ Scatter</p>

<p><b>Gather</b></p>	<p>For the invoked method on the group every member returns one data element. The final result on the caller's side is a list of data elements.</p>	 <p>Figure 38 POP-C++ Gather</p>
<p><b>Reduce</b></p>	<p>The main idea is to do the same as the <i>Gather</i> operation. But instead of storing every data element separately, only the result of an arithmetic or logic operation of these elements is stored.</p>	 <p>Figure 39 POP-C++ Reduce</p>
<p><b>AllGather</b></p>	<p>This operation corresponds to a multi broadcast where every object of the group calls a method on every other member <math>i</math> by passing the <math>i</math>th data element of its list as a parameter. After the operation, all objects contain the same list of data elements (same order). The data element passed by object <math>i</math> is at the <math>i</math>th position.</p>	 <p>Figure 40 POP-C++ Allgather</p>
<p><b>All-to-All</b></p>	<p>Object <math>i</math> invokes a method on every object <math>k</math> by passing the <math>k</math>th data element of its list as a parameter. Object <math>k</math> stores the received data element at the <math>i</math>th position of its list.</p>	 <p>Figure 41 POP-C++ All-to-all</p>

<b>AllReduce</b>	This operation corresponds to a multi broadcast with following reduction operation.	<p>Figure 42 POP-C++ Allreduce</p>
------------------	---	------------------------------------

Table 10 Explanation and visualization of POP-C++ collective communication functions

### 2.3. The POP-C++ parser

The POP-C++ parser is built with *flex* and *bison*. It is implemented as a finite state machine (for more details on *flex* and *bison*, consult the online manuel [9]). *bison* is designated to process a file (see example below) describing the grammar which the parsed files are supposed to respect. For example, a class in C++ is constituted of its head declaration followed by its member list (attributes, methods) between braces. Head declaration and member list declarations are described in another rule which may contain again other sub rules. All these rules together are finally describing the finite state machine. Every rule can start several actions which are coded in C/C++. After having processed this grammar file with *bison*, a C/C++ source file is created which can parse the input following this grammar.

```
...
%token PARCLASS_KEYWORD CLASS_KEYWORD
...
class_declaration: class_head '{' member_list '}' ';'
{
    currentClass=NULL;
    insideClass=false;
}
;

class_head: class_key pure_class_decl base_spec
;

class_key: PARCLASS_KEYWORD ID
{
    insideClass=true;
    Class t;
    currentClass=t;
}
;

member_list: /*empty*/
| member_declaration member_list
| access_specifier ':' member_list
;
...
```



*flex* processes another file (see code below) to create a lexical analyzer (coded in C/C++) recognizing tokens in the parsed files.

```
...  
parclass { return PARCLASS_KEYWORD; };  
  
class { return CLASS_KEYWORD; };  
...
```

The lexical analyzer provides tokens to the parser which performs a certain action corresponding to the delivered token in the current state and then updates the state machine. In the example above, the token `PARCLASS_KEYWORD` is returned to the parser whenever the lexical analyzer encounters “parclass” in the input. Processing this file with *flex* means creating a C/C++ source file doing the work described above.

Once these C/C++ source files are created, they are compiled and linked together into the executable parser. How POP-C++ source code is processed by this parser is illustrated in the figure below.

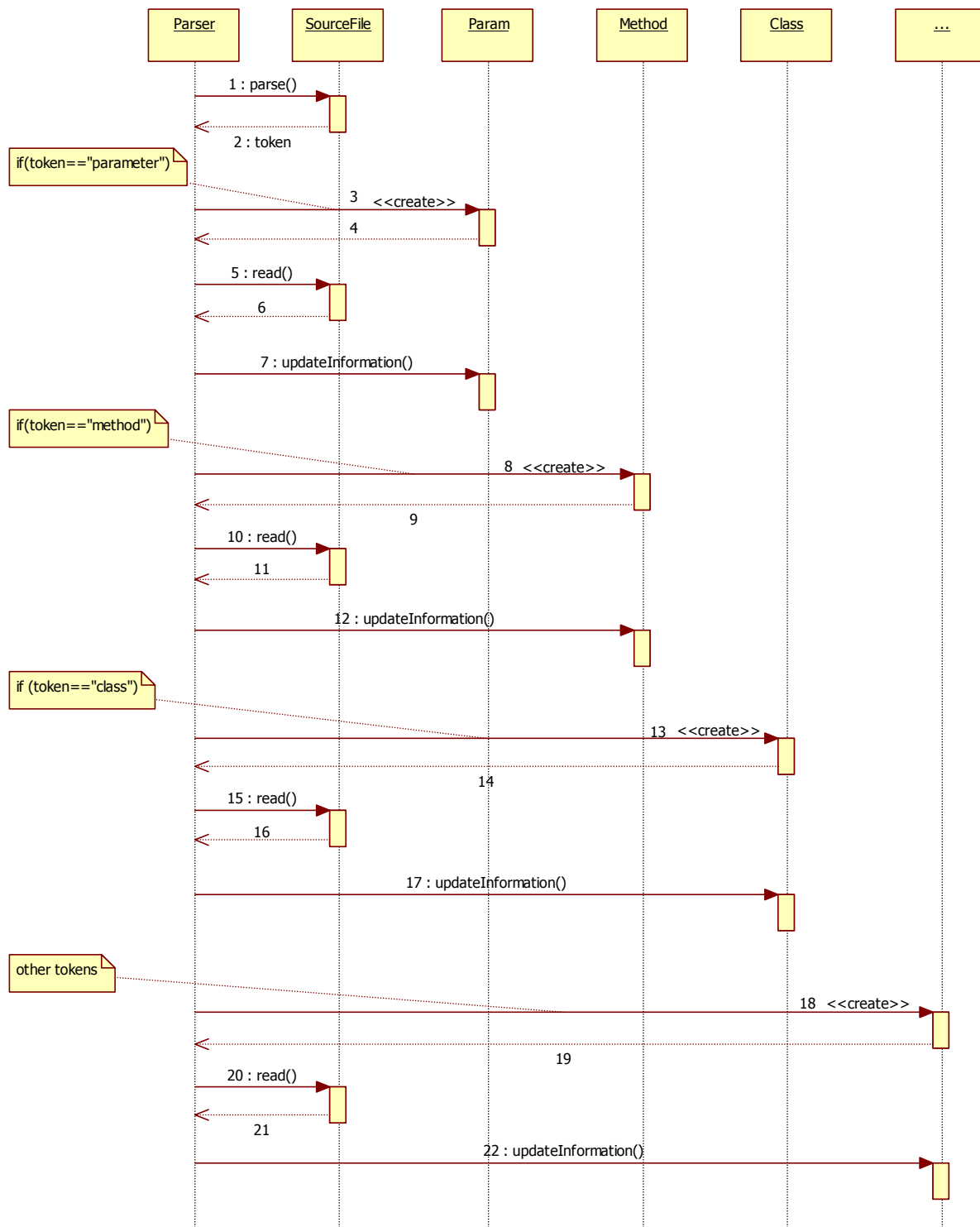


Figure 43 Vastly simplified functionality of the POP-C++ parser

When the POP-C++ parser is processing source files, it creates objects in memory representing parameters, methods, classes etc. encountered during parsing. This is necessary to remember important information (such as the name of a class, the type of a parameter, the return type of a method, ...) which is used later on to generate corresponding C++ code. This information is updated continuously during the parsing process. The order of what's created is not necessarily the one showed in Figure 43.

### 3. Design

The goal of this conceptual method of adding collective communication to POP-C++ is to consider the object oriented programming paradigm the most possible. The design presented in the following sections is unfortunately not implementable as a generic library (independent of the object type). It is necessary to generate code dynamically at the compilation of an application. A group parser in charge of this task (see section 3.4) must be included in the POP-C++ compilation process. But before discussing the group parser, information about how collective communication is built into POP-C++ is described, assuming that the necessary source code exists.

#### 3.1. Group representation

It is necessary to specify what a group is in POP-C++ to define which objects a collective method invocation will affect.

A group of objects is represented by a container knowing all the objects which are part of the group. Let's call this container *POPGroup*. Every member of a *POPGroup* is uniquely identified by its rank and all objects are of the same type. Figure 44 shows the operations which are possible on a *POPGroup* container. For explanations of every use case, read the *Use case description sheets* located on the CD coming with this report.

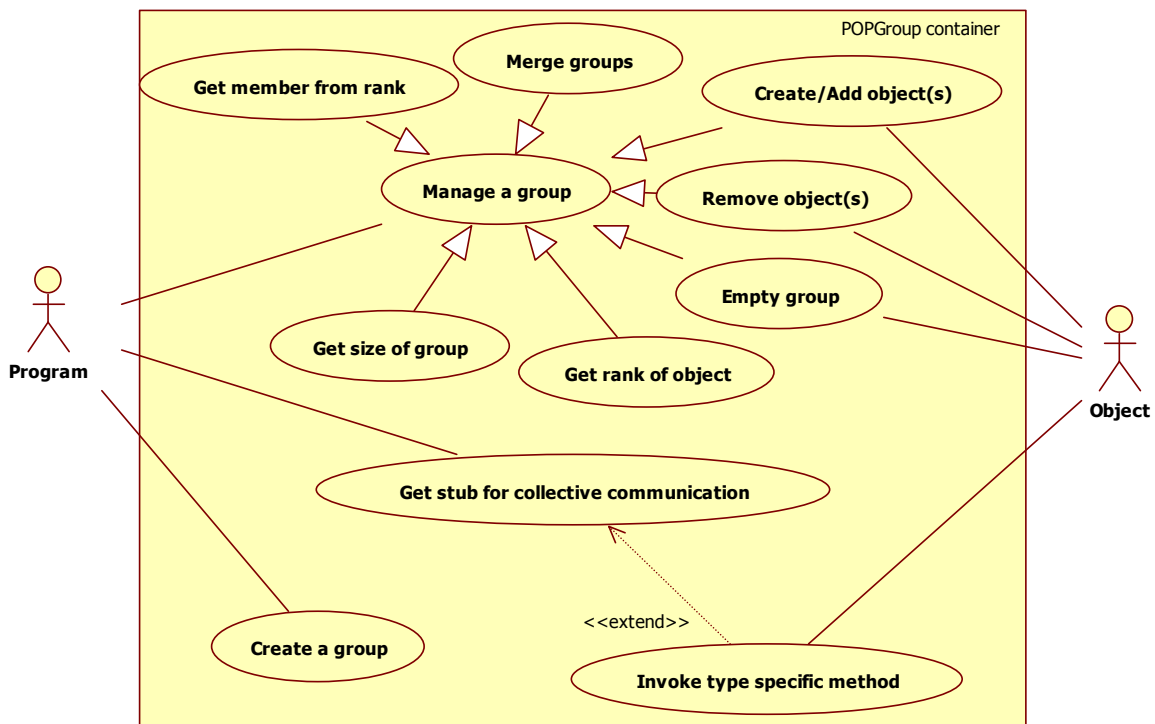


Figure 44 Use case diagram to handle a group

As we can see, the container provides different functions to manage a group such as adding or removing objects. To use collective communication on the group, the program has to get the stub of this type of group (see pseudo-code below). The stub is to invoke type specific methods on the entire group. This mechanism is used to explicitly make the difference between group management functions (add, remove ...) and collective communication (type specific method).

```
POPGroup<Test> myTestGroup;           // Create group of type Test
myTestGroup.add(objects);             // Add objects to the group
myTestGroup.comm().helloWorld();      // helloWorld() is specified in class Test
```

### 3.2. Collective communication operations

The collective communication library of POP-C++ provides four basic operations which use all point to multipoint communication:

- broadcast
- scatter
- gather
- reduce

**Broadcast:** The same data is passed to every group member. In this example, the group invokes *setMethod(3)* on every member.

```
int param = 3;
myGroup.comm().setMethod(param);      // Invokes setMethod on every member with parameter param
```

**Scatter:** Each remote object receives different data. The programmer specifies an array of parameters which is passed to the group by indicating the array's size.

```
int nb = 4;                           // Size of parameter array
int params[nb] = {1,2,3,4};           // Parameter array
myGroup.comm().setMethod(params, nb); // Invokes setMethod on first nb members by passing
// one parameter from params array according to rank
```

In this example, the group would invoke *setMethod(1)* on member with rank 0, *setMethod(2)* on member with rank 1 and so on. The group uses a circular procedure to invoke methods on members to handle the case where the number of members doesn't correspond to the size of the passed array. If the array size is smaller than the number of members, not every member will be called. If the array size is greater, some members will be called multiple times.

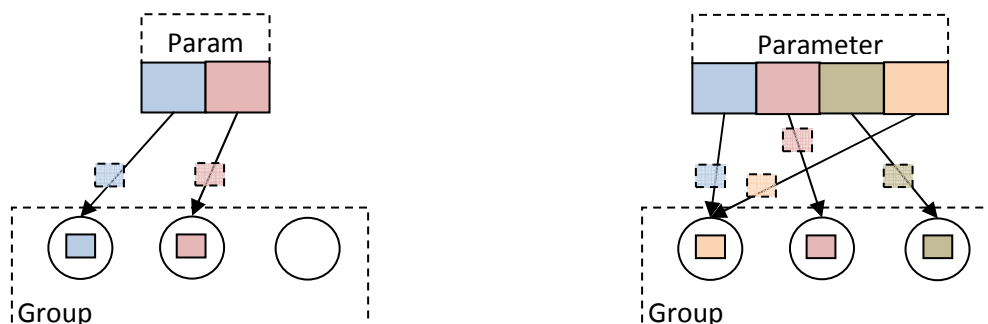


Figure 45 Scatter operation in case of inequality of parameter size and group size



**Gather:** This method returns an array of results which's size corresponds to the number of group members. The results are ordered according to the rank of the members.

```
int res = new int[myGroup.getSize()];           // Create the array to store results
myGroup.comm().getMethod(res);                 // Invoke getMethod on all group members
```

In this case, *res[0]* will contain the result returned by the member with rank 0.

**Reduce:** After implicitly doing a *Gather*, a reduce operation is executed on the array of results (min, max, etc...) before returning the result.

```
int res;                                         // Variable to store result
res=myGroup.comm().getMethod(POPGroup<type>::_MAX); // Invoke getMethod on all group members and
                                                    // performs reduce operation MAX
```

In this example, *res* would contain the maximum value of the returned results of all members.

Figure 46 illustrates the above described operations in a more generic way.



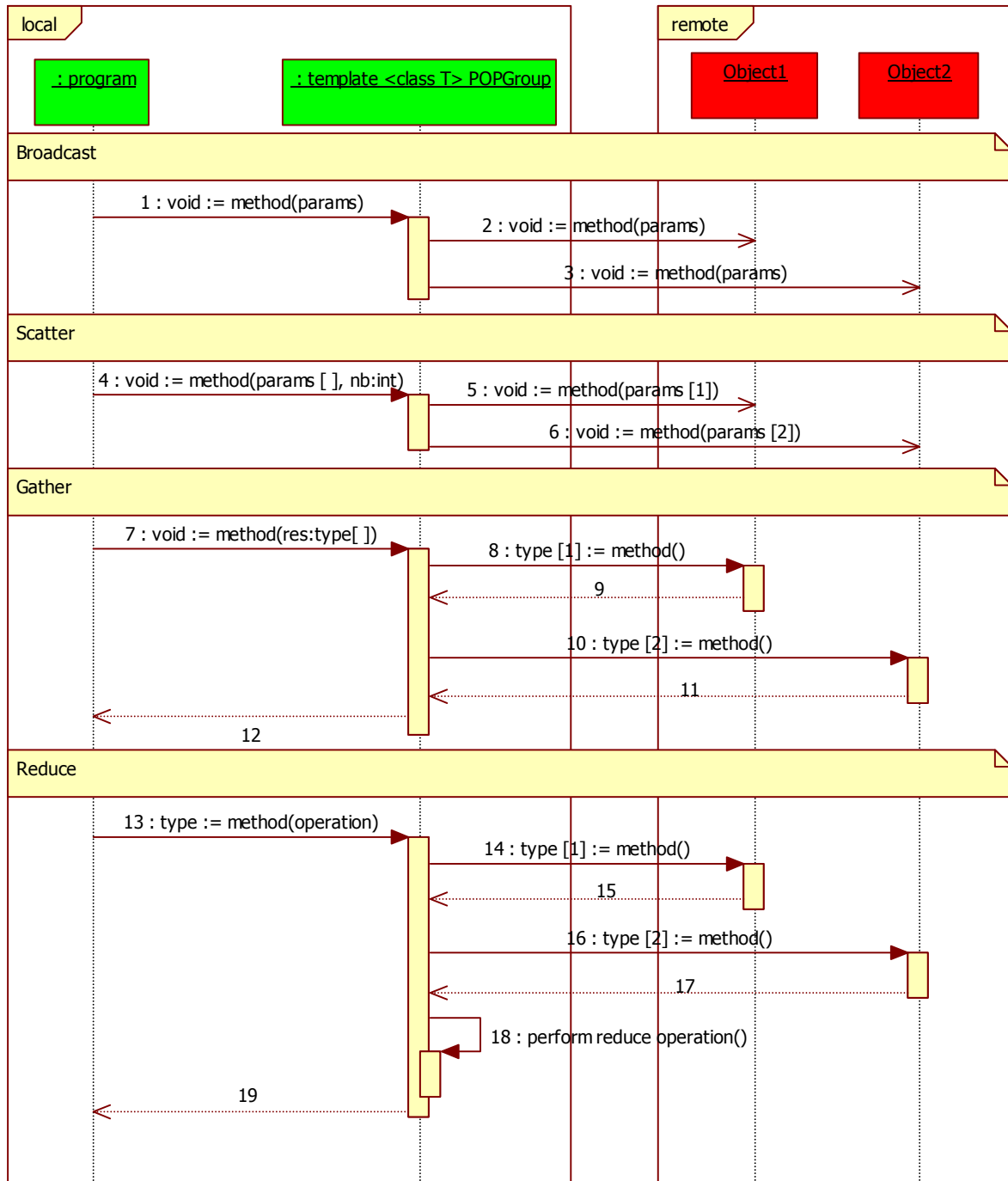


Figure 46 Sequence diagram of use case, "Call type specific method"

### 3.3. Architecture

Figure 47 illustrates the architecture of the collective communication library. The *template* [8] mechanism of C++ is used to implement the generic management methods. Another class contains type specific methods used for collective communication. The classes *T*, *stubT* and the method *comm()* in the *POPGroup* template returning a reference to a *stubT* instance are dynamic components in this architecture. They depend on whatever class *T* contains, where *T* can be any imaginable C++ type. These components must be generated during the POP-C++ compilation process.

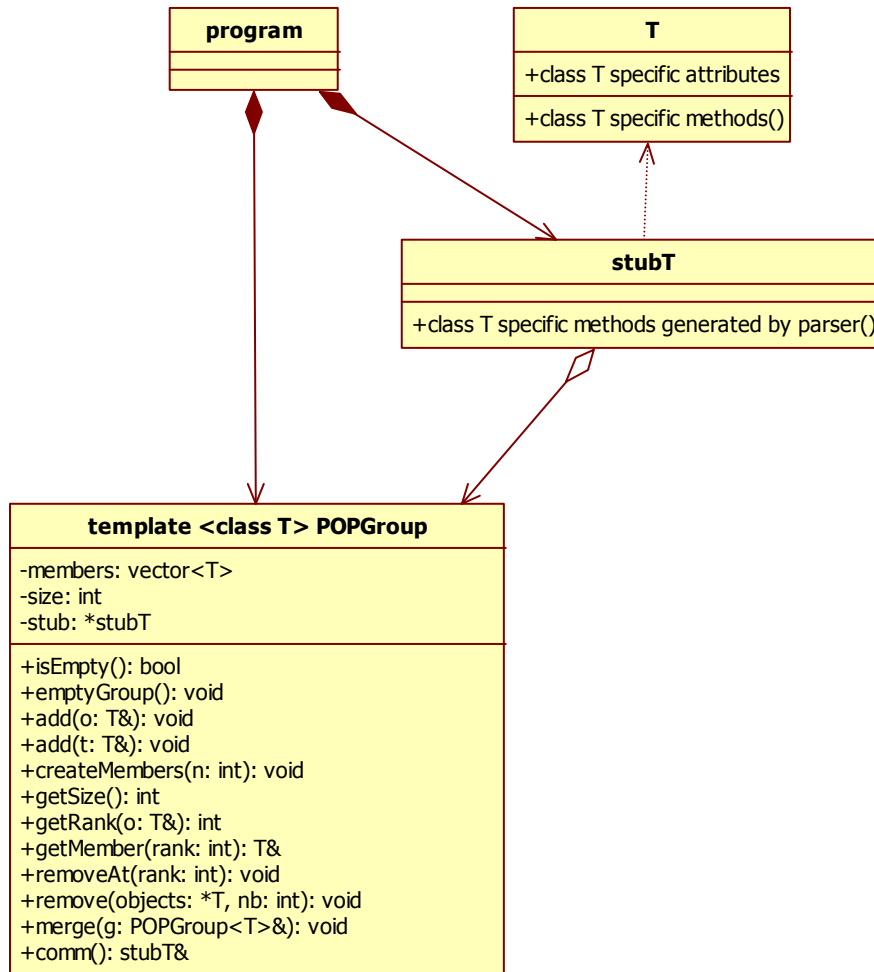


Figure 47 Architecture of the collective communication library

The main reason why group management methods are separated from the type specific methods is to avoid method name conflicts. If they were placed in the same class it could be possible that the class wouldn't be accepted by the compiler. For example if code is generated from a class *T* containing the method *bool isEmpty()*, the resulting class would contain two methods with exactly the same signature. Separating them also decreases the risk of confusion for the programmer or somebody who reads the source code using this library when method names are similar in the stub class and the *POPGroup* container.

A simple program using collective communication would be implemented like the following example:

**The main program:** includes all needed libraries plus the *POPGroup.h* file which represents the group container.

```
#include "example.ph"
#include "POPGroup.h"

int main(int argc, char **argv) {
    POPGroup<Example> myGroup;
    // POP-C++ program using collective communication (or not)
}
```

**The stub class:** This class contains all the type specific methods. When a group container is created, an object of the corresponding stub class is also instantiated. A reference to this instance is returned when the main program calls the *comm()* method on the group container.

```
#include <vector>
using namespace std;

class _parocGroupExample
{
    vector<Example*> *members;
    int *size;

public:
    _parocGroupExample(vector<Example*> *mem, int *sz) {
        members=mem;
        size=sz;
    }

    //list of type specific methods for broadcast, scatter, gather and reduce
    ...
};
```

**The *POPGroup* container:** Contains group management functions. To define the container for *Example* objects the template class *POPGroup* is specialized. As there is no inheritance in templates, all the class members must be redefined for every class used for collective communication. This is why the base template doesn't contain any members at all.

```
#include <vector>
using namespace std;

template <class U>
class POPGroup {
};

#include "_parocGroupExample.h"

template<>
class POPGroup<Example> {

    vector<Example*> members;
    vector<Example*>::iterator it;
    int size;
    _parocGroupExample *stub;

public:
    const static int _MAX=0;
    const static int _MIN=1;
    const static int _OR=2;

    POPGroup() {
        size=0;
        stub = new _parocGroupInteger(&members, &size);
    }

    bool isEmpty() { }
    void emptyGroup() { }
    void add(Example &o) { }
```

```
void add(Example *o, int nb) { }
void createMembers(int nb) { }
void removeAt(int rank) { }
void remove(Example *o, int nb) { }
int getRank(Example &o) { }
int getSize() const { }
Example &getMember(int rank) { }
void merge(POPGroup<Example> &g) { }
  _parocGroupExample &comm() { }
};
```

The only thing the programmer has to take care about is to include the *POPGroup.h* file in the main program.

### 3.4. Creating a “group parser”

The POP-C++ compiler *parocc* executes 3 different programs: the C++ preprocessor, the POP-C++ parser and the C/C++ compiler. They are executed in the order they were mentioned. As we’ve seen in section 3.3 the programmer has to include the *POPGroup.h* file to compile the application using collective communication. The problem when creating this file directly in the POP-C++ parser is that it won’t exist at the moment when the C++ preprocessor does its work. In consequence, the file generation has to be made before.

Considering the issue mentioned above, an additional program which accomplishes the requested tasks must be implemented. Let’s call it *group parser*. It generates the files containing the source code for the *POPGroup* container(s) and the stub classes. This source code has to be written in specific files respecting the following convention:

- The *POPGroup* container for every class using collective communication in the application is written to a file called **POPGroup.h**.
- Source code for the stub class for every class using collective communication in the application is written to a file called **\_parocGroupT.h** when *T* has to be replaced by the name of the class

Even if the group parser can be seen as an independent program it can be invoked implicitly when compiling a POP-C++ application with *parocc*. The fact that the application is using collective communication can be told to the POP-C++ compiler by adding the optional *-group* command line argument.

```
parocc -group -c example.ph example.cc
```

The above command generates automatically the collective communication source code for all class specifications in the file *example.ph*. Consider the example below to see how the group parser translates a header file into a stub class:

```
parclass Example {
public:
  void method_0();
  void method_1(int n);
  int method_2();
  int method_3(int n);
};
```

The group parser generates the code below from the input file above.

```
class _parocGroupExample {
...
public:
//generated methods for method_0
    void method_0();                // Implements broadcast

//generated methods for method_1
    void method_1(int n);           // Implements broadcast
    void method_1(int *n, int nb); // Implements scatter

//generated methods for method_2
    void method_2(int res[]);      // Implements gather
    int method_2(int op);         // Implements reduce

//generated methods of method_3
    void method_3(int n, int res[]); // Implements broadcast and gather
    void method_3(int *n, int nb, int res[]); // Implements scatter and gather
    int method_3(int n, int op);    // Implements broadcast and reduce
    int method_3(int *n, int nb, int op); // Implements scatter and reduce
};
```

- Normal parameter
- Array of parameters for *Scatter* operation
- Output parameter for *Gather* operation
- Type of *Reduce* operation to perform
- Size of parameter list for scatter operation

If the programmer wants to collect results in an array by doing a gather operation, the array has to be passed as a parameter to the called method on the group (output parameter). To perform a reduce operation on this array before returning the result, an additional parameter is needed to indicate what operation to perform (max, min, etc.). The return and parameter types are recognized during the parsing process of course and thus not limited to a range of specific types.

Only public methods are translated into the stub. All the other methods are not considered during the parsing process, because they are not supposed to be called from the outside of the class.



Table 11 recapitulates the generation of stub methods having a *.ph* as the input.

Method in <i>.ph</i> file	Generated methods in stub
<code>void method()</code>	<pre>/* broadcast : call the method on every member */ <b>void method()</b></pre>
<code>void method(int n, int m, ...)</code>	<pre>/* broadcast : call the method with same parameters (n,m,...) on every member */ <b>void method(int n, int m, ...)</b>  /* scatter : call the method with different parameters (n[i],m[i],...) on every member by specifying the size of the array(s) (size). Default is size of group. */ <b>void method(int *n, int *m, ..., int size=getSize())</b></pre>
<code>int method()</code>	<pre>/* gather : call the method on every member. Gather the return values in an array (res[])* <b>void method(int res[])</b>  /* reduce : call the method on every member. Return result of reduce operation passed as a parameter (op) */ <b>int method(char *op)</b></pre>
<code>int method(int n, int m, ...)</code>	<pre>/* broadcast and gather : call the method with same parameters (n,m,...) on every member. Gather results in an array (res[]) */ <b>void method(int n, int m, ..., int res[])</b>  /* scatter and gather: call the method with different parameters (n[i],m[i],...) on every member by specifying the size of the array(s) (size). Default is size of group. Gather the return values in an array (res[]) */ <b>void method(int *n, int *m, ..., int res[], int size=getSize())</b>  /* <b>broadcast</b> and reduce : call the method with same parameters (n,m,...) on every member. Return result of reduce operation passed as a parameter (op) */ <b>int method(int n, int m, ..., char *op)</b>  /* scatter and reduce : call the method with different parameters (n[i],m[i],...) on every member by specifying the size of the array(s) (size). Return result of reduce operation passed as a parameter (op) */ <b>int method(int *n, int *m, ..., int nb=getSize(), char *op)</b></pre>

Table 11 How the POP-C++ parser generates methods from *.ph* file

### 3.5. Handling a group of remote objects for N-N communication

Table 10 in section 2.2 also presents N-N functions (All-to-all, Allgather, Allreduce) in POP-C++. The difference compared to 1-N/N-1 functions is mainly that there is no entity which initiates a method call on all members or reassembles the return values. For 1-N/N-1 communication, this entity would be the main program using the group library. This makes it difficult to find a common denominator for N-N functions and object oriented programming. It would need major changes in the POP-C++ runtime to handle this type of collective communication properly and efficient. Therefore, this project contains design and implementation for 1-N/N-1 functions only.

## 4. Implementation

This chapter describes how the group parser has been implemented to generate source code files needed to use collective communication in a POP-C++ application. In the design section, no difference has been made between parallel classes and usual C++ classes. The mechanism is designed for both of them. In the implementation, only parclasses are mentioned because the group parser doesn't take into consideration C++ classes (see also section 6).

### 4.1. Code models

Code model files are used by the group parser to generate container(s) and stub classes by reading these files and writing their content to the output file *POPGroup.h*.

The first code model (section 3.1) specifies the base template class *POPGroup*, which is in fact an empty class. It is only defined to use a common syntax for any type of created groups:

```
POPGroup<Example> myGroupExample;  
POPGroup<Test>    myGroupTest;
```

Another code model (section 3.2) specifies the specialized template class *POPGroup<T>* for every type. The special character *T* is replaced by the name of the current parclass when writing to the output file (*POPGroup.h*) to generate the final *POPGroup* container for this parclass which allows the programmer to invoke management function on the group.

```
Example o1;  
myGroupExample.add(o1);
```

This code model also contains the implementation of the group management functions.

### 4.2. Group parser

The goal of this parser is parsing files to find parclasses and generate a stub class corresponding to the public methods it encounters inside these parclasses. It is built on the same grammar as the POP-C++ parser and uses the same technologies to generate the C/C++ source files (*flex and bison*).

Essentially, the parser does the following work (see grammar extract below): it searches the input for the keyword *parclass* followed by the parclass' name. When this situation arrives, a new *Class* object is created in memory, representing the parclass by its name and id. After this, the parser calls a method to generate the *POPGroup* container for this class. All that is needed to generate this container is the name of the parclass. One or both of the following scenarios can happen:

- The current parclass is the first encountered in the parsing process ( $id=0$ ). Generate the base template class *POPGroup* by using the corresponding code model in the output file *POPGroup.h*. This file is created if it doesn't exist and replaced if it does.
- The current parclass is the first or not the first encountered in the parsing process ( $id \geq 0$ ). Generate the specialized template class *POPGroup<T>* by using the corresponding code model and write to the output file *POPGroup.h*. During the writing process, *T* is replaced by the parclass' name. The generated code is appended to the output file which has been created by the scenario above.

```
/*
Parallel class declaration
*/
class_declaration: class_head '{' member_list '}' ';'
{
    currentClass->~Class();
    currentClass=NULL;
    insideClass=false;
}
;

class_head: class_key pure_class_decl base_spec
{
    accessmodifier=PUBLIC;
}
;

class_key: PARCLASS_KEYWORD ID
{
    insideClass=true;
    char *cname=GetToken($2);
    Class *t=new Class(cname, crtClassID++);
    currentClass=t;
    currentClass->GenerateGroupTpl();
}
;
```

The first part of the file containing the stub class (*\_parocGroupExample.h*) for this parclass is also generated at this point because it depends only on its name (in this case “Example”).

```
#include <vector>
using namespace std;

class _parocGroupExample
{
    vector<Example*> *members;
    int *size;

public:
    _parocGroupExample(vector<Example*> *mem, int *sz) {
        members=mem;
        size=sz;
    }
}
```

After this step, the parser is in the state of being in a parclass definition. It is now expecting the “{” character followed by a list of members (attributes and methods). This member list is described as follows in the grammar (simplified):



```
/*
Parallel class member declaration
*/
member_list: /*empty*/
| member_declaration member_list
| access_specifier ':' member_list
;
member_declaration: function_definition ';'
| attribute_definition ';'
;
/*
Method declaration
*/
function_definition: constructor_definition
| destructor_definition
| method_definition pure_virtual_decl
{ method->GenerateClient();
}
;
pure_virtual_decl: /*empty*/
| '=' INTEGER
{ method->SetPureVirtual();
}
;
method_definition: decl_specifier pointer_specifier ref_specifier function_name '('
argument_declaration ')'
| fct_specifier decl_specifier pointer_specifier ref_specifier function_name '('
argument_declaration ')'
| '[' marshal_opt_list ']' decl_specifier pointer_specifier ref_specifier function_name '('
argument_declaration ')'
| fct_specifier '[' marshal_opt_list ']' decl_specifier pointer_specifier ref_specifier
function_name '(' argument_declaration ')'
;
function_name: ID
{
method=new Method(currentClass,accessmodifier);
currentClass->AddMember(method);
strcpy(method->name,GetToken($1));
returntype=currenttype;
currenttype=NULL;
}
;
/*
METHOD ARGUMENT DECLARATIONS
*/
argument_declaration: /*empty*/
| argument_list;

argument_list: arg_declaration
| arg_declaration ',' argument_list
;

arg_declaration: marshal_decl cv_qualifier decl_specifier pointer_specifier ref_specifier
argument_name array_declarator arg_default_value
{
Param *t=method->AddNewParam();
}
;
```

Attributes are ignored by the group parser. The only interesting members are public, non pure virtual methods. Once the method name is known, a new *Method* object is created in memory, representing the method by its name and the parclass it belongs to. When all the parameters have been parsed and added to the list of parameters in the *Method* object, all information is gathered to generate the corresponding collective method(s) (see Table 11) in the stub class (`_parocGroupT.h` file). At the moment when all members have been parsed, the only object still in memory is the current *Class* object. By deleting it, the file generation for this parclass completes by adding the terminating “};” characters to the end of the output file.

#### 4.2.1. Including the group parser in the POP-C++ compilation process

As described in section 3.4 the group parser should be called implicitly during the compilation of a POP-C++ application when the optional `-group` command line argument is specified. When this is the case, *parocc* scans the command line for files with the extension `.ph` and collects their names in an array. After this it invokes the group parser followed by the list of `.ph` file names stored in the array. Every `.ph` file may contain multiple parclass declarations. All this parclasses are parsed to generate the corresponding containers and stubs (Figure 48).

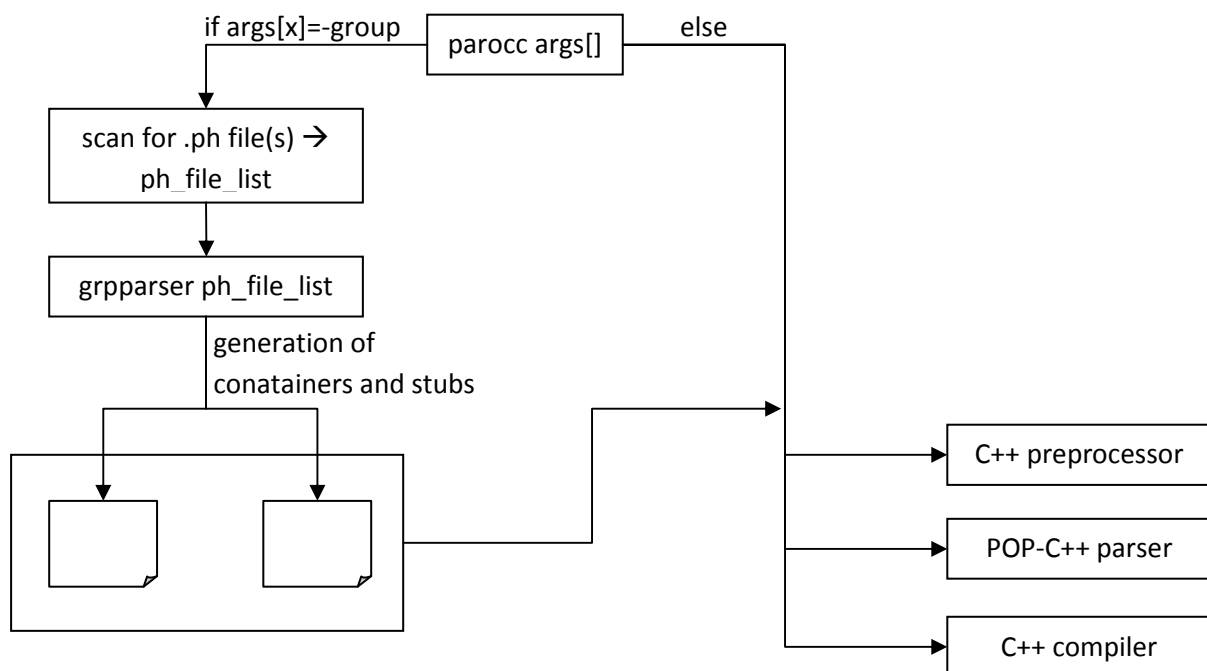


Figure 48 Integration of the group parser in the POP-C++ compilation process

Afterwards the normal POP-C++ compilation process takes place where *parocc* calls the C++ preprocessor, then the POP-C++ parser and at the end the C++ compiler.

### 4.3. Where to find the source code for the group parser?

The group parser is entirely integrated in a modified distribution of POP-C++ version 1.2. This distribution contains the usual POP-C++ runtime plus all the necessary sources to build the group parser. All these additional sources have been added with the GNU development tools *autoconf* and *automake* to keep the installation process identical to the unmodified distribution.

Three files can be in the *popc-1.2/include* directory:

- *parocGroup.h*: code model (section 3.1)
- *parocGroupSpec.h*: code model (section 3.2)
- *RankException.h*: Defines a class to handle the rank out of bounds exception in group management functions (section 3.3)

The *popc-1.2/grpparser* directory contains all the files that are necessary to create the group parser executable.

Two test example programs using collective communication can be found in the *popc-1.2/test* directory.

## 4.4. Limitations and recommendations

### 4.4.1. Installing the distribution

During the installation process of the distribution you have to build libraries, objects etc. from the source code with *make*. This step also compiles the test programs of the distribution which use the group parser to generate needed files. When another version of POP-C++ is already installed the environment variable **PAROC\_LOCATION** may be defined in the current context. This leads to a compilation error in the *make* process, because the group parser cannot be found at this location. It is therefore recommended to install this distribution in a context where this environment variable is not defined.

### 4.4.2. Split application in logical parts

To avoid compilation problems, a POP-C++ application using collective communication should be split in at least 3 parts. One part is the main program including *POPGroup.h*. The *.ph* file(s) containing the parclass declaration(s) represent the second part. Part 3 consists in the implementation of the methods defined in the *.ph* header files.

### 4.4.3. Remote objects as parameters

In POP-C++ it is possible to pass references to remote objects as parameters in a method call. This is true for collective operations also. But it is dangerous to pass a reference of a remote object which is itself a member of the group. This operation can cause a dead-lock situation like illustrated in the example below with the parclass *Integer*:

```
parclass Integer {
public:
    Integer();
    Integer(int wanted, int minp) @ { power = wanted ? : minp; };
    Integer(paroc_string machine) @ { od.url(machine); };
    ~Integer();
};
```

```
seq async void Set(int val);
conc int Get();
mutex void Add(Integer &o);
async conc void Wait(int t);
conc int Sum([in] int x[5000]);

private:
    int data;
};
```

The interesting methods to explain the problematic are highlighted. When the method *Add(Integer &o)* is invoked,

```
Integer o1;
o1.Add(o1);
```

the following code is executed on the remote object *o1*:

```
void Integer::Add(Integer &other)
{
    data+=other.Get();
}
```

A dead-lock situation occurs. The invocation *o1.Add(o1)* implicitly calls the *Get()* method. But since *Add(Integer &o)* is declared *mutex* it prevents the *Get()* method to be executed on this object.

Such situations will not be detected during the POP-C++ compilation process. Therefore the error will show up at runtime only. It is up to the programmer to see whether it is safe to use this mechanism or not depending on the situation.

## 5. Tests

The two test programs included in the implemented POP-C++ distribution serve as basis to test the functionality of collective communication in a POP-C++ application. Especially the *integer* test program explores all group management functions and contains 10 type specific methods on which collective communication is tested. The main program tests all types of collective communication operations.

```
parclass Integer
{
public:
    Integer() @ { od.power(50,20); };
    Integer(int wanted, int minp) @ { power= wanted ?: minp; };
    Integer(paroc_string machine) @ { od.url(machine); };
    ~Integer();

    seq async void Set(int val);
    conc int Get();
    mutex void Add(Integer &other);
    mutex void Add([in] Integer &o1, [in] Integer &o2);

    async conc void Wait([in, size=1] int *t);

    conc int Sum([in] int x[5000]);

private:
    int data;
    classuid(1000);
};
```

## 6. Future work

### 6.1. Parse C++ class declarations

At the actual state, only *.ph* files are parsed to find parclass declarations. The parser could be modified to parse sequential class declarations to generate the same code as it does for parclasses. The question is if it is useful to create groups of sequential objects which are all executed on the local machine.

### 6.2. Clean source code

The group parser has been developed from the POP-C++ parser. At the end of the project, I didn't have enough time to clean the code completely. This is why there are still pieces of code which are useless for the group parser left in the source code.

### 6.3.Sophisticated algorithms for group method invocations

Group invocations are implemented using a very simple algorithm. The group goes through every member within a loop and calls the requested method on it by respecting the POP-C++ semantics. It could be a big interest to find more sophisticated algorithms to improve performances of the collective communication mechanism in POP-C++. For example: use a binary tree structure for broadcast and reduce operations to propagate data to the group members like illustrated in Figure 49.

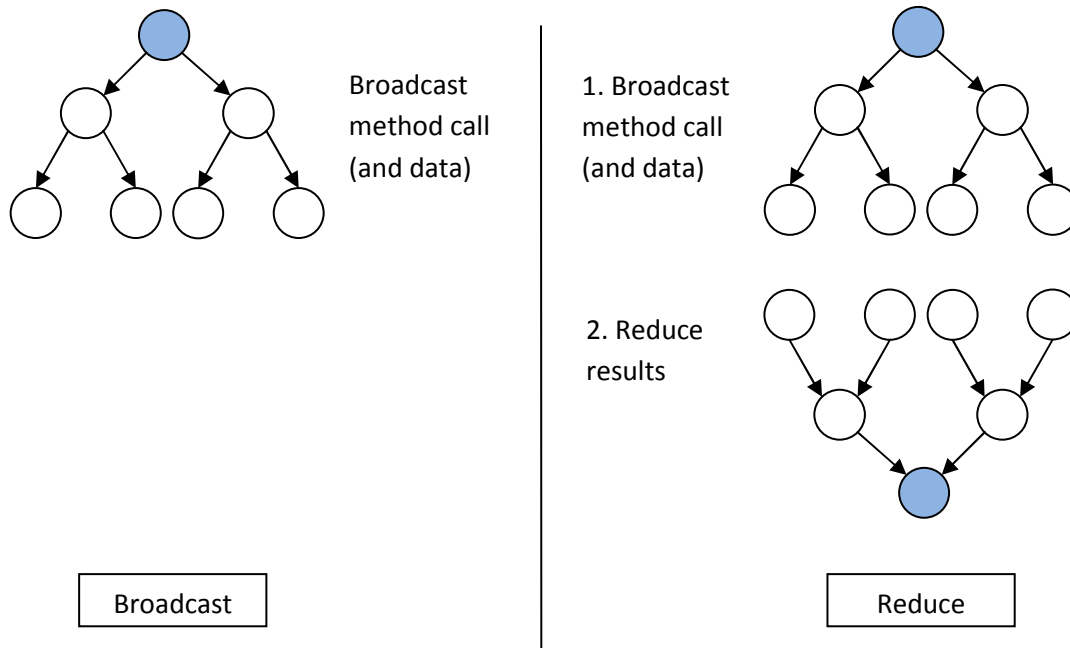


Figure 49 Binary tree structure for broadcast and reduce

Other structures than binary trees are imaginable (e.g. mesh).

Another performance issue is that a group invocation respects the POP-C++ method invocation semantics. The issue is negligibly for asynchronous calls, but synchronous calls force the caller to stay blocked at the point where the invocation takes place and no other work is done during this waiting time. Creating a thread which waits for the group invocation to finish could solve this problem (**Error! Reference source not found.**). The main thread could continue to do other work and check at some point if the before created thread has completed. If this is the case, it can get the results from the temporary thread and the kill it.

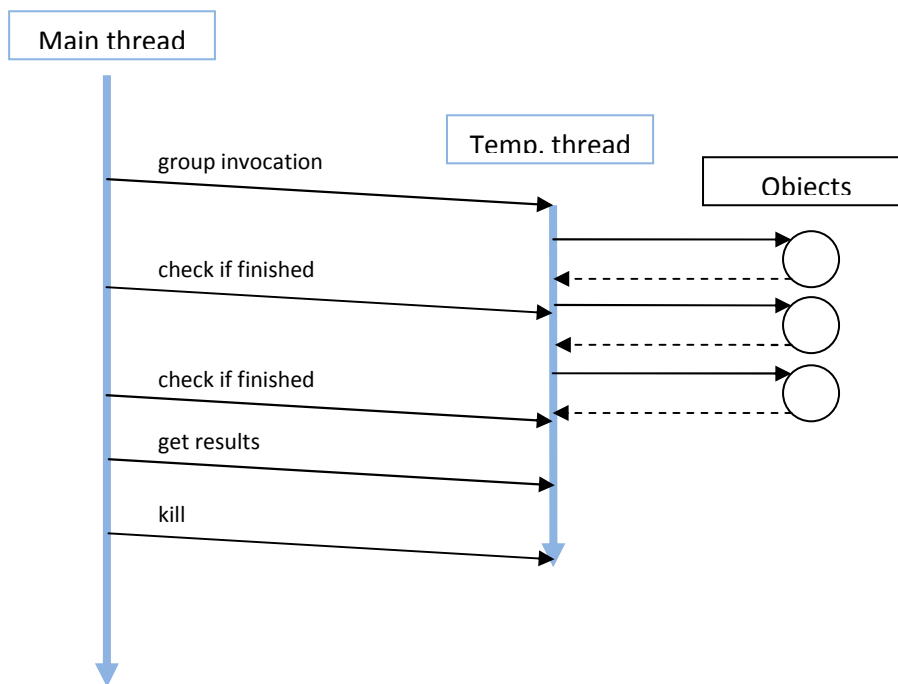


Figure 50 Asynchronous group calls with temporary thread

## 7. Conclusion

The entire implementation of the group parser is embedded in a distribution of the most recent POP-C++ version (1.2). This makes it easy to install and test collective communication operations because the installation procedure is exactly the same as in usual distributions of POP-C++. The runtime has not been affected by the changes; all the additional work to use collective communication is done during the compilation of an application.

---

# General conclusion

---

## 1. Personal conclusion

The first part of the project gave me a good introduction to HPC related problems. A lot of different components have to be considered when doing benchmarks on a cluster (hardware, protocol, OS, technology, exclusive access etc.). It was useful to have the possibility to work on a cluster with some assistance from more experienced people in this domain.

It was very interesting to implement collective communication for POP-C++ by modifying the POP-C++ compilation process. I could study the functionality of the parser which allowed me to better understand the powerful tools *flex* and *bison* (lex and yacc). It was a big challenge, because the source code of the POP-C++ compiler (including the parser) consists of several thousands of lines. But once I achieved the global understanding, discovering more and more details was very enriching for my education. Also embedding the group parser into an existing distribution of POP-C++ helped me to understand how *autoconf* and *automake* can ease the development of a big application.

## 2. Thanks

We'd like to thank all the persons who supported us during this project:

The responsible professors at the *College of engineering and architecture* in Fribourg:

- Pierre Kuonen: for his advices during the entire project and for giving us the possibility to accomplish it in Albuquerque
- François Kilchoer: for his advices during the entire project and for the English corrections in the report
- Jean-François Roche: for his advices during the entire project
- Guilherme Peretti Pezzi: for his advices to solve POP-C++ issues

The responsible externs:

- Thuan-Anh Nguyen: for his help to solve POP-C++ issues on the cluster and for providing the lexer and grammar source files to study the POP-C++ parser
- Barney Maccabe: for giving us the possibility to do this project in Albuquerque
- Rolf Riesen: for his explanations about MPI and collective communication.
- The developers of the *reflpcc* library: for responding to our questions via e-mail
  - Tharaka Devadithya (Indiana University, USA)
  - Kenneth Chiu (SUNY Binghamton, USA)
  - Wei Lu (Indiana University, USA)



# Appendix

## 1. Unrealized designs

This section contains short descriptions of designs which were discussed during the project but not developed. They are mentioned here because they may give ideas of how the current implementation could be modified to improve it.

### 1.1. Syntax for collective communication

The presented design in section 3 on page 46 is one among other ideas that were discussed between me and the responsible persons of this project.

#### 1.1.1. Create a stub for every collective communication operation

This design previews to generate for every parclass a stub class for every collective communication operation (broadcast, scatter, gather and reduce), thus 4 for every parclass. In this design, the programmer wouldn't be allowed to implement methods which have a return value and parameters. They have to be either outgoing methods or incoming methods.

```
myGroup.broadcast().setValue(param);  
myGroup.scatter().setValue(param[]);  
res[] = myGroup.gather().getValue(); //myGroup.gather().getValue(res);  
res = myGroup.reduce().getValue(operation);
```

This design also wants to collect results from the gather operation by using a different syntax. The gather operation should return an array which is not possible in this case because an array is only a pointer in C and the size of the array is not known. To implement that syntax the parser would have to look out for this kind of situation in the programmer's source code and replace the highlighted code by the one in comment on the same line. But it is not the philosophy of POP-C++ to change the programmer's source code.

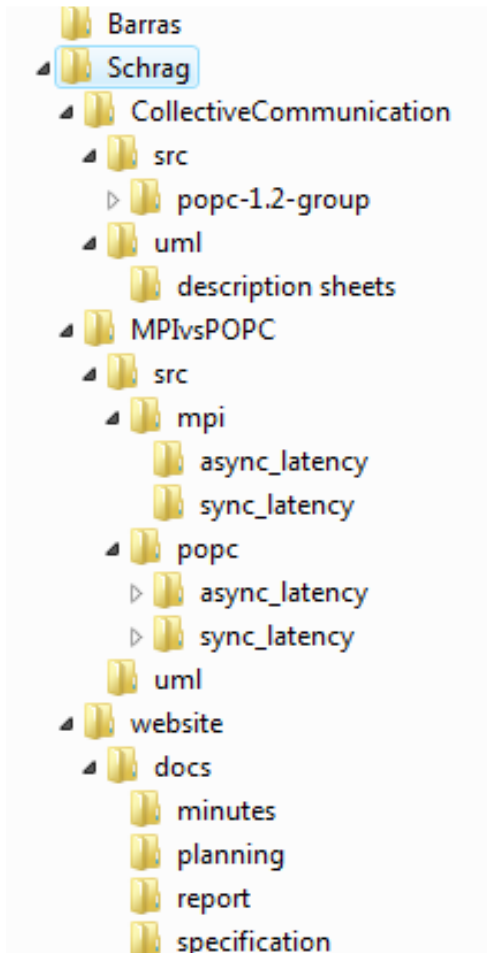
#### 1.1.2. Differentiate incoming and outgoing methods

In this design, the programmer is forced to distinguish incoming and outgoing messages. Two stubs per parclass would have to be generated to do that. But this restriction of allowing only "one-way" methods in parallel classes wouldn't match with the POP-C++ programming language.

```
myGroup.out().setValue(param); // broadcast  
myGroup.out().setValue(param[]); // scatter  
myGroup.in().getValue(res[]); // gather  
myGroup.in().getValue(operation); // reduce
```

## 2. CD structure

Everything concerning the project can be found on the CD coming with this report. There are two folders at the root directory. The folder *Barras* contains the part of my colleague Frédéric Barras and the folder *Schrag* contains everything discussed in this report.



The final report and 3 subfolders are located directly under *Schrag*.

Subfolder *MPIvsPOPC* (first part of the project): Contains the sources of the test programs and the UML file including sequence diagrams of the design of these programs.

Subfolder *CollectiveCommunication* (second part of the project): Contains the sources of the implemented POP-C++ distribution including collective communication. UML files with diagrams describing the POP-C++ parser and the design of collective communication in POP-C++ are also located there.

Subfolder *website*: All the meeting minutes, plannings, report extracts and goal specifications are in this subfolder.

(UML files have been edited with StarUML)

### 3. Source code

This section includes only source code which is helpful for a better understanding when reading this report. The entire source code of the group parser can be found and consulted on the CD which is coming with the report.

#### 3.1.Code model for the base POPGroup template

```
#include <iostream>
#include <vector>
#include "RankException.h"
using namespace std;

template <class U>
class POPGroup {
};
```

#### 3.2.Code model for specialized POPGroup templates

At the actual state, the group parser only generates code for three different reduce operations which are: MAX, MIN and the logical OR. Other functions could be added in the future by adding constant static members to the POPGroup template and adding the code to generate to the file *popc-1.2/grpparser/grpclassmember.cc*. Remember that *T* is a special character in the file below and should never be used in appropriate.

```
template<>
class POPGroup<T> {

    vector<T*> members;
    vector<T*>::iterator it;
    int size;
    _parocGroupT *stub;

public:
    const static int _MAX=0;
    const static int _MIN=1;
    const static int _OR=2;

    POPGroup() {
        size=0;
        stub = new _parocGroupT(&members, &size);
    }

    bool isEmpty() {
        return size==0;
    }

    void emptyGroup() {
        members.clear();
        size=0;
    }

    void add(T &o) {
        members.push_back(&o);
        size++;
    }

    void add(T *o, int nb) {
```

```
        for(int i=0;i<nb;i++) {
            members.push_back(&(o[i]));
            size++;
        }
    }

    void createMembers(int nb) {
        for(int i=0;i<nb;i++) {
            members.push_back(new T());
            size++;
        }
    }

    void removeAt(int rank) {
        if(rank<0 || rank>(size-1))
            throw RankException(rank);
        it=members.begin()+rank;
        members.erase(it);
        size--;
    }

    void remove(T *o, int nb) {
        int rank;
        for(int i=0;i<nb;i++) {
            rank = getRank(o[i]);
            removeAt(rank);
        }
    }

    int getRank(T &o) {
        for(int i=0;i<size;i++) {
            if(members.at(i)==&o) {
                return i;
                break;
            }
        }
        return -1;
    }

    int getSize() const {
        return size;
    }

    T &getMember(int rank) {
        if(rank<0 || rank>(size-1))
            throw RankException(rank);
        else
            return *(members.at(rank));
    }

    void merge(POPGGroup<T> &g) {
        for(int i=0;i<g.getSize();i++)
            add(g.getMember(i));
    }

    _parocGroupT &comm() {
        return *stub;
    }
};
```

### 3.3. Exception handling in collective communication

```
class RankException {
    int rank;
    char message[40];

public:
    RankException(int r) {
        rank=r;
        sprintf(message,"Exception: Rank %d out of range\n",rank);
    }

    char *getMessage() {
        return message;
    }
};
```

## 4. Definitions

Many of the definitions are copied, translated or inspired by definitions found on Wikipedia [10].

<b>Autoconf</b>	Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of it when they are run
<b>Automake</b>	GNU Automake is a programming tool that produces portable makefiles for use by the make program, used in compiling software. It is part of the GNU build system. The makefiles produced follow the GNU Coding Standards.
<b>Bandwidth</b>	The amount of data which can be transferred in a certain period from one computer to another. A higher bandwidth means faster access to the requested data (translated from <a href="http://www.isllight.de/woerterbuch.htm">http://www.isllight.de/woerterbuch.htm</a> )
<b>Benchmarking</b>	In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it
<b>Bison</b>	GNU bison is a free parser generator computer program written for the GNU project. It is mostly compatible with Yacc, and offers several improvements over the earlier program. It is commonly used in conjunction with flex, but lexical analysers can also be hand-written or produced by some other automated method
<b>C</b>	C is a general-purpose, block structured, procedural, imperative computer programming language. Although C was designed as a system implementation language, it is also widely used for applications
<b>C++</b>	C++ (pronounced "see plus plus") is a general-purpose programming language with high-level and low-level capabilities. It is a statically typed, free-form, multi-paradigm, usually compiled language supporting procedural programming, data abstraction, object-oriented programming, and generic programming
<b>Cluster</b>	A computer cluster is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. The components of a cluster are commonly,

	but not always, connected to each other through fast local area networks
<b>CPU</b>	A central processing unit (CPU), or sometimes simply processor, is the component in a digital computer capable of executing a program
<b>BNF</b>	The Backus–Naur form (BNF) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages
<b>Ethernet</b>	Ethernet is a family of frame-based computer networking technologies for local area networks (LANs)
<b>Finite state machine</b>	A finite state machine (FSM) or finite state automaton (plural: <i>automata</i> ) or simply a state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions
<b>Flex</b>	Flex is a tool for generating scanners: programs which recognize lexical patterns in text. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules
<b>GNU</b>	GNU is a computer operating system composed entirely of free software. Its name is a recursive acronym for <i>GNU's Not Unix</i> , which was chosen because its design is Unix-like, but differs from Unix by being free software and by not containing any Unix code
<b>HPC</b>	The term high performance computing (HPC) refers to the use of (parallel) supercomputers and computer clusters, that is, computing systems comprised of multiple processors linked together in a single system with commercially available interconnects
<b>InfiniBand</b>	InfiniBand is a switched fabric communications link primarily used in high-performance computing. The InfiniBand architecture specification defines a connection between processor nodes and high performance I/O nodes such as storage devices
<b>Lex</b>	In computer science, lex is a program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. It reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language
<b>Linux</b>	Linux is a Unix-like computer operating system. Linux is one of the most prominent examples of free software and open source development; its underlying source code can be freely modified, used, and redistributed by anyone.
<b>MPI</b>	The MPI is a language-independent communications protocol used to program parallel computers
<b>Myrinet</b>	Myrinet is a high-speed local area networking system to be used as an interconnect between multiple machines to form computer clusters
<b>Node</b>	A node is a critical element of any computer network. It can be defined as a point in a network at which lines intersect or branch, a device attached to a network, or a terminal or other point in a computer network where messages can be created, received, or transmitted
<b>SPMD</b>	SPMD (Single Process, Multiple Data) or (Single Program, Multiple Data) is a technique employed to achieve parallelism. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster

<b>SSH</b>	Secure Shell or SSH is a network protocol that allows data to be exchanged over a secure channel between two computers. Encryption provides confidentiality and integrity of data. SSH uses public-key cryptography to authenticate the remote computer and allow the remote computer to authenticate the user, if necessary.
<b>TCP</b>	The Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite. TCP provides reliable, in-order delivery of a stream of bytes, making it suitable for applications like file transfer and e-mail.
<b>Templates</b>	In computer programming, templates are a feature of the C++ programming language that allow code to be written without consideration of the data type with which it will eventually be used.
<b>UNIX</b>	Unix is a computer operating system. Today's Unix systems are split into various branches, developed over time by AT&T as well as various commercial vendors and non-profit organizations.
<b>Yacc</b>	Yacc generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF. Yacc generates the code for the parser in the C programming language.

## 5. References

[1] Project description	<a href="http://www.eif.ch/gestionprojets/private/rechercher.jsp?inoid=1620">http://www.eif.ch/gestionprojets/private/rechercher.jsp?inoid=1620</a>
[2] MPI Introduction	<a href="http://www.mhpcc.edu/training/workshop/mpi/MAIN.html">http://www.mhpcc.edu/training/workshop/mpi/MAIN.html</a>
[3] POP-C++ runtime	Semester work summer 2007 "WSDL and POP-C++ in the library of the EIA-FR, Switzerland
[4] The POP-C++ User manual	<a href="http://www.eif.ch/gridgroup/popc/docs/manual.pdf">http://www.eif.ch/gridgroup/popc/docs/manual.pdf</a>
[5] Phoenix overview	<a href="http://sims.cs.unm.edu/ssl/doku.php?id=machine:phoenix:overview">http://sims.cs.unm.edu/ssl/doku.php?id=machine:phoenix:overview</a>
[6] OpenMPI and TCP	<a href="http://www.beowulf.org/archive/2006-November/016904.html">http://www.beowulf.org/archive/2006-November/016904.html</a> <a href="http://www.open-mpi.org/faq/?category=tuning">http://www.open-mpi.org/faq/?category=tuning</a>
[7] Illustration of MPI functions	<a href="http://www.cs.unm.edu/~riesen/lesson_10.pdf">http://www.cs.unm.edu/~riesen/lesson_10.pdf</a>
[8] C++ templates	<a href="http://www.cplusplus.com/doc/tutorial/templates.html">http://www.cplusplus.com/doc/tutorial/templates.html</a>
[9] Flex/Bison	<a href="http://www.gnu.org/software/bison/manual/index.html">http://www.gnu.org/software/bison/manual/index.html</a> <a href="http://www.gnu.org/software/flex/manual/">http://www.gnu.org/software/flex/manual/</a>
[10] Wikipedia	<a href="http://www.wikipedia.org/">http://www.wikipedia.org/</a>
[11] Autoconf/Automake	<a href="http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual_toc.html">http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual_toc.html</a>

## 6. Figures

Figure 1 The POP-C++ class diagram .....	8
Figure 2 Simplified remote method call in POP-C++ .....	9
Figure 3 Different object-sided invocation requests [4] .....	9
Figure 4 Distributed Memory System [2] .....	10
Figure 5 Message passing with buffering at receive [2] .....	11
Figure 6 Asynchronous MPI latency and data sending test scenario .....	15
Figure 7 Synchronous MPI latency and data sending test scenario .....	16
Figure 8 Asynchronous POP-C++ latency and data sending test scenario .....	17
Figure 9 Synchronous POP-C++ latency and data sending test scenario .....	17
Figure 10 Screenshot of asynchronous test program in MPI .....	19
Figure 11 Screenshot of synchronous test program in MPI .....	20
Figure 12 Screenshot of asynchronous test program in POP-C++ .....	21
Figure 13 Screenshot of synchronous test program in POP-C++ .....	22
Figure 14 Output to define latency of OpenMPI asynchronous .....	24
Figure 15 Output to define latency of OpenMPI synchronous .....	25
Figure 16 Output to define latency of POP-C++ asynchronous .....	26
Figure 17 Output to determine latency of POP-C++ synchronous .....	27
Figure 18 Prediction of the test scenarios .....	28
Figure 19 Test in range of single bytes .....	29
Figure 20 Messages of 0 to 1KB in increments of 64B .....	30
Figure 21 Messages of 1KB to 10KB in increments of 1KB .....	31
Figure 22 Messages of 10KB to 100KB in increments of 10KB .....	32
Figure 23 Messages of 0 to 1MB in increments of 64KB .....	33
Figure 24 Global differences between prediction and measurement in OpenMPI async .....	34
Figure 25 Global differences between prediction and measurement in OpenMPI sync .....	34
Figure 26 Global differences between prediction and measurement in POP-C++ async .....	35
Figure 27 Bandwidth decrease in POP-C++ sync .....	35
Figure 28 Global differences between prediction and measurement in POP-C++ sync .....	36
Figure 29 MPI broadcast [7] .....	40
Figure 30 MPI Scatter [7] .....	40
Figure 31 MPI Gather [7] .....	40
Figure 32 MPI Reduce [7] .....	40
Figure 33 MPI Allgather [7] .....	40
Figure 34 MPI All-to-All [7] .....	41





Figure 35 MPI Allreduce [7] .....	41
Figure 36 POP-C++ Broadcast .....	41
Figure 37 POP-C++ Scatter .....	41
Figure 38 POP-C++ Gather .....	42
Figure 39 POP-C++ Reduce .....	42
Figure 40 POP-C++ Allgather .....	42
Figure 41 POP-C++ All-to-all .....	42
Figure 42 POP-C++ Allreduce .....	43
Figure 43 Vastly simplified functionality of the POP-C++ parser .....	45
Figure 44 Use case diagram to handle a group .....	46
Figure 45 Scatter operation in case of inequality of parameter size and group size .....	47
Figure 46 Sequence diagram of use case, "Call type specific method" .....	49
Figure 47 Architecture of the collective communication library .....	50
Figure 48 Integration of the group parser in the POP-C++ compilation process .....	58
Figure 49 Binary tree structure for broadcast and reduce .....	62
Figure 50 Asynchronous group calls with temporary thread .....	63

## 7. Tables

Table 1 Optimal benchmarking for data sending in POP-C++ and MPI .....	13
Table 2 Output to define average bandwidth of OpenMPI asynchronous .....	24
Table 3 Output to define average bandwidth of OpenMPI synchronous .....	25
Table 4 Output to define average bandwidth of POP-C++ asynchronous .....	26
Table 5 Output to define average bandwidth of POP-C++ synchronous .....	27
Table 6 Standard deviation for messages of 10KB to 100KB .....	32
Table 7 Disturbed measurement on the cluster .....	37
Table 8 Performances of POP-C++ vs. MPI .....	38
Table 9 Explanation and visualization of MPI collective communication functions .....	41
Table 10 Explanation and visualization of POP-C++ collective communication functions .....	43
Table 11 How the POP-C++ parser generates methods from .ph file .....	54