A Comparison of existing Python modules of MPI

by

WENJING LIN

THESIS

for the degree of

MASTER OF SCIENCE

(Master i Anvendt matematikk og mekanikk)

Department of Mathematics
Faculty of Mathematics and Natural Sciences
University of Oslo

August 2010

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Preface

This is the written part of my master-degree project at the Department of Mathematics, University of Oslo.

I would like to thank a great number of people who helped and supported me during the last years. First and foremost I want to thank my supervisor, Xing Cai, for his work in this project. Without his inspiration, patience and guidance I could not finish this project.

Many thanks to my friends at Simula Research Laboratory for technical support and constructive criticism. Especially Wenjie Wei has given me great support and help in programming techniques, research ideas and answering my strange questions about MPI programming. Many thanks to Arne Jørgen Arnesen and Else-Merete Bergene, with whom I share the study room with, for many interesting discussions.

I also want to thank the IT-support group of Simula Research laboratory for all kinds of technical support and IT-support group of Stallo at UiT for help with installation of the packages.

A lot of thanks should be given to my parents for their long-term encouragement and support. It is because of their love that I am what I am today. And many thanks to my fellow students and friends who have helped me tremendously with comments, discussion and encouragement. It is them who make studies and life enjoyable here in Norway.

Wenjing Lin
Oslo, August 2010

Contents

Chapter 1

Introduction

Python programming language[1] has gradually gained popularity in the field of scientific computing. It provides easy-to-use syntax and lots of build-in functions operating efficiently on arrays. In connection with parallel computing, Python has been used to simplify, in particular, message-passing based parallel programming. A number of different Python Message Passing Interface (MPI) modules have lately been developed. In this project, we will show the possibilities with Python in parallel computing. Six of the existing Python MPI modules are presented and compared through a set of well-designed experiments. A guideline on choosing suitable MPI modules and implementation issues is produced after comparison.

1.1 Idea and motivation

In the last decade, computer clusters and supercomputers became an affordable resource for many scientific researchers. The concept of high performance computing is also widely spread with the rapid development of hardware. Considering an advanced computation problem, parallel computing makes the computations carried out simultaneously in which the efficiency is greatly improved. The idea is splitting up a large computational problem into small pieces of tasks and solving them concurrently. In real world, a large group of applications require processing of huge amounts of data in sophisticated ways, for instance, oil exploration, weather forecast and early warning of tsunami. For common usage, there are several parallel programming models developed for exploiting the parallel environments: message passing, shared memory and threads. Among them, the message passing model, which is commonly used on distributed-memory machines, has proved to be a comparatively efficient one.

Traditional MPI applications are mostly developed in compiled language like C/C++ and Fortran. However, there are many other languages that have capability to wrap those routine libraries, for example Java, OCaml and Python[2]. Python is one of them with great features for the purpose of scientific computa-

tions.

Nowadays, there are more and more researchers willing to use Python to solve scientific problems. Python has its great advantage in friendliness and flexible syntax. And it supports multi-dimensional array operations by slicing [3]. There are a large number of efficient numerical modules and extensions available as open source [4]. These give huge convenience for scientific computing. Moreover, these features will not only simplify the vector operations dealing numerical problems[5][6], but also greatly simplify MPI function calls in parallel programming.

1.2 Problem description and research methods

This thesis will introduce the usage of six Python MPI modules for parallel programming and apply them in two classic scientific applications. Through various test cases and benchmarks to compare the functionalities and the parallel performances. The work of this master thesis project consists of the following five stages:

1. Searching for existing Python MPI modules and related libraries, then installing them on a Linux cluster.

2. Designing two representative numerical problems and implementing them using different Python MPI modules.

3. Designing a set of tests to compare the modules in different aspects and presenting them in tables and charts

4. Analyzing the measurements with respect to syntax, functionality and performance.

5. Concluding a guideline on choosing suitable Python MPI modules.

Much time have been spent on implementing of the applications and comparison of different packages on the basis of a large number of experiments. In order to give a fair enough comparison results, we have made great efforts on optimization of the code in terms of parallel performance.

Two numerical applications, solving wave equation on 3-dimensional space in parallel and parallel implementation of conjugate gradient method, are implemented in different Python MPI modules. The comparison is performed on the following aspects:

- Completeness of functionality

- Programming style and usage

- Parallel performance

The technical content of this thesis is written for a large group of Python users willing to try MPI-based parallel programming in scientific computing. A basic background knowledge of Python and MPI are also provided in next Chapter. We remark that this thesis only concerns the functionalities and parallel performances using Python in MPI-based parallel programming. For other parallel processing models in Python, efficiency and optimization issues of Python and MPI performance topics, the readers are referred to [7],[8],[9].

1.3 Experimental environments

The experiments are all performed on a Linux cluster with Xeon(R) E5420 2.50GHz processors, inter-connected through Gigabit Ethernets network, which has theoretical peak bandwidth of 1000Mbit/s. The detailed experimental environments and package settings are described in Appendix A.

1.4 Thesis structure

Chapter 2 Background Knowledge. As background for MPI modules in Python, Python programming language and Message Passing Interface are presented. And some numerical python tools are briefly introduced.

Chapter 3 Presentation of Python MPI modules. Six Python MPI modules *Pypar*[10], *MPI for Python (mpi4py)*[11], *MYMPI (pydusa)*[12], *pyMPI*[13], *Scientific.MPI*[14] and *boostmpi*[15] are introduced.

Chapter 4 Comparison of Python MPI modules. Some detailed functionality and implementation features regarding to scientific programming are discussed in this chapter.

Chapter 5 Performance comparison through benchmarks. We apply four micro-benchmarks to evaluate some most important MPI routines in the six Python modules and provide a performance comparison of them.

Chapter 6 Test Cases and Numerical Experiments. Six Python MPI modules are used in parallel programming to solve two real scientific problems. The focus is on evaluating and comparing the flexibility and performance of Python implementations with the MPI modules. Detailed programming issues in using those modules are discussed.

Chapter 7 Conclusions and future work. This chapter concludes the project with some plans for the future work.

Appendix. Appendix A gives a presentation of the hardware and software experimental environments used in this project. And Appendix B provides a C-version MPI Hello World example for comparison with Python-versions with different Python MPI modules.

All experiments and examples in this thesis can be downloaded as a tar package from [16], and executed in the reader's own computing environment. There are also a number of scripting codes for collecting and analyzing the experimental results included in the source code package [16].

Chapter 2

Background Knowledge

There are many parallel programming models and a lot of programming languages designed for all kinds of purposes. Some comparison and study are needed to give most users an understanding about the packages and their features. In this project, the combination of MPI and Python is presented for developing parallel scientific applications efficiently and conveniently.

The concept and theory of Message Passing Interface (MPI) is presented in Section 2.1. Traditionally MPI programs are implemented in compiled languages like C/C++ and Fortran. However, in this project, we show MPI implementations in a high-level script language Python. The concepts of Python programming language in scientific computing are discussed in Section 2.2. A short introduction to some essential Python numerical tools is presented in Section 2.3.

In order to show some basic ideas about MPI programming in Python, a simple example of computing inner product in parallel are implemented in Python with *Pypar*[10] and *NumPy*[17] in Section 2.4. And a C version of implementation is also provided for comparison. Both wall-clock time and speedup results are reported for performance comparison Python and C in MPI programming. More information and discussions about these packages will be presented in the following chapters.

2.1 MPI

MPI is an interface specification of using message passing libraries including definition of protocols and semantic of message passing routines on a wide variety of parallel computers.[2][18]. The first version of MPI [19]was released in 1994 known as MPI-1, which has a static runtime environment. After two-year discussing and developing, some new features such as parallel I/O, dynamic process management, additional language bindings for C++ and remote memory operations were added. It was finalized in 1996 with the name MPI-2. At present, MPI implementations are a combination of MPI-1 and MPI-2.

2.2 Python in parallel world

About thirty years ago, Python programming language was developed by Guido van Rossum at National Research Institute for Mathematics and Computer Science in Netherlands[20]. Nowadays, Python is considered to be a great programming language for scientific computing and has attracted significant interest among computational scientists[1][21]. It has shown the advantage of scripting language for scientific programming and moreover for parallel programming. Python's great properties for scientific application development has been discussed in paper [5] and [6]. In this project, we will specially discuss Python's properties in parallel scientific programming and compare the existing Python Interfaces for MPI.

2.3 Numerical Python packages

In order to achieve good performance of a scientific application, we use the following two numerical python modules: *Numpy* and *Scipy*.

Numpy[17] provides a large library of efficient array operations implemented in C. The computational capabilities are similar to *MATLAB*, where the array operation involves a loop over all array entries is efficiently implemented in C. Moreover, the vectorization[22] and slicing[3] greatly simplify the scientific program and improve the performance.

Scipy[23] is library of scientific tools for Python programming, including various high level science and engineering modules available as an open source package. We will use the 2D sparse matrix module from *Scipy* package to construct a sparse matrix and apply the build-in functions to develop the parallel Conjugate Gradient method in Chapter 6.

2.4 A simple example to compare Python with C in MPI programming

Let us first take an example to see how well Python could be in high performance computing. The idea is to show that Python give sufficiently good parallel performance as C, but Python clearly has easy-to-read and intuitive syntax.
we will show that Python-MPI implementation with the *Pypar* package is able to provide fully comparable parallel performance in comparison with C-MPI implementation.
We write a program in C and Python to calculate the inner product of two arrays. And we evaluate the parallel performance of inner product of two arrays in the compiled language C and the scripting language Python.
We have prepared two arrays `v1` and `v2` with the same size `n`. Then we partition

them evenly to `numproc`[1] processes. Each process gets two sub-arrays `vs1` and `vs2` of length `local_n=n/numproc`. For each process `myrank`[2], we first apply the build-in function `inner` from the *Numpy* package to compute the inner product `res_local` of the two sub-arrays. Then we sum those `res_local` values from each process to the master process and store it in `res` by calling MPI function `MPI_Reduce`. Here we just show some basic ideas, more details are described in the following chapters.

We measure the wall-clock time (in seconds) of the part of code calculating the inner product, and calculate the speed-up results. The performance comparison is illustrated in the form of table and chart.

2.4.1 Parallel computing inner product in Python with Pypar

We first see a segment of code to compute the inner product in Python with package *Pypar* and numerical tool *NumPy*. For two given arrays `v1` and `v2` of length `n`, each process gets two segments of the two arrays of length `local_n`. In python, the operation is performed by using *slicing*, where sub-arrays `vs1` and `vs2` take the references to two segments of the two arrays started from index `myrank*local_n` to `(myrank+1)*local_n`. After that, the inner product of two sub-arrays `res_local` is calculated by calling the build-in function provided in numerical package *Numpy*. Then `res_local` is converted to the type of numerical array with only one element. The type of this element is set to 'd' which is Python standard floating point numbers. . Another one-element numerical array `res` is prepared as a pre-allocated buffer for taking the result in the next operation. At last, the inner product of `v1` and `v2` is calculated by calling the MPI routine `reduce` in package *Pypar*. The value of `res_local` on each process is summed up to process 0 and store in `res`. Here, The reason of preparing two one-element numerical arrays instead of two scalers is that the function `pypar.reduce` can only perform MPI reduce operation on array type. In chapter 4, this will be explained in detail.

Python implementation of parallel inner product, using ***pypar****[10] and* ***NumPy****[17]*

```
1 import numpy
2 vs1=v1[myrank*local_n:(myrank+1)*local_n]
3 vs2=v2[myrank*local_n:(myrank+1)*local_n]
4 res_local=numpy.inner(vs1,vs2)
5 res_local=numpy.array(res_local,'d')
```

[1] The number of processes
[2] The rank number of one process

```
6 res=numpy.array(0.0,'d')
7 pypar.reduce(res_local,pypar.SUM,0,buffer=res)
```

2.4.2 Parallel computing inner product in C

In C, the procedure is almost the same, but three differences are necessary to point out. Instead of using *slicing*, `vs1` and `vs2` take the first element's address of the sub-arrays. And then the inner product `res_local` is calculated in a a for-loop by adding the product of each element in two sub-arrays with length `local_n`. Comparing to the Python version with *Pypar*, C-MPI reduce can perform on scalers directly. `res` is initialized as a C double-precision floating point number.

C implementation of parallel inner product

```
1  include "mpi.h"
2  int i;
3  double *vs1,*vs2,res=0.0,res_local=0.0;
4  #take the first element's address of the subarray on this process
5  vs1=&v1[myrank*local_n];
6  vs2=&v2[myrank*local_n];
7  for(i=0;i<local_n;i++){#for loop calculating inner product
8      res_local+=vs1[i]*vs2[i];
9    }
10 MPI_Reduce ( &res_local, &res, 1,MPI_DOUBLE,MPI_SUM,0, MPI_COMM_WORLD
11 );
```

2.4.3 Measurements and performance analyses

We prepare two arrays with length 2^{18} to test our examples and measure the wall-clock time for comparing performance.

From Table 2.1 and Figure 2.1 2.2 we can see that the wall-clock time consumptions of the two implementations are of the same level. Python is little slow due to wrappers overhead and numpy is little slower than pure C according to reference.

The reason for the bad performance result while the number of processes is larger than 16 may cause by the small computation/communication ratio. That means the time spent on collecting and summing all the `res_local` results takes the comparatively large part of the entire time cost.

Figure 2.1: A comparison of the wall-clock time measurements (in seconds) of computing inner product in parallel among implementations in C and Python

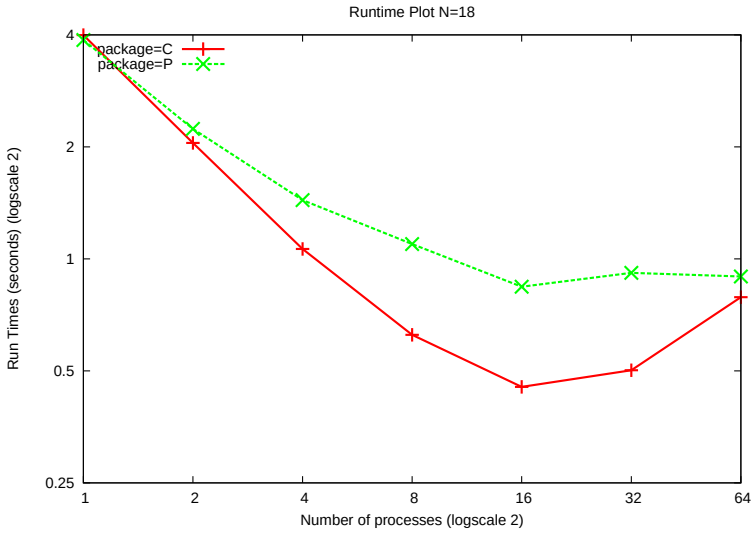


Figure 2.2: A comparison of the speedup results of computing inner product in parallel among implementations in C and Python

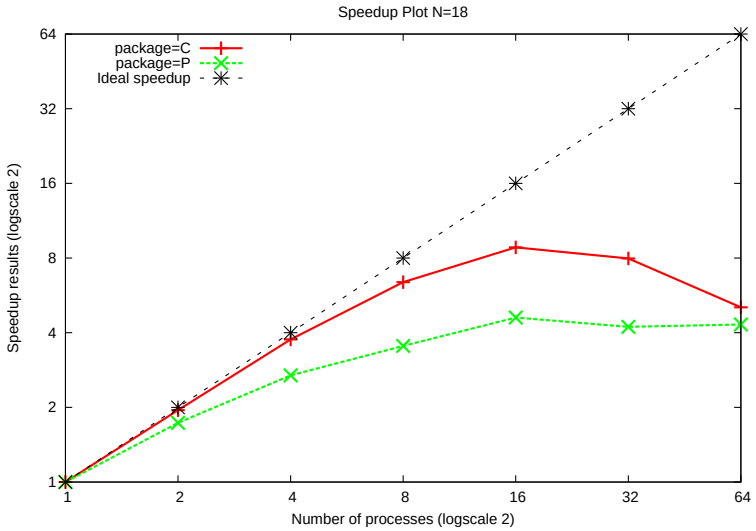


Table 2.1: A comparison of the wall-clock time measurements (in seconds) and speedup results of MPI implementations in C and Python for computing inner product of two vectors of length 2^{18}. The measurements on single core (free of communication) as the reference.

`numproc`	C		Python	
	wall-clock (seconds)	speed-up	wall-clock time (seconds)	speed-up
1	3.996	N/A	3.875	N/A
2	2.049	1.950	2.236	1.733
4	1.063	3.759	1.439	2.693
8	0.625	6.397	1.096	3.537
16	0.453	8.830	0.842	4.602
32	0.502	7.966	0.917	4.225
64	0.789	5.062	0.897	4.319

Chapter 3

Presentation of Python MPI Modules

A number of Python MPI modules have been developed for parallel programming. At the start of the present project, six of them were found warmly discussed on Internet. This chapter provides a presentation and references for those available MPI solutions to develop parallel scientific applications in the high-level programming language Python. Six listing Python MPI modules are briefly introduced respectively in terms of implementation features for scientific computing.

- Pypar[10]
- MPI for Python[11]
- MYMPI[12]
- pyMPI[13]
- Scientific.MPI[14]
- boostmpi[15]

For the purpose of illustrate how to build a parallel application with Python MPI modules from scratch, one simple parallel Hello-world program is implemented with each Python MPI modules. From the simple Hello-world examples, the following steps are discussed:

1. Importing the Python MPI module and initializing the MPI execution environment.
2. Determining the size of the group associated with a communicator and the rank of the calling process in the communicator.
3. Terminating MPI execution environment.

A C-version MPI Hello-world example is given in Appendix for comparison. These packages are all developed as an Python interface of the corresponding MPI routines in C. Some of them could transmit any type of Python objects, some provide a interactive parallel computing environment.We will discuss these features of each module respectively.

In the last section, these different features are collected in one table to give an overview of these modules. The exhaustive comparisons of some most important MPI routines from these modules are presented in Chapter 4 and the parallel performance is measured and compared in Chapter 5.

3.1 pypar

Pypar [10] is developed as a project of the Australian National University since 2002. *Pypar* is well known for its simply syntax and sufficiently good performance. It is considered as an efficient and easy-to-use module for parallel programming in the spirit of Python. It wrappers a small but essential subset of MPI routines.

Following is the parallel Hello-world example implemented with *Pypar*,

```
1 #!/usr/bin/env python
2 import pypar              #automatically initialize MPI
3 numproc=pypar.size()
4 myrank=pypar.rank()
5 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
6 pypar.finalize()
```

Running on four cores with

```
1 >>mpirun -np 4 small_pypar.py
```

In this example, there is no explicit function call to initialize MPI execution environment. MPI is initialized automatically while importing the module using `import pypar`. The number of processes are determined by command-line input `mpirun -np 4`. At last, the MPI execution is terminated by calling `finalize()`. *Pypar*, as the best known Python MPI module, has the following features:

Easy-to-use. The syntax is easy and clear which we have already seen in the parallel Hello-world example. Comparing with the 12 lines in the C-version parallel Hello-world example, Python-version with *Pypar* takes only 6 lines.

Flexible. General Python objects of any types can be communicated with *Pypar*. *Pypar* users don't need to specify the detailed information for a buffer, like the data type and the buffer size.

Minimal MPI routines. Only some most useful and important MPI functions are included in *Pypar*. Both and initializing MPI execution environment and communicator are declared implicitly. The user has no control of how and when MPI is initialized.

3.2 MPI for Python (mpi4py)

MPI for Python (*mpi4py*) package is constructed on top of the MPI-1/2 specifications and provides an object oriented interface which closely follows MPI-2 C++ bindings[11]. The C MPI user could use this module without learning a new interface[24]. Therefore It is widely used as an almost full package of MPI library in Python. The parallel Hello-world example is shown as below.

```
1 #!/usr/bin/env python
2 from mpi4py import MPI
3 comm=MPI.COMM_WORLD
4 numproc=comm.Get_size()
5 myrank=comm.Get_rank()
6 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
7 MPI.Finalize()
```

In this example, comparing with *Pypar*-version, there is one additional line before getting the number of processes. A MPI communicator `MPI.COMM_WORLD` is declared explicitly, which signifies that the operation is taking place within all the processes associated with this communicator. Apart from this, *mpi4py* has the following features:

Containing most of MPI routines. *mpi4py* is developed to have most MPI routines. Not only point-to-point and collective communications, but also dynamic process management and topologies are provided in this package.

Better syntax match with C MPI function calls. *mpi4py* provides most similar syntax with MPI functions in traditional compiled languages. But on the other hand, it is not easy for a Python user without MPI background, which wants to build a parallel Python program from scratch.

Flexible *mpi4py* provides two types of communications according to data types. The MPI function calls with capitalized first letter, i.e., the `Send(), Recv(), Sendrecv(), Reduce(), Gather() and Bcast()` methods provide support for communications of memory buffers. The variants `send(), recv(), sendrecv(), reduce(), gather() and bcast()` can communicate general Python objects. For the former type, *mpi4py* allows the user to construct a derived MPI data types to describe complicated non-contiguous

data layouts, like what we do in C MPI programming. This helps to avoid the overhead of packing/unpacking operations. One example of defining derived data types for the communication of multidimensional arrays is presented in Chapter 6.

MPI-enabled Python interpreter. Sometimes, it is convenient and easy for debugging and learning with MPI-enabled interpreter. mpi4py provides this capability, but the user need to re-build the package with some special configuration steps described in the user manual[11].

3.3 MYMPI

MYMPI (also called *pydusa*) [12] was developed by Tim Kasiser as part of the National Biomedical Computation Resource. It contains about 30 of the most important routines of MPI library and provide closely matched syntax with C and Fortran MPI routines. We remark that *MYMPI* provides different approach for handling the received values. MPI routines provided by the module *MYMPI* return the received values, instead for storing it directly in a pre-allocated buffer, like what we do in C-MPI implementation.

Following is the parallel Hello-world example implementation in Python with *MYMPI*.

```
1 #!/usr/bin/env python
2 import mpi,sys
3 sys.argv=mpi.mpi_init(len(sys.argv),sys.argv)
4 comm=mpi.MPI_COMM_WORLD
5 numproc=mpi.mpi_comm_size(comm)
6 myrank=mpi.mpi_comm_rank(comm)
7 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
8 mpi.mpi_finalize()
```

Explicit MPI_Init call. From the example we can see that one more line is added comparing to the *mpi4py*-version. The user needs to call `mpi.mpi_init` explicitly on every process to initialize the MPI execution environment. It provides better control of how and when MPI is initialized. And `mpi.mpi_init` returns the command line arguments after called.

Better match with C and Fortran. *MYMPI* provides similar MPI syntax and semantics with C and Fortran. The purpose is for implementing MPI programs in Python mixed with C and Fortran.

Only scalars and numerical arrays supported. *MYMPI* is one of the two Python MPI modules which does not support communication of arbitrary

Python objects in these six modules. The other one is *Scientific.MPI* which will be presented in section 3.5. The user needs to specify the buffer size and the data type as what we do in C MPI programming. The detailed syntax is explained in Chapter 4.

3.4 pyMPI

pyMPI[13] was developed by Patrick Miller at Lawrence Livermore National Laboratory in 2002. *pyMPI* provides a simplified Python interface suitable for basic parallel programming based on MPI. About 120+ of MPI routines are implemented.

From the parallel Hello-world example we can see that it initialize the MPI execution environment implicitly. The syntax is as clear and easy as *Pypar*. The difference is that the `size` and `rank` attributes of the module indicate the number of operating processes and the identifier of the calling process, instead of calling the MPI functions. We remark that the first line of the example, the environment variable is set to `pyMPI` instead of `python`. Otherwise, a command-line argument should be given while running. See the following example,

```
1 #!/usr/bin/env pyMPI
2 import mpi
3 numproc=mpi.size
4 myrank=mpi.rank
5 proc_name=mpi.name
6 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
```

The user could also use command-line argument `pyMPI` instead of setting the environment variable. An example of MPI running it without setting the environment variable in code is:

```
1 >>mpirun -np 4 pyMPI small_pyMPI.py
```

In addition, *pyMPI* has following features:

Python interpreter with MPI. It can be used not only as a Python interface of MPI routines, but also a custom version of Python interpreter with `MPI_Init` built in.[25]. *pyMPI* provides a simple-to-use console input on the master node and broadcast it to the slave nodes. This means the user can test and learn parallel codes iteratively.

Value return model. The same with *MYMPI*, *pyMPI* provides a value return model to return the received value for MPI calls like recv, reduce, allreduce and gather. This allows *pyMPI* to make all objects communicated without packing them into buffers.

Error check. *pyMPI* checks MPI calls strictly and converts MPI errors into Python exceptions.

The "two message" model. In order to transmit large arbitrary Python objects, *pyMPI* cut the message into two parts: one with the size of the message and a small part which smaller than the eager message limit sent first, and the other carrying the rest of the message sent afterwards. This may brings some extra overhead for communications.

3.5 Scientific.MPI

Scientific.MPI is one module of *ScientificPython* which is a package contains a collection of Python modules specially for scientific computing. Only 14 MPI routines are implemented. From the following parallel Hello-world example, we remark that the first line of the example, the environment variable is set to `mpipython` instead of `python`.

```
1 #!/usr/bin/env mpipython
2 from Scientific import MPI
3 comm=MPI.world.duplicate()
4 numproc=comm.size
5 myrank=comm.rank
6 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
```

The user could also use command-line argument `mpipython` instead of setting the environment variable. An example of MPI running it without setting the environment variable in code is:

```
1 >>mpirun -np 4 mpipython small_Scientific.py
```

In addition, *Scientific.MPI* has following features:

Python interpreter with MPI The same with pyMPI, Scientific.MPI also patches the interpreter. One can start with typing `mpipython` from command line and run parallel code interactively.

Explicitly declaration of communicator The same with *MYMPI*, from the example above we can see, *Scientific.MPI* declare a communicator explicitly. And the number of processes `size` and the identifier of the calling process `rank` are attributes of the communicator, instead of calling the MPI functions.

Only string, scalers and numerical arrays supported The same with *MYMPI*, for collective communication, i.e. broadcast(),reduce(),allreduce() and share(), *Scientific.MPI* only supports numerical arrays. For blocking communication, i.e. send() and receive(), only string and numerical array are supported.

3.6 boostmpi

Boost.MPI is an alternative C++ interface to MPI which provides better support of C++ development [26] than the existed C++ bindings for MPI. It offers more MPI functionality over the C bindings. *boostmpi*[15] is a Python interface built directly on top of the C++ *Boost.MPI* by using the Boost.Python library[26]. It supports most commonly used MPI routines.

Any Python object can by communicated by apply *boostmpi*. User-defined Python functions are needed for collective communication calls, such as reduce and allreduce, which is explained and shown in next chapter. The programming philosophy and style are more close to C++, so it is considered to be suitable for mixed MPI programming with C++ and Python.

From the parallel Hello-world example below we can see that, *boostmpi* has to declare a communicator explicitly. And the number of processes `size` and the identifier of the calling process `rank` are attributes of the communicator, instead of calling the MPI functions.

```
1 #!/usr/bin/env python
2 import boostmpi
3 comm=boostmpi.world
4 numproc=comm.size
5 myrank=comm.rank
6 print "Hello, world! I'm process %d of %d."%(myrank,numproc)
7 boostmpi.finalize()
```

Apart from these, *boostmpi* has one special feature that for collective communication, i.e. reduce() and allreduce() need a user-defined function as the collective operation. For example,

```
1   result=allreduce(comm,result_local,lambda x,y:x+y)
```

where `lambda x,y:x+y` is a user-defined function with the same functionality as the collective operation `sum`.

3.7 Conclusion

At last of this chapter, the different features are collected in the Table 3.1 for comparison. More comparison in functionality and implementation features are presented in next chapter.

	pypar	mpi4py	myMPI	pyMPI	Scientific.MPI	boostmpi
Pre-allocated buffer for send-recv	Yes	Yes			Yes	
Explicit MPI_Initialize		Yes	Yes			
Explicit communicator		Yes	Yes		Yes	
Interactively parallel run		Yes		Yes	Yes	
Arbitrary Python object	Yes	Yes		Yes		Yes

Table 3.1: Comparison of different features in six Python MPI modules

Chapter 4

Comparison of Python MPI Modules

Besides the characteristics presented in the previous chapter, some detailed functionality and implementation features regarding to scientific programming will be discussed in this chapter. For the purpose of better illustration, this chapter is organized as following:

In section 4.1, a list of some important MPI routines are presented. In section 4.2 and 4.3, both point-to-point communications and collective communications, the syntax of 12 MPI routines are compared. For each MPI routine, the main functionality is described first, and then for each MPI routine, the syntax differences are compared in a tabular form. At last, in section 4.4, some suggestions and guidelines are discussed regarding to functionality and implementation features.

The user can use this chapter as a brief user manual or reference for MPI programming in Python. The parallel performance of each Python MPI module is compared in Chapter 4. Examples and test cases in real scientific applications are given in Chapter 6.

4.1 Important MPI functions

The MPI standard contains hundreds of functions. However, only a small essential subset of MPI are used in the majority of scientific parallel applications. This section describes 12 of the most important MPI functions listed in Table 4.1, where we can see which of them are provided by the six different packages. And then we will compare the syntax differences among C-MPI and six Python MPI modules one by one. Some detailed usage issues of each function calls are pointed out for further implementation. Please refer to the MPI tutorial [18] for more concepts and theories about MPI programming.

Table 4.1: A list of 12 most important MPI functions in different Python MPI modules

	pypar	mpi4py	myMPI	pyMPI	Scientific.MPI	boostmpi
MPI_Send	√	√	√	√	√	√
MPI_Recv	√	√	√	√	√	√
MPI_Sendrecv		√		√		
MPI_Isend		√		√	√	√
MPI_Irecv		√		√	√	√
MPI_Bcast	√	√	√	√	√	√
MPI_Reduce	√	√	√	√	√	√
MPI_Allreduce		√		√	√	√
MPI_Gather	√	√	√	√		√
MPI_Allgather		√		√		√
MPI_Scatter	√	√	√	√		√
MPI_Alltoall		√	√			√

4.2 Point-to-point communications

A point-to-point communication involves the transmission of a message between a pair of MPI processes, from one sender to one receiver. It is heavily used in scientific applications for exchanging messages between adjacent processes. For instance, we use point-to-point communications to exchanging boundary conditions between the neighboring sub-domains in Chapter 6.

4.2.1 Blocking communication mode

From Table 4.2 we see the syntax differences of `MPI_Send` and `MPI_Recv` among different Python MPI modules. The C MPI routines are listed in the first entry for comparison. Clearly, Python-MPI routines have clear and easy syntax and some parameters are optional.

The following are some comments of the functions and parameters for the blocking communication routines:

- In *Pypar*, the `send` and `reveive` routines provide three protocols which handle any picklable Python objects. The three protocols are array, string and vanilla, which can be decided by setting the optional argument `vanilla` to corresponding value. Values 'array' and 'string' set the protocol to handling numpy arrays and text strings respectively. Setting the value to 1 forces the protocal for any type which will be less efficient. Besides, there is another optional parameter `bypass`. The default value is '`False`'. When it is '`True`', all error checks are set to passed in order to reduce the latency. Furthermore, the user can choose the third keyword parameter

Table 4.2: The funcion calls of the blocking communication in different Python MPI modules

C	`int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
Pypar	`pypar.send(sendbuf, destination, tag=default_tag, use_buffer=False, bypass=False, vanilla=False)` `recv_ref=pypar.receive(source, tag=default_tag, buffer=None, return_status=False, bypass=False, vanilla=False)`
mpi4py	`mpi4py.MPI.COMM_WORLD.Send(sendbuf, int destination, tag=t)` `mpi4py.MPI.COMM_WORLD.Recv(self, buf, int source, tag=t, Status status=None)`
myMPI	`mpi.mpi_send(sendbuf, sendcount, sendtype, destination, tag, communicator)` `recvbuf=mpi.mpi_recv(recvcount, recvtype, source, tag, communicator)`
pyMPI	`mpi.send(sendbuf, destination, tag=t)` `recvbuf, status=mpi.recv(source, tag=t)`

Scientific.MPI	`Scientific.MPI.MPICommunicator.send(sendbuf, destination, tag)`
	`message, source, tag, count =`
	`Scientific.MPI.MPICommunicator.receive(recvbuf, source, tag)`

boostmpi	`boostmpi.world.send(destination, tag=0, value=sendbuf)` `recvbuf=boostmpi.world.recv( source=-1, tag=-1, return_status=False)`

'`use_buffer=True`' which assume the existence of buffer at the recipient. Meanwhile, the corresponding `pypar.receive` will specify an optional parameter `buffer=recvbuf`, where `recvbuf` is pre-allocated. This is used to avoid creating a new buffer for received values.

- *mpi4py* is the only one of these six modules which implements `MPI_PROC_NULL` correctly as it should be. It greatly simplifies the exchanging procedure to check whether the destination or the source process exists or not. We will see an example in Chapter 6. However, in order to passing non-contiguous memory buffer efficiently, the user needs to define derived data types, as what we do in C MPI programming.

- While using *MYMPI*, we remark that the received value is returned instead of storing in a pre-allocated buffer. And only scalars and numerical arrays are supported for message-passing. The user needs to specify the size and the date type of `sendbuf`.

- While using `receive` provided by *pyMPI*, the function returns a tuple with received values and status. While using `receive` provided by *Scientific.MPI*, the function returns a tuple with received values, source process identifier, tag and size received.

From Table 4.3 we see the syntax differences of `MPI_Sendrecv` between the only two modules that provide this MPI routine, *mpi4py* and *pyMPI*. The MPI routine is commonly used for avoiding deadlock[1].

Table 4.3: The funcion calls of MPI_Sendrecv in different Python MPI modules

C	`int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,`
	`int dest, int sendtag,`
	`void *recvbuf, int recvcount, MPI_Datatype recvtype,`
	`int source, int recvtag, MPI_Comm comm,`
	`MPI_Status *status)`

mpi4py	`mpi4py.MPI.COMM_WORLD.Sendrecv(self, sendbuf, int dest=0, int sendtag=0,` `recvbuf=None, int source=0, int recvtag=0, Status status=None)`
pyMPI	`recvbuf,status=mpi.sendrecv(sendbuf, dest, source)`

[1]Deadlock: the message passing procedure cannot be completed, normally because of wrong send-recv patterns.

4.2.2 Non-blocking communication mode

From Table 4.4 we see the syntax differences of `MPI_Isend` and `MPI_Irecv` among different Python MPI modules. Non-blocking communication mode is used for overlapping of communication and computation to use the common parallelism more efficient. Comparing with blocking communications, while waiting for the completeness of receive operations, the computation-only part is running simultaneously.

Table 4.4: The funcion calls of the non-blocking communication in different Python MPI modules

C	`int MPI_Isend( void *buf, int count, MPI_Datatype datatype,`
	`int dest, int tag, MPI_Comm comm, MPI_Request *request )`
	`int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,`
	`int tag, MPI_Comm comm, MPI_Request *request )`

mpi4py	`mpi4py.MPI.COMM_WORLD.Isend(selfbuf, destination=0, tag=0)` `mpi4py.MPI.COMM_WORLD.Irecv(recvbuf, source=0, tag=0)`
pyMPI	`mpi.isend(sendbuf, destination, tag=t)` `request=mpi.irecv(source, tag=t)`

Scientific.MPI	`Scientific.MPI.MPICommunicator.nonblockingSend(sendbuf,`
	`destination, tag)`
	`Scientific.MPI.MPICommunicator.nonblockingReceive(sendbuf,`
	`source=None, tag=None)`

boostmpi	`request=boostmpi.world.isend( communicator, destination ,`
	`tag=0, value=sendbuf)`
	`request_with_value=boostmpi.world.irecv( communicator,`
	`source=-1, tag=-1)`

4.3 Collective communications

A collective communication involves every process in a group. We will present seven of the most commonly used routines. From Table 4.5 to Table 4.11, we

show the syntax and parameter differences among the six Python MPI modules. C MPI routines are listed as the first entry for comparison. Some explanations regarding to the syntax and implementation features are discussed.

- The collective MPI routines provided by *Pypar* and *Scientific.MPI* only support numerical arrays, but not scalars. While transmitting a scalar, the user needs to define a numerical array with the scalar as the first element.

- The collective MPI routines provided by *MYMPI*,*pyMPI* and *boostmpi* need to take the return values instead of storing them in pre-allocated buffers.

- The user needs to define a collective operation while using collective MPI routines provided by *boostmpi*. For instance, `lambda x,y:x+y` is defined as the collective operation `sum`.

From Table 4.5 to Table 4.11 we see the syntax differences of `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Gather`, `MPI_Allgather`, `MPI_Scatter` and `MPI_Alltoall` among different Python MPI modules.

Table 4.5: The funcion calls of `MPI_Bcast` in different Python MPI modules

C	`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`
Pypar	`pypar.broadcast(sendbuf, root)`
mpi4py	`mpi4py.MPI.COMM_WORLD.Bcast(sendbuf, root=0)`
myMPI	`recvbuf=mpi.mpi_bcast(sendbuf, sendcount, sendtype, root, communicator )`
pyMPI	`recvbuf=mpi.bcast(sendbuf, int root=host_rank)`
Scientific.MPI	`Scientific.MPI.MPICommunicator.broadcast(sendbuf, root)`
boostmpi	`recvbuf=boostmpi.broadcast(Communicator, sendbuf, root)`

Table 4.6: The funcion calls of `MPI_Reduce` in different packages

C	`int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
Pypar	`pypar.reduce(sendbuf, op, int root, buffer=recvbuf)`
mpi4py	`mpi4py.MPI.COMM_WORLD.Reduce(sendbuf=None, recvbuf=None, op=SUM, root=0)`
myMPI	`mpi.mpi_reduce(sendbuf, int sendcount, sendtype, op, int root, communicator)`
pyMPI	`recvbuf=mpi.reduce(sendbuf, op)`
Scientific.MPI	`Scientific.MPI.MPICommunicator.reduce(sendbuf, recvbuf, op, root)`
boostmpi	`recvbuf=boostmpi.reduce(communicator, sendbuf,op)`

Table 4.7: The funcion calls of `MPI_Allreduce` in different Python MPI modules

C	`int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
mpi4py	`mpi4py.MPI.COMM_WORLD.Allreduce( sendbuf=None, recvbuf=None, op=SUM)`
pyMPI	`recvbuf=mpi.allreduce(sendbuf, op)`
Scientific.MPI	`Scientific.MPI.MPICommunicator.allreduce(sendbuf,recvbuf,op)`
boostmpi	`recvbuf=boostmpi.all_reduce(communicator, sendbuf, op)`

Table 4.8: The funcion calls of `MPI_Gather` in different Python MPI modules

C	`int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Pypar	`recvbuf = pypar.gather(sendbuf, root, buffer=None, vanilla=False):`
mpi4py	`mpi4py.MPI.COMM_WORLD.Gather(sendbuf=None, recvbuf=None, root=0)`
myMPI	`recvbuf=mpi.mpi_gather(sendbuf,sendcount,sendtype,recvcount, recvtype,root,communicator)`
pyMPI	`recvbuf = mpi.gather(sendbuf)`
boostmpi	`recvbuf = boostmpi.gather( communicator=boostmpi.world, sendbuf, root)`

Table 4.9: The funcion calls of `MPI_Allgather` in different Python MPI modules

C	`int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
mpi4py	`mpi4py.MPI.COMM_WORLD.Allgather(self, sendbuf=None, recvbuf=None)`
pyMPI	`recvbuf = mpi.allgather(sendbuf)`
boostmpi	`recvbuf = boostmpi.all_gather(communicator=world, sendbuf)`

Table 4.10: The funcion calls of `MPI_Scatter` in different Python MPI modules

C	`int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,`
	`  void* recvbuf, int recvcount, MPI_Datatype recvtype,`
	`  int root, MPI_Comm comm)`

Pypar	`pypar.scatter(x, root, buffer=None, vanilla=False):`
mpi4py	`mpi4py.MPI.COMM_WORLD.Scatter(self, sendbuf, recvbuf, int root=0)`
myMPI	`recvbuf = mpi.mpi_scatter(sendbuf,sendcount,sendtype,` `  recvcount,recvtype,root,communicator)`
pyMPI	`recvbuf = mpi.scatter(sendbuf)`
boostmpi	`recvbuf = boostmpi.scatter(communicator=world, sendbuf, root)`

Table 4.11: The funcion calls of `MPI_Alltoall` in different Python MPI modules

C	`int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,`
	`  void* recvbuf, int recvcount, MPI_Datatype recvtype,`
	`  MPI_Comm comm)`

mpi4py	`mpi4py.MPI.COMM_WORLD.alltoall(self, sendobj=None, recvobj=None)`
myMPI	`recvbuf = mpi. mpi_alltoall(sendbuf,sendcount,sendtype,` `  recvcount,recvtype,communicator)`
boostmpi	`recvbuf = boostmpi.alltoall(communicator=world, sendbuf)`

4.4 Conclusion

After comparing syntax and implementation features among the different Python MPI modules, we can now take a look back at Table 4.1. *mpi4py* provides all the most important MPI routines and close syntax with C-MPI implementation. However, *Pypar* and *pyMPI* are with clear and easy-to-use syntax. Comparatively, *Pypar* is more flexible with different protocols for passing different Python objects, different options for using buffer and different options for returned status. Module *boostmpi* supports better C++ development style and nearly all the features of C++ library are available in this *Boost.MPI* Python bindings. Therefore, it is particularly suitable for mixed programming users in Python and C++. The other two modules *MYMPI* and *Scientific.MPI* do not support communication of arbitrary Python objects. It is more specified for the scientific computing purpose as a part of package *ScientificPython*.

Chapter 5

Performance Comparison through Benchmarks

In the previous chapter, we discussed the functionalities and implementation features of six different Python MPI modules. In order to compare the MPI performance of each module, in this chapter, we apply four micro-benchmarks to evaluate some most important MPI routines in these modules and provide a performance comparison of them.

Four single micro-benchmarks are performed: ping-pong test, bcast test, reduce test and allreduce test. The timing is measured in the following manner[27]:

1: set up the test (initialize a message of length 2^{23})
2: start timing
3: loop of the MPI operations over different message sizes in power of two
4: stop timing
5: compute the appropriate metric

The experiments are all performed on a Linux cluster with Xeon(R) E5420 2.50GHz processors, inter-connected through Gigabit Ethernets network. The detailed experimental environments and package settings are described in Appendix A.

We measures messages with lengths from 8 bytes to 2^{23} bytes. Each test is run 200 iterations. To avoid the overhead of calling the timer for each operation, we perform the timing operation outside the iteration loop. Then we take the average value of measurements over a number of iterations. The measurements may fluctuate based on the network traffic and the load on the cluster.

After the measurements, performance reports are generated first in raw data files. Then we collect the raw data for each modules and plot graphs for better illustration.

In following text, each test is described in terms of purpose, methodology and performance results. Then a performance comparison is generated in char form.

5.1 Performance comparison of point-to-point communication with Ping-pong test

The combination of latency and bandwidth defines the speed and capacity of a network. Through testing the latency and bandwidth of each implementation, we could measure the communication overhead which the Python MPI modules bring in To measure the actual performance of Python-MPI communication, in terms of latency and bandwidth, we use the Ping-pong micro-benchmark. It makes the use of `MPI_Send` and `MPI_Recv`. We test these six Python MPI modules and compare them with a C-MPI implementation. The test methodology are described in the *Pypar* tutorial documents provided in the *Pypar* distribution[10]. The C-MPI test program and the Python-*Pypar* test program are provided in an early version of *Pypar* distribution. We implemented five more versions of the Ping-pong test program with *mpi4py, MYMPI, PYMPI, Scientific.MPI* and *boostmpi*.

The benchmark test involves two MPI processes. We measure the time used for exchanging a series of messages with different lengths between these two processes. The time used to pass a message can be approximated by the following mathematical model[10]:

$$t = latency + message\_size \times \frac{1}{bandwidth} \tag{5.1}$$

The measurement results for each of the implementations include the message size used for testing and the corresponding time used for passing the message with this size.

After the measurements, we evaluated the actual values of latency and bandwidth by using a least squares strategy.

	C	pypar	mpi4py	myMPI	pyMPI	Scientific.MPI	boostmpi
Latency	8	25	14	33	133	23	162
Bandwidth	967.004	898.949	944.475	364.18	150.901	508.972	100.658

Table 5.1: Performance comparison among six Python MPI modules layered MPI and C-version MPI on a Linux-cluster with respect to the latency (micro-seconds) and bandwidth(Mbytes/second)

From Table 5.1 we can see that mpi4py provides the best performance in the bandwidth and the latency.Regarding the latency, the extra Python layer of MPI routines results in larger overhead than standard C MPI implementation. Comparatively, pyMPI and boostmpi have bad performance in both the bandwidth and the latency. As we have mentioned in Chapter 3, pyMPI uses "two messages model" to handle the communication of arbitrary Python objects. This may bring some extra overhead to the communications. And for boostmpi, it builds on top of a C++ MPI interface boost.MPI by using the *Boost.Python* library. The bad performance may be caused by this reason.

5.2 Performance comparison of collective communication

Collective communications are heavily used in many parallel scientific applications. We use three micro-benchmarks provided by LLCBench.MPBench[27] to test the performance of three collective MPI routines: `MPI_Bcast`, `MPI_Reduce` and `MPI_Allreduce`. The original benchmarks are implemented in C. We implemented each of them in Python with six different Python MPI modules for our purpose of comparison. These benchmarks measures the number of megabytes per second computed from the iteration times and the lengths of messages.

From Figure 5.1, we can see that *Scientific.MPI* and *mpi4py* have as comparatively good performance comparing with other Python MPI modules. For messages with size larger than 4MB, they have as good performance as a C-MPI implementation.

From Figure 5.2, we can see that *mpi4py* has the best performance comparing with the other five Python MPI modules. But both the performance of *mpi4py* and *Scientific.MPI* are unstable related to the different message sizes.

For some modules without MPI Allreduce routine provided(*pypar* and *MYMPI* [1]), a combination of `MPI_Bcast` and `MPI_Reduce` is used instead of *MPI_ Allreduce*. We first run a reduction operation to collect the values to the root process. And then the result is broadcast from the root to the other processes.

From Figure 5.3, we can see that both *mpi4py* and *Scientific.MPI* have a fairly good performance. *pyMPI* and *boostmpi* have comparatively bad performance in this experiments. The bad performance may be caused by the same reason mentioned in the previous section.

[1] see Table 4.7 for more information

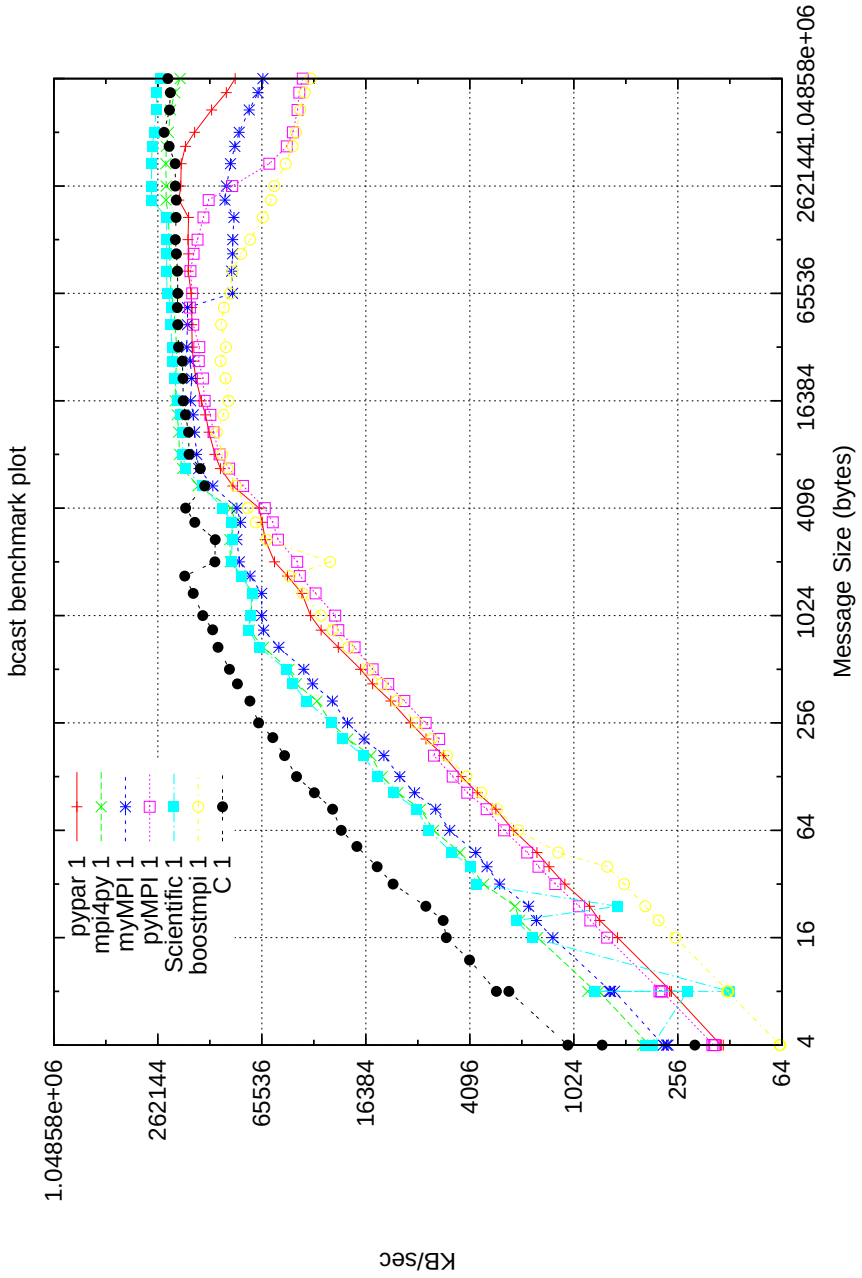


Figure 5.1: Performance comparison between six Python MPI modules layered MPI and C-version MPI on a Linux-cluster, with bcast test

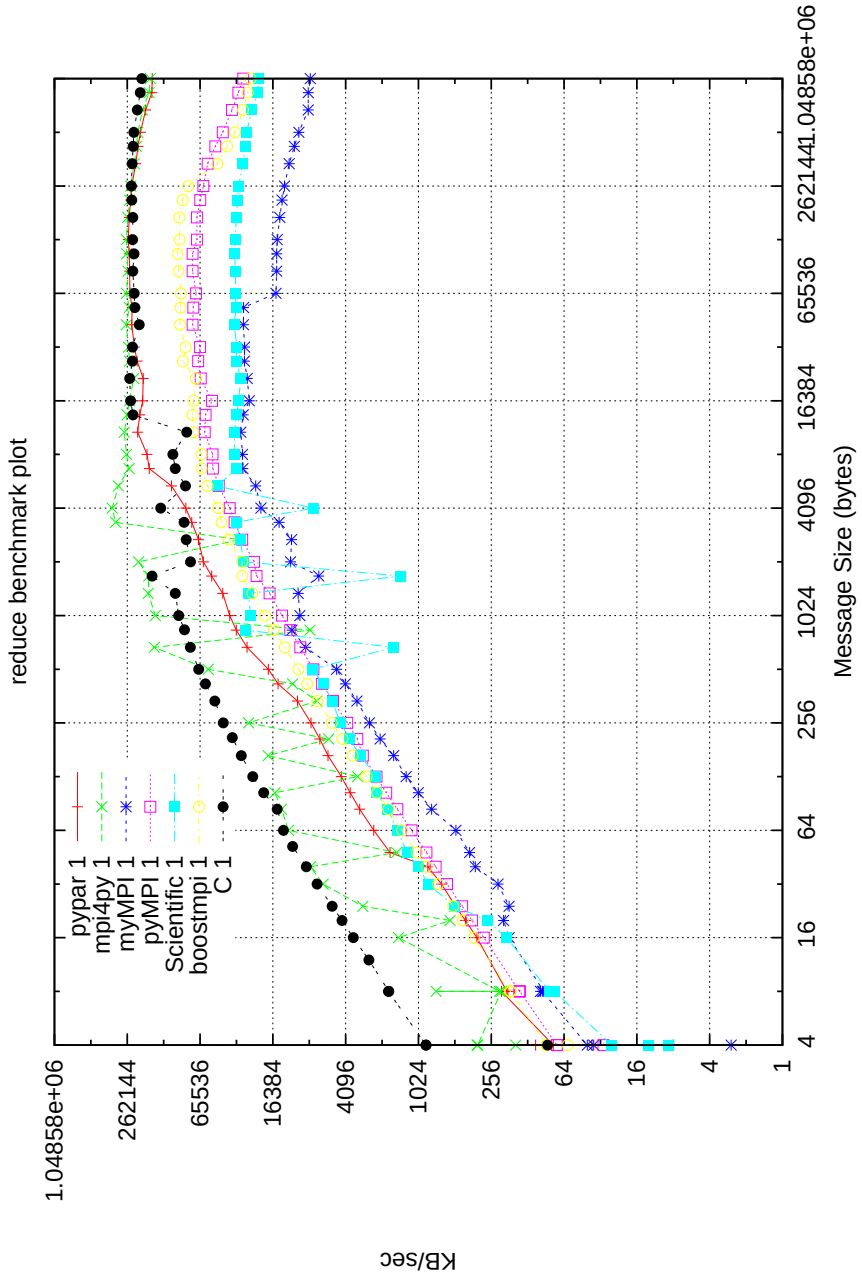


Figure 5.2: Performance comparison between six Python MPI modules layered MPI and C-version MPI on a Linux-cluster, with reduce test

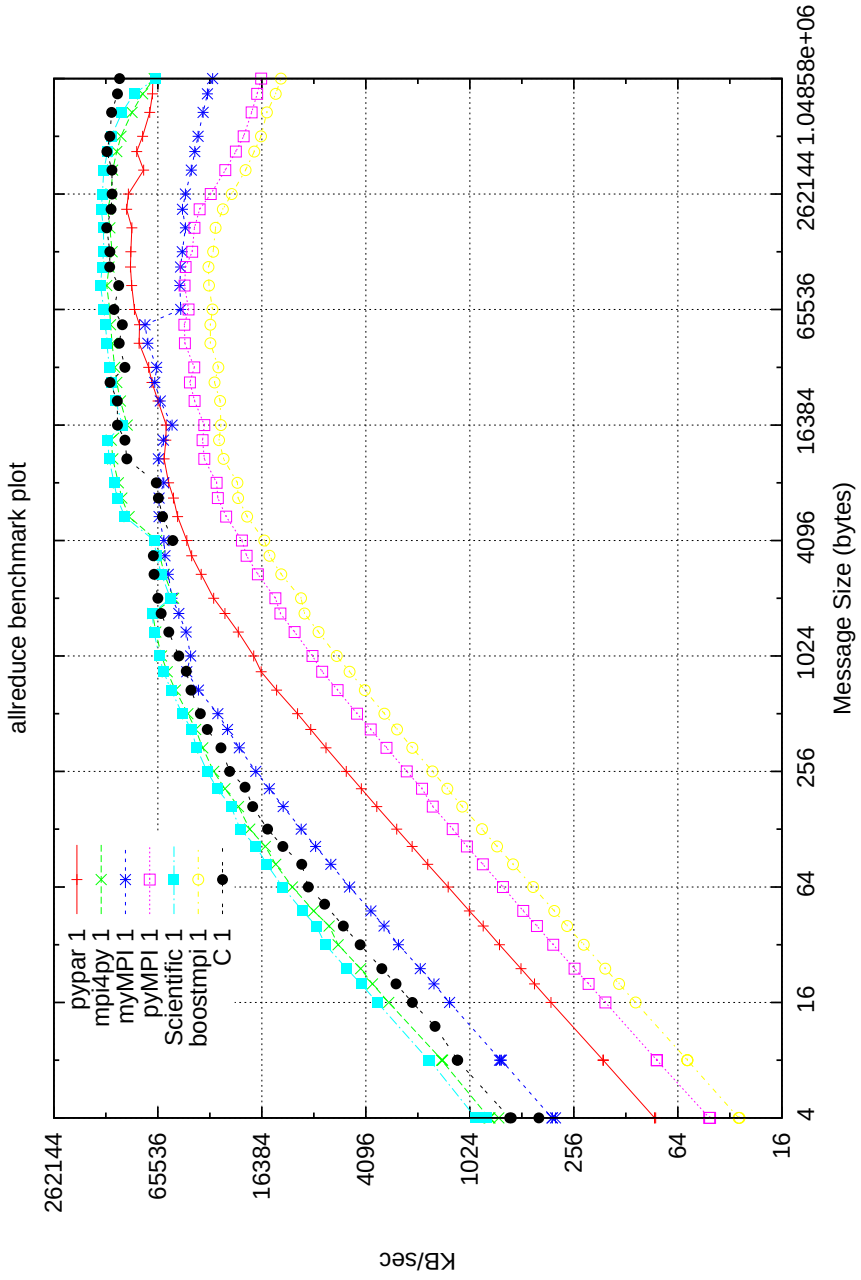


Figure 5.3: Performance comparison between six Python MPI modules layered MPI and C-version MPI on a Linux-cluster, with allreduce test

5.3 Conclusion

In the chapter, some MPI routines are tested by micro-benchmarks. The experimental results show that two of these six Python MPI modules, *mpi4py* and *Scientific.MPI* have comparatively good performance in both point-to-point and collective communications. In the next chapter, we will use these routines to solve two scientific problems. Then we can see the actual performance of these modules in real scientific applications.

Chapter 6

Test Cases and Numerical Experiments

In this chapter, the six Python MPI modules from the previous chapters will be used in parallel programming to solve two real scientific problems. Our focus will be on evaluating and comparing the flexibility and performance of Python implementations with the MPI modules. Detailed programming issues in using those modules will be discussed.

The upcoming numerical experiments of parallel Python implementations are performed on the Linux cluster described in Section 1.3 and detailed information and settings for each package are listed in Appendix A.

6.1 Application I: Solving Wave Equation in 3-Dimensional Space in Parallel

The wave equation is an example of hyperbolic partial differential equation of waves. The problem we solved is defined in three-dimensional space and solved with finite difference methods in both spatial and temporal directions. The problem is briefly described in next section. After the problem description, a parallel algorithm is generated in Section 6.1.2. In Section 6.1.3, the parallel algorithm is implemented in six Python MPI modules. We will discuss the performance issues according to the experimental profiling results in Section 6.1.4.

6.1.1 Problem description

Consider a simple mathematical model for three-dimensional initial-boundary value problem for the wave equation on a space-time domain $\Omega_T := \Omega \times (0, T]$

$$\frac{\partial^2 u(\mathbf{x}, t)}{\partial t^2} = c^2 \nabla^2 u(\mathbf{x}, t) \quad \text{in} \quad \Omega \tag{6.1}$$

$$\vec{n} \cdot \nabla u(\mathbf{x}, t) = 0 \quad \text{on} \quad \partial\Omega \tag{6.2}$$

where c is a constant representing the propagation speed of the wave, coordinate $\mathbf{x}$ is in three space dimensions, and $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$. The mathematical model Equation 6.1 and Equation 6.2 can have initial value conditions:

$$\frac{\partial u(\mathbf{x}, 0)}{\partial t} = 0 \quad \text{at} \quad t = 0, \quad \mathbf{x} \quad \text{in} \quad \Omega \tag{6.3}$$

$$u(\mathbf{x}, 0) = I(\mathbf{x}) \tag{6.4}$$

Here we apply finite difference method on uniform spacial computational mesh with constant cell lengths $\Delta x = \frac{1}{NX-1}$, $\Delta y = \frac{1}{NY-1}$, $\Delta z = \frac{1}{NZ-1}$. The mathematical model Equation 6.1 and Equation 6.2 can be translated to the following explicit scheme for one time-step forward in three dimensions:

$$\begin{aligned} u_{i,j,k}^{l+1} = & \frac{c^2\Delta t^2}{\Delta x^2}(u_{i+1,j,k}^l + u_{i-1,j,k}^l) \\ & + \frac{c^2\Delta t^2}{\Delta y^2}(u_{i,j+1,k}^l + u_{i,j-1,k}^l) \\ & + \frac{c^2\Delta t^2}{\Delta z^2}(u_{i,j,k+1}^l + u_{i,j,k-1}^l) \\ & - 2\Big(\frac{c^2\Delta t^2}{\Delta x^2} + \frac{c^2\Delta t^2}{\Delta y^2} + \frac{c^2\Delta t^2}{\Delta z^2} - 1\Big)u_{i,j,k}^l - u_{i,j,k}^{l-1} \end{aligned} \tag{6.5}$$

with subscripts $i = 1, \ldots, NX$, $j = 1, \ldots, NY$, $k = 1, \ldots, NZ$,
and the superscript $l = 0, 1, \ldots$ refer to the time level. Due to the stability reason, Δt is set to be no greater than $c\Big(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\Big)^{-\frac{1}{2}}$, according to observation based on some numerical calculation in book [28].

From the initial value conditions (6.3), we have $u_{i,j,k}^{-1} = u_{i,j,k}^1$, then the artificial quantity

$$\begin{aligned} u_{i,j,k}^{-1} = & \frac{c^2\Delta t^2}{2\Delta x^2}(u_{i+1,j,k}^l + u_{i-1,j,k}^l) + \frac{c^2\Delta t^2}{2\Delta y^2}(u_{i,j+1,k}^l + u_{i,j-1,k}^l) \\ & + \frac{c^2\Delta t^2}{2\Delta z^2}(u_{i,j,k+1}^l + u_{i,j,k-1}^l) - \Big(\frac{c^2\Delta t^2}{\Delta x^2} + \frac{c^2\Delta t^2}{\Delta y^2} + \frac{c^2\Delta t^2}{\Delta z^2} - 1\Big)u_{i,j,k}^l \end{aligned} \tag{6.6}$$

According to the boundary condition (6.2), the ghost grids[28] are formulated as:
$u^l_{0,j,k} = u^l_{2,j,k}$ and $u^l_{NX+1,j,k} = u^l_{NX-1,j,k}$
$u^l_{i,0,k} = u^l_{i,2,k}$ and $u^l_{i,NY+1,k} = u^l_{i,NY-1,k}$
$u^l_{i,j,0} = u^l_{i,j,2}$ and $u^l_{i,j,NZ+1} = u^l_{i,j,NZ-1}$

6.1.2 Parallelization

Regarding parallelization, the starting point is a partition of the global computational work among the processes. For this type of explicit finite difference schemes, we will divide the work based on sub-domains, where the computational work on one sub-domain is assigned to one process. The global solution domain Ω is decomposed into $P = Dim_x \times Dim_y \times Dim_y$ [1] sub-domains for three dimensional cases. Therefore, each sub-domain will have certain number of neighboring sub-domains. The numbers of neighbors varies from three to six with respect to the sub-domain positions.

We see Figure 6.1 as a two-dimensional example of partitioning the global solution domain on four processes. The blue parts are the outer boundary and the green parts are the internal boundary. The yellow parts are defined for ghost grids. The area of interior points are marked as red. For each sub-domain, the values of the ghost grids (marked with yellow)are updated by communicated with the internal boundary points (marked with green) of neighboring sub-domains. And the values of the interior points (marked with red) and the values of the grid points on the outer boundary (marked with blue) of each sub-domain are updated separately in serial.

The problem is devided into P small problems. For each sub-problem, the mesh size is reduced from $NX \times NY \times NZ$ to $Lx \times Ly \times Lz$ where $Lx \approx NX/Dims_x$, $Ly \approx NY/Dims_y$ and $Lz \approx NZ/Dims_z$. For each small sub-problem, the computational part for updating values of interior and boundary points can be done simultaneously. The main algorithm is in Algorithm 1 on the next page.

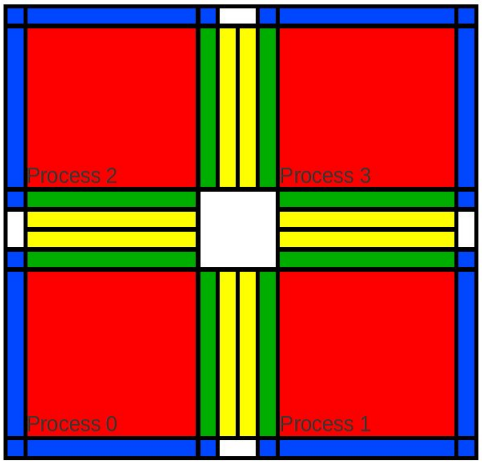
In the next section, we will implement the above algorithm and carry on some measurements and performance analysis in Section 6.1.4.

6.1.3 Implementation

In this section, six versions of Python MPI implementations with different packages are presented to solve the problem defined in Section 6.1.1. (Refer to the source code [16] for the full implementation details). From Algorithm 1 we can see

[1] From the implementation point of view, P is the number of processes which the parallel application is applied on.

Figure 6.1: A 2D example of partitioning the global domain on four processes.



Algorithm 1 The Parallel 3D Wave Equation Solver

1: Partitioning the global sub-domains into P sub-domains
2: Setting initial conditions
3: Setting boundary conditions
4: **while** $t <= tstop$ **do**
5: Updating the values of the grid points on the outer boundary
6: Exchanging the values of the ghost points on the internal boundary with neighboring processes
7: Updating the values of the interior points of each sub-domain
8: $t+ = dt$
9: **end while**

that the implementation for each time step in while-loop consists of three stages. The first and third stages, referring to the fifth and seventh lines in Algorithm 1, are the computational part for each sub-domain. They are carried out sequentially on the assigned process. Since no communication is involved in this part, those six versions call the same functions `update_inner` and `update_boundary`. On the sixth line in Algorithm 1, the second stage is the parallel part defined for exchanging the values of the ghost grids on the internal boundary between neighboring sub-domains. Six different versions of `exchange_boundary` are implemented for this part. For both parts, a C-version of implementation is also provided for comparison.

The Computational Part

The function `update_inner` is defined to calculate Equation 6.5. For Equation 6.6, the similar calculation is implemented with different coefficients.

Python version of ***`update_inner`***:

```
def update_inner(un,u,up,dx2,dy2,dz2,Lx,Ly,Lz):
    un[1:Lz+1,1:Ly+1,1:Lx+1]=\
        dz2*(u[2:Lz+2,1:Ly+1,1:Lx+1]+u[0:Lz,1:Ly+1,1:Lx+1])\
        +dy2*(u[1:Lz+1,2:Ly+2,1:Lx+1]+u[1:Lz+1,0:Ly,1:Lx+1])\
        +dx2*(u[1:Lz+1,1:Ly+1,2:Lx+2]+u[1:Lz+1,1:Ly+1,0:Lx])\
        -2*(dz2+dy2+dx2-1)*(u[1:Lz+1,1:Ly+1,1:Lx+1])\
        -up[1:Lz+1,1:Ly+1,1:Lx+1]
    return un
```

Here, the three-dimensional arrays `un, u, up` refer to u^{l+1}, u^l and u^{l-1}, respectively. The variables `dx2, dy2, dz2` contains the values $c^2\Delta t^2/\Delta x^2$, $c^2\Delta t^2/\Delta y^2$ and $c^2\Delta t^2/\Delta z^2$, respectively. The problem size for each sub-domain is defined as $Lx \times Ly \times Lz$ in the previous section.

From this Python version of `update_inner`, we can see that array slicing[3] is heavily used for three-dimensional arrays. Slicing is considered to be one of the best features of Python's NumPy package for scientific programming, which returns a pointer to the original values. This feature greatly simplifies the traditional approach to access multi-dimensional arrays which is shown in the C version of `update_inner`.

C version of ***`update_inner`*** *with three nested for loops:*

```
#define IJKth(u,i,j,k) (u[(i)+(j)*(lxyz[0]+2)+(k)*(lxyz[1]+2)*(lxyz[0]+2)])
void update_inner(double dx2,double dy2,double dz2, int lxyz[3],
                  double* un, double *u, double *up) {
  int i,j,k;
  for (k=1;k<=lxyz[2];k++){
```

```
    for(j=1;j<=lxyz[1];j++){
      for(i=1;i<=lxyz[0];i++){
        IJKth(un,i,j,k)=(dz2*(IJKth(u,i,j,k+1)+IJKth(u,i,j,k-1)) \
                      +dy2*(IJKth(u,i,j+1,k)+IJKth(u,i,j-1,k)) \
                      +dx2*(IJKth(u,i+1,j,k)+IJKth(u,i-1,j,k)) \
                     -2*(dx2+dy2+dz2-1)*(IJKth(u,i,j,k))\
                      -IJKth(up,i,j,k));
      }
    }
  }
}
```

Here, `lxyz[3]` is defined for the problem size of the sub-domain Lx, Ly and Lz. One micro `IJKth` is defined for index operations of `u, up, un` in 3-dimension for replacement of the long and repeat fragments of codes.

Communication Part

The communication part is mainly in function `exchange_boundary` for exchanging internal boundary values with neighboring sub-domains. Each sub-domain is adjacent with three to six other sub-domains depending on the position in global solution domain. One exchanging operation is needed for each neighboring sub-domain. We take the exchanging operations in x-direction as an example to show the communication procedure. The y and z directions have the same exchanging pattern.

In x-direction, the adjacent surfaces between sub-domains are exchanged, namely a 2-dimensional arrays are sent to or received from left and right neighbors.

This procedure is implemented with six different Python MPI modules. Different interfaces for `MPI_Send` and `MPI_receive` calls will be presented. And some detailed implementation issues will also be discussed in this section. We will first see a C-version of the exchange operations in x-direction to illustrate the MPI send-receive routines in the traditional approaches with the compiled language.

C version of ***exchange_boundary:***

```
MPI_Type_vector((lxyz[1]+2)*(lxyz[2]),1,(lxyz[0]+2),MPI_DOUBLE,&new_type_x);
MPI_Type_commit (&new_type_x);
MPI_Sendrecv(&(IJKth(u,pos_x[0],1,1)),1,new_type_x,ranks_x[0],tags_x[0],
                 &(IJKth(u,pos_x[1],1,1)),1,new_type_x,ranks_x[1],tags_x[1],
                 MPI_COMM_WORLD,&status);
MPI_Sendrecv(&(IJKth(u,pos_x[2],1,1)),1,new_type_x,ranks_x[2],tags_x[2],
```

```
7                 &(IJKth(u,pos_x[3],1,1)),1,new_type_x,ranks_x[3],tags_x[3],
8                  MPI_COMM_WORLD,&status);
```

The parameters `lxyz,ranks_x,tags_x,pos_x` are defined for the size of sub-domain, the neighboring processes, the message tags and the position of the send-bufs, respectively. They are all calculated before the exchanging operations. For transmitting a non-contiguous memory buffer, a derived data type `new_type_x` is defined to index the two-dimensional array on the adjacent surface between two neighboring sub-domains.

The following six implementations with different Python MPI modules are carried out to achieve the same communication procedure. But there are some differences in using buffer, return values and checking boundary processes. We will see these differences from some small segments of the source codes.

The C-MPI provides a send-recv combination routine `MPI_Sendrecv`, we use it for this exchanging operation to avoid deadlock. For those Python MPI modules without this routine, we need to define a "even-odd" (decided by the position of the sub-domain) model for passing the messages. It is defined as the "odd" processes choose to send while the "even" processes receive, followed by a reversal order. For those Python MPI modules without `MPI_PROC_NULL` implemented, we need to check whether the process is null for each sending or receiving operation.

An example of exchanging operations with *pypar*

Here, we take the *Pypar* version as an example to describe the procedure.

We defined six functions for the exchanging operations in x-direction. `x1` and `x2` are the neighboring processes. In *pypar* version, we set optional parameter `use_buffer` to `True` for avoiding re-allocate places for buffer and set `bypass` to `True` for avoiding the extra work of error checks. Both parameters are optional, and the default value are `False`. However, this usage will greatly improve the parallel performance while dealing with large numerical arrays.

pypar version of the exchanging operations in x direction

```
1  def sendrecv_x_even(myrank,lx,ly,lz,x2,buf_x):
2      pypar.send(u[1:lz+1,1:ly+1,lx],x2, use_buffer=True , bypass=True)
3      u[1:lz+1,1:ly+1,lx+1]=pypar.receive(x2, buffer=buf_x, bypass=True)
4  def sendrecv_x_even_null(myrank,lx,ly,lz,x2,buf_x):
5      pass
6  def sendrecv_x_odd(myrank,lx,ly,lz,x1,buf_x):
7      u[1:lz+1,1:ly+1,0]   =pypar.receive(x1, buffer=buf_x, bypass=True)
8      pypar.send(u[1:lz+1,1:ly+1,1], x1, use_buffer=True , bypass=True)
9  def sendrecv_x_odd_null(myrank,lx,ly,lz,x1,buf_x):
10     pass
11 def comm_x_1(myrank,lx,ly,lz,x1,x2,buf_x):
12     sendrecv_even(myrank,lx,ly,lz,x2,buf_x)
13     sendrecv_odd(myrank,lx,ly,lz,x1,buf_x)
```

```
def comm_x_2(myrank,lx,ly,lz,x1,x2,buf_x):
    sendrecv_odd(myrank,lx,ly,lz,x1,buf_x)
    sendrecv_even(myrank,lx,ly,lz,x2,buf_x)
```

We apply function pointers to predefine the send-recv order, the communication pattern and null processes, so that we only need to define them only once in our program before the exchanging operation. After that, we don't need to check the null processes and the send-recv orders anymore. The following is the example to define the function pointers. This procedure is also used in the other implementation which the module does not provide `MPI_Sendrecv` and implement `MPI_PROC_NULL`.

```
comm_x=comm_x_1 if np[0]%2==0 else comm_x_2 # to check the even/odd process
sendrecv_x_odd=sendrecv_x_odd if x1!=-2 else sendrecv_x_odd_null
sendrecv_x_even=sendrecv_x_even if x2!=-2 else sendrecv_x_even_null
comm_x(myrank,lx,ly,lz,x1,x2,buf_x) # do exchange operation in x-direction
```

An example of exchanging operations with *mpi4py*

Following we see the implementation with *mpi4py*. *mpi4py* has both `MPI_Sendrecv` and `MPI_PROC_NULL` implemented. Therefore this version of exchanging operations has the almost same pattern with C-MPI version.

mpi4py version of the exchanging operations in x direction

```
new_type_x=mpi.DOUBLE.Create_vector((Ly+2)*(Lz+2),1,(Lx+2))
newtype_x.Commit()
ut=u.reshape(-1)#convert to 1-d array
comm.Sendrecv([ut[(pos_x[0]):],1,new_type_x],ranks_x[0],tags_x[0],\
[ut[(pos_x[1]):],1,new_type_x],ranks_x[1],tags_x[1])
comm.Sendrecv([ut[(pos_x[2]):],1,new_type_x],ranks_x[2],tags_x[2],\
[ut[(pos_x[3]):],1,new_type_x],ranks_x[3],tags_x[3])
```

Derived data type is specially needed for this module. The starting positions of arrays, neighboring ranks and tags are calculated in advance for transmitting the non-contiguous memory buffers.

An example of one exchanging operation with *MYMPI*

The *MYMPI* version has the same procedure with pypar version. The codes below only shows the message-passing to the right neighbor, the other directions are quite similar but with correct positions of sendbuf, corresponding tags and destination ranks. The version looks like in between *pypar* and *mpi4py* . Slicing is supported but the user needs to specify the size and data type of the buffer.

MYMPI version of one exchanging operation in x direction

```
mpi.mpi_send(u[1:Lz+1,1:Ly+1,Lx],size_x,mpi.MPI_DOUBLE,x2,\
                        tags_x[0],mpi.MPI_COMM_WORLD)
```

```
u[1:Lz+1,1:Ly+1,Lx+1]=array(mpi.mpi_recv(size_x,mpi.MPI_DOUBLE,\
x2,tags_x[1],mpi.MPI_COMM_WORLD)).reshape(Lz,Ly)
```

There are two small details that need to pay attention to. One is that the buffer received is not numerical array, and need to convert to numerical arrays manually and reshape to fit in the correct position. The other is that `MPI_PROC_NULL` is not implemented as it is in C version. If the destination is `MPI_PROC_NULL`, the sending will run anyway. But the receiver receives an array with all zero values. It will end up to wrong calculation results without any MPI error messages.

An example of one exchanging operation with *pyMPI*

The *pyMPI* version also has the same procedure with *pypar* version. The codes below only shows the message-passing to the right neighbor, the other directions are quite similar but with correct positions of sendbuf and corresponding destination ranks.

pyMPI version of one exchanging operation in x direction

```
mpi.send(u[1:Lz+1,1:Ly+1,Lx],x2)
u[1:Lz+1,1:Ly+1,Lx+1],status=mpi.recv(x2)
```

The module provides `MPI_Sendrecv` routine, so we could also use the function `mpi.sendrecv` instead.

An example of one exchanging operation with *Scientific.MPI*

The *Scientific.MPI* version has the same procedure with *pypar* version and similar syntax with *pyMPI*. The codes below only shows the message-passing to the right neighbor, the other directions are quite similar but with correct positions of sendbuf and corresponding destination ranks.

Scientific.MPI version of one exchanging operation in x direction

```
comm.send(u[1:Lz+1,1:Ly+1,Lx],x2,100 )
u[1:Lz+1,1:Ly+1,Lx+1], source, tag, count=comm.receive(buf_x,x2)
```

We remark that the tag is of type integer and could not be great than 2^{31}. And the returned values of function `comm.receive` is a tuple including the reference of the received buffer, the source process, the message tag and the size of the received buffer.

An example of one exchanging operation with *boostmpi* The boostmpi version has the same procedure with *pypar* version. The codes below only shows the message-passing to the right neighbor, the other directions are quite similar but with correct positions of sendbuf and corresponding destination processes.

boostmpi version of one exchanging operation in x direction

```
comm.send(value=u[1:Lz+1,1:Ly+1,Lx],x2)
u[1:Lz+1,1:Ly+1,Lx+1]=comm.recv(source=x2)
```

We remark that the received value is returned instead of storing it in a pre-allocated buffer. This may have some impact on the parallel performance.

6.1.4 Measurements and performance comparison

We have chosen a test problem with the wave speed $c = 1$ and initial values

$$I(\mathbf{x}) = 2\cos(\pi x)\cos(\pi y)\cos(\pi z) \tag{6.7}$$

Two global uniform meshes are defined in Table 6.1. We chose the mesh size as the power of 2 and one grid point in addition ($2^7 + 1 = 129$ and $2^8 + 1 = 257$) to reduce the roundoff error. The size and the number of time steps are independent of the number of processes, only determined by the global mesh size. Figure 6.2 - 6.5 shows the performance comparison of the wall-clock time measurements and speedup results among seven different implementations: pypar, mpi4py, myMPI, pyMPI, Scientific.MPI, boostmpi and C. The experimental environments are described in Appendix A. All the measurements in this section measures the wall-clock time (in seconds) of the time-stepping part for solving the wave equation. The number of processes is varied between 1 and 64. The speedup results use the measurements on single core (free of communication) as the reference. The measurements are labeled respectively with different colors and point types. For the purpose of better illustrating, both x and y axes are scaled with $\log 2$.

Mesh size	$129 \times 129 \times 129$	$257 \times 257 \times 257$
The size of time step	0.004510	0.002255
The number of time steps	444	887

Table 6.1: Two global meshes and time steps defined for solving the wave equation

We can observe from Figure 6.2 - 6.5 that C implementation is 8 to 10 times faster than any of the implementations with Python MPI wrapper. The overhead of Python MPI wrapper is one of the reasons, as we have discussed in the previous chapter. But we notice that the measurements with the program executed on a single core also have this kind of distribution, even without communication overhead. From Table 3 in paper [5], we see that the stand-alone C program is more than 9 times faster than vectorized Python version slices provided by numerical python modules, while updating a seven-point finite difference stencil on a $100 \times 100 \times 100$ grid mesh. Vectorized Python use *slicing* provided by numerical Python modules implemented in C. Because of the large computation/communication ratio, the most time-consuming part for solving this problem is the computation part.

In Figure 6.2, the wall-clock time measurements show slightly different from each other, while the number of processes is as large as 16. The computation load on each process became comparatively smaller, then the difference may caused by the communication overhead. The communications are mainly on

boundary exchanging operations. In connection with the performance comparison of pingpong test in the previous chapter, we can see that the module with large latency have relatively bad performance in this case.

In Figure 6.3, the speed-up results show that the implementations are no longer efficient while the number of processes is larger than 32. The communication time took the most part of the overall time comsumption for solving this problem.

In Figure 6.4, the wall-clock time measurements of six different versions almost draw on the same line. The large computation/communication ratio of this test case makes the differences of the communication overhead from different modules have small influence on the overall parallel performance.

However, the speed-up results on the mesh size $257 \times 257 \times 257$ illustrates that all the seven implementations have almost same speedup results. These Python MPI modules provide sufficient parallel efficiency regarding to Python MPI programming for solving scientific problems.

Figure 6.2: A comparison of the wall-clock time measurements (in seconds) of solving the wave equation on mesh $129 \times 129 \times 129$ among seven different implementations: pypar, mpi4py, myMPI, pyMPI, Scientific.MPI, boostmpi and C

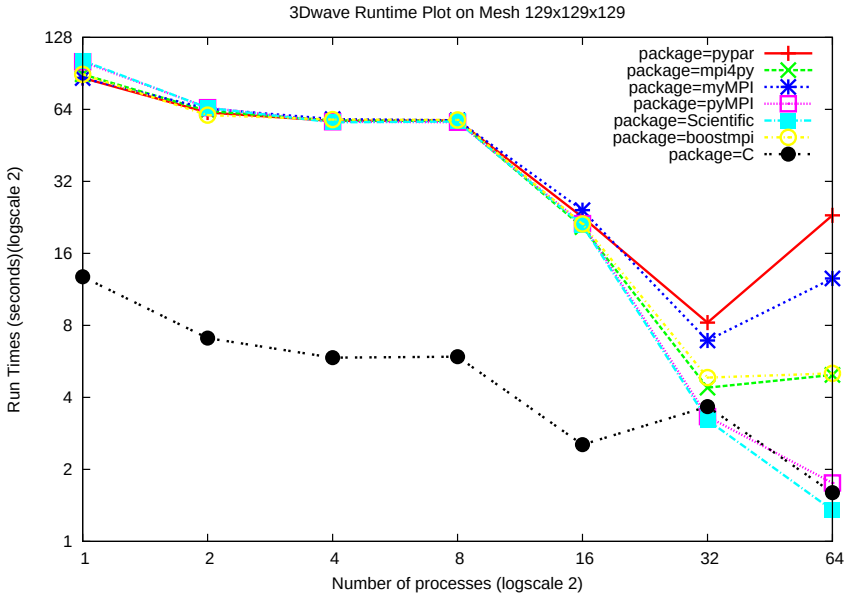


Figure 6.3: A comparison of the speedup results of solving the wave equation on mesh $129 \times 129 \times 129$ among seven different implementations: pypar, mpi4py, myMPI, pyMPI, Scientific.MPI, boostmpi and C

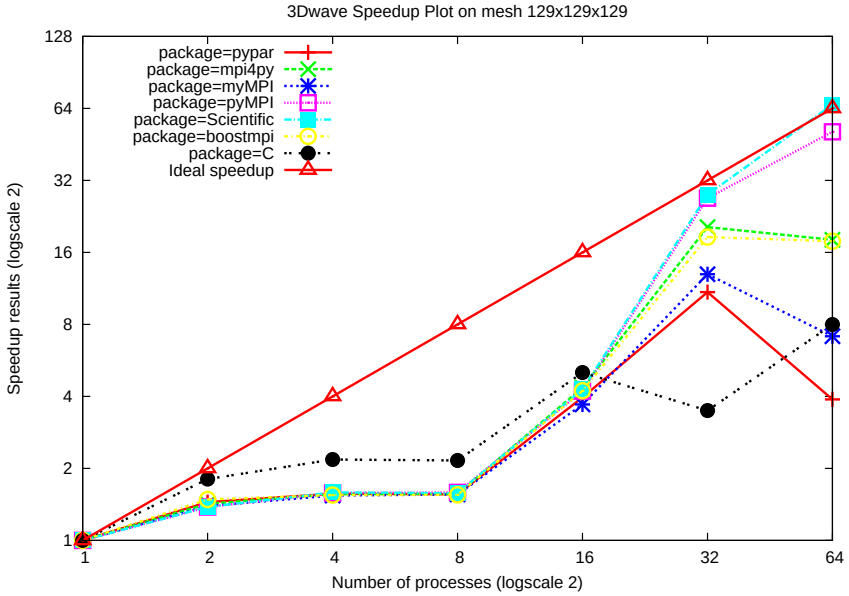


Figure 6.4: A comparison of the wall-clock time measurements (in seconds) of solving the wave equation on mesh $257 \times 257 \times 257$ among seven different implementations: pypar, mpi4py, myMPI, pyMPI, Scientific.MPI, boostmpi and C

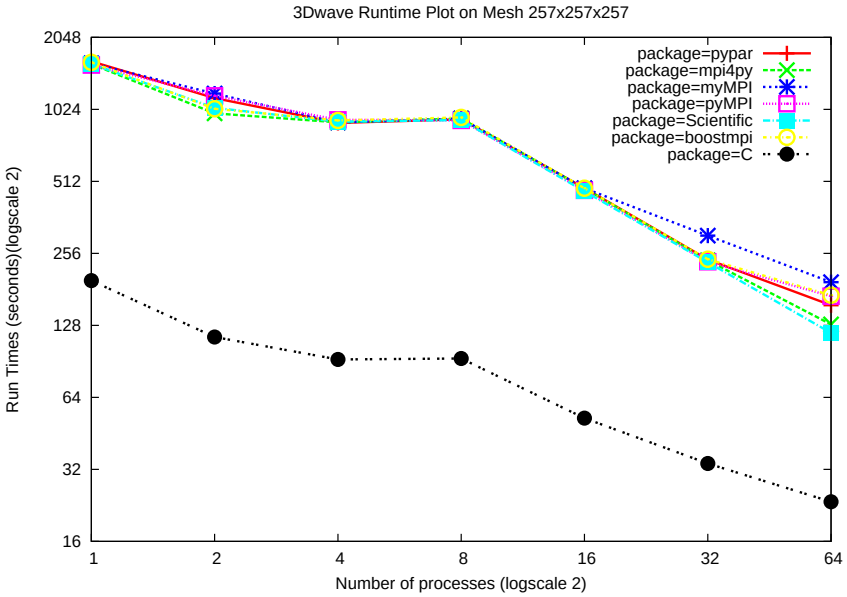
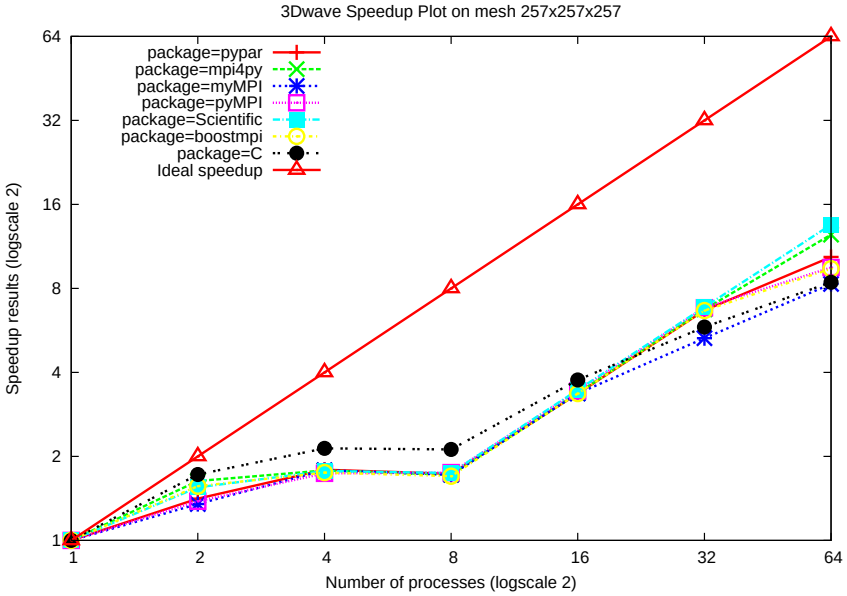


Figure 6.5: A comparison of the speedup results of solving the wave equation on mesh $257 \times 257 \times 257$ among seven different implementations: pypar, mpi4py, myMPI, pyMPI, Scientific.MPI, boostmpi and C



One Python implementation with *mpi4py* non-blocking communication is also implemented for comparison. The same environmental conditions are used for testing. In Figure 6.6, we can see the non-blocking version has better performance than the blocking version. The large computation takes the advantage of the waiting time for non-blocking communications. However, In Figure 6.7, the advantage of non-blocking communication is not obvious any more. Even on some point, blocking version is slightly faster than the non-blocking version. For more discussion about the overlapping communication and computation mode can be found in most of the MPI tutorials, i.e.[18].

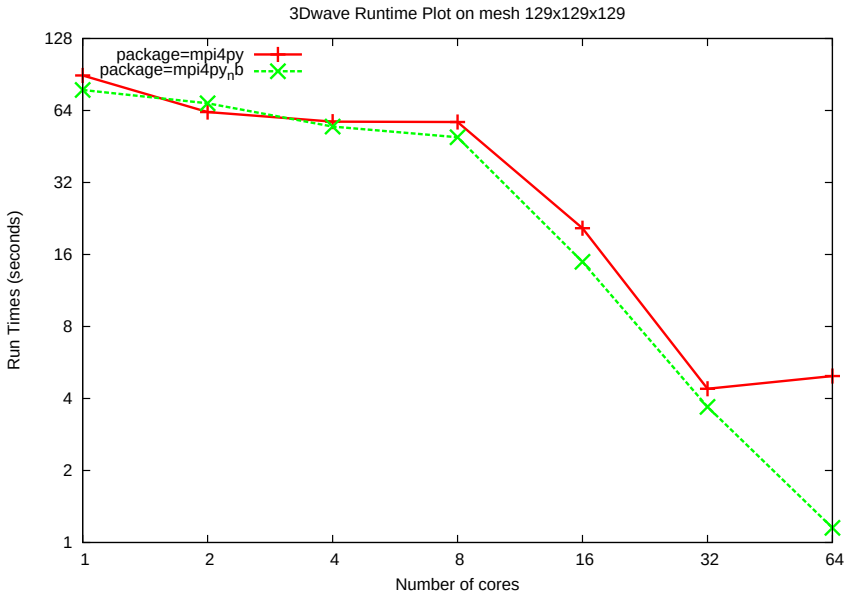


Figure 6.6: A comparison of the wall-clock time measurements (in seconds) of solving the wave equation on mesh $129 \times 129 \times 129$ between blocking and non-blocking communications implemented with the module *mpi4py*

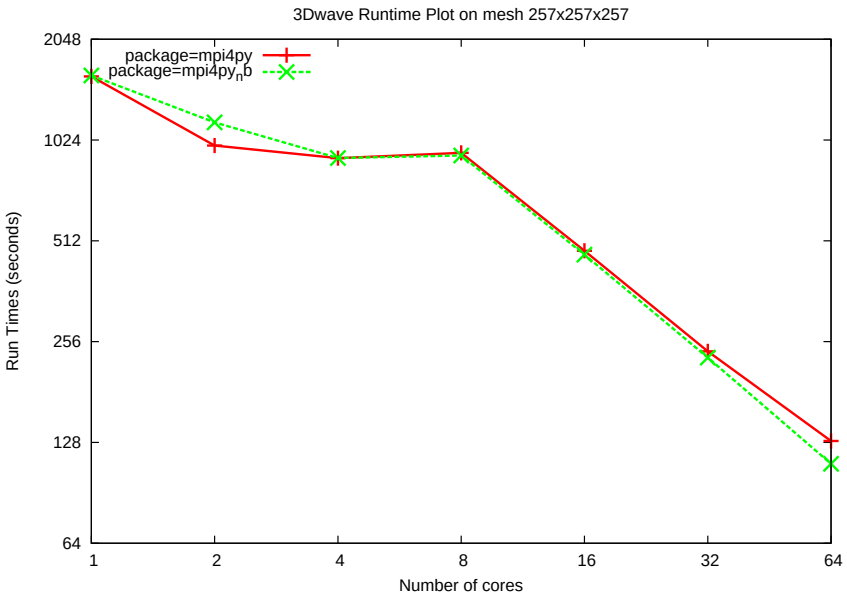


Figure 6.7: A comparison of the wall-clock time measurements (in seconds) of solving the wave equation on mesh $257 \times 257 \times 257$ between blocking and non-blocking communications implemented with the module *mpi4py*

6.2 Application II: Parallel Implementation of Conjugate Gradient Method

In computational mathematics, while numerically solving PDEs by finite difference methods or finite element methods, it always leads to a large system of linear algebraic equations. Iterative methods[29] play an important role in solving those large linear systems. In contrast to direct methods(like Gaussian elimination[30]), iterative methods achieve better performance in computational efficiency. One of the best known iterative methods is the Conjugate Gradient(CG) Method[31], which is widely used for finding successive approximations to the solutions of symmetric positive-definite systems. Here in this section, we introduce a parallel algorithm of Conjugate Gradient Method and implement it with six different Python MPI modules. Then we apply them on solving the Poisson's equation on two-dimensional space for experiments and performance comparison.

Through the implementations, we compare the usages and functionalities of these Python MPI modules, and measure the parallel performances on a Linux cluster described in Appendix A. The comparisons of parallel performances are intuitively illustrated in chart form.

6.2.1 The Parallel Algorithm of Conjugate Gradient Algorithm

Suppose we want to solve the system of linear equations

$$Ax = b \tag{6.8}$$

where A is a n-by-n symmetric[1] and positive definite[2] sparse[3] matrix, and b is a vector of length n. For a given matrix/vector partitioning scheme(i.e.row partitioning scheme), the vectors b and x is partitioned among computing processes. The distribution of the matrix A should be consistent with the distribution of the vectors. We define A_s as a sub-matrix of A, b_s as a sub-vector of b and x_s as a sub-vector of x on one process. x_{0s} is defined as a sub-vector of the initial vector x_0. Remarking x_{0l} and p_l are extra vectors prepared for matrix-vector multiplication. According to the partitioning scheme, on each process, only a segment of the original vector is stored. In order to do matrix-vector multiplication, all non-zero entries in the sub-matrix should be multiples with the corresponding elements of the global vector. Some more elements are needed besides the elements

[1]Symmetric matrix: $A^T = A$. See more about symmetric matrix in [32]

[2]Positive definite matrix: $x^T Ax > 0$ for all non-zero vectors $x \in R^n$. See more about symmetric matrix in [33]

[3]Sparse matrix: a matrix that allows special techniques to take advantage of the large number of zero elements[34].

in the sub-vector. Therefore x_{0l} and p_l are prepared as the local vectors with all the elements needed for one multiplication operation with the corresponding sub-matrix.

The main content of the parallel algorithm of Conjugate Gradient Method is shown in Algorithm 2. The red marked parts in the algorithm involve MPI-based parallel implementation, which will be explained in Section 6.2.2

6.2.2 Implementation

In this section, six versions of Python implementations of the Algorithm 2 are presented with different Python MPI modules. We can see that in each iteration step, three matrix/vector operations are involved, matrix-vector multiplication, vector inner product, vector addition and scalar-vector multiplication. In the following text, we will explain them one by one with emphasize on the matrix-vector multiplication and the vector inner product which involve message-passing communications. The main differences among those six implementations are in the message-passing communication part. Before that, the sparse storage format used in the implementations is introduced first.

The sparse matrix storage format

In chapter 2, we have introduced the scientific tools for Python *SciPy*[23]. *SciPy* provides seven sparse matrix types[4] for different purposes. For more explanations about the types, the readers are referred to the reference guide[23]. Among those seven types, Compressed Sparse Column format(CSC) and Compressed Sparse Row format(CSR) are specially defined for performing matrix multiplication such as matrix-vector multiplication efficiently. Therefore we choose CSR formats in related to the given matrix/vector partitioning scheme(i.e. row partitioning scheme).

Matrix-vector multiplication

As we have explained that local vector p_l is necessary for matrix-vector multiplication with the corresponding sub-matrix A_s. On each process, besides the elements already locate in this sub-vector p_s, some more elements are needed from the other processes. These elements are obtained from the relevant processes according to the partitioning scheme. The function `local_value` are defined for this purpose. The procedure is the same for different implementations. However, the obtaining operation, which is performed by calling MPI routines, have some differences in using MPI send-recv function calls among different Python MPI

[4]Seven sparse matrix types: Compressed Sparse Column format, Compressed Sparse Row format, Block Sparse Row format, List of Lists format, Dictionary of Keys format, Coordinate format, and Diagonal format. [23]

Algorithm 2 The Parallel Conjugate Gradient Algorithm

1: Calculate $Ax = b$
2: $A_s \leftarrow$ **sub_matrix**(A), $bs \leftarrow$ **sub_vector**(b)
3: $x_{0s} \leftarrow b_s$ {initializing problem vector for first iteration step}
4: $x_{0l} \leftarrow$ **local_value**(x_{0s}) {preparing a local vector corresponding to each nonzero entry in sub-matrix by point-to-point communication with relevant processes containing certain elements A_s}
5: $r_s \leftarrow b_s -$ **matvec**(A_s, x_{0l}) {generating local residual r_s by multiply sub-matrix and corresponding local vector x_{0l} }
6: $p_s \leftarrow r_s$
7: $\rho_0 \leftarrow < r_s, r_s >$
8: $\rho_0 \leftarrow all\_reduce(\rho_0, op = sum)$
9: $k = 0$
10: **while** $\rho_k/\rho_0 > tolerance$ **do**
11: $p_l \leftarrow$ **local_value**(p_s)
12: $v_s \leftarrow$ **matvec**(A_s, p_l)
matrix-vector multiplication
13: $a1_s \leftarrow < r_s, r_s >$
vector inner product
14: $a1 \leftarrow$ **all_reduce**$(a1_s, operation = sum)$
15: $a2_s \leftarrow < p_s, v_s >$
vector inner product
16: $a2 \leftarrow$ **all_reduce**$(a2_s, operation = sum)$
17: $a \leftarrow a1/a2$
18: $x_s \leftarrow x_s + a * p_s$
vector addition and scalar-vector multiplication
19: $r_s \leftarrow r_s - a * v_s$
vector addition and scalar-vector multiplication
20: $\rho_{k+1} \leftarrow < r_s, r_s >$
vector inner product
21: $\rho_{k+1} \leftarrow$ **all_reduce**$(\rho_{k+1}, operation = sum)$
22: $\beta \leftarrow \rho_{k+1}/\rho_k$
23: $p_s \leftarrow r_s + \beta * p_s$
vector addition and scalar-vector multiplication
24: $k \leftarrow k + 1$
25: **end while**

modules. The detailed usage of MPI send-recv has been explained in the previous application case. Here we will take one of the six different implementations as an example to show how it works.

Python implementation of obtaining the missing elements, using `pypar`[10]

```
1 pypar.send(send_buf,dest,tag=tag1,use_buffer=True, bypass=True)
2 pypar.recv(source,buffer=recv_buf,tag=tag2)
```

For each process, the elements needed for the other processes are stored in `send_buf` and sent to the relevant processes. Then the missing elements are received from the corresponding processes and stored in `recv_buf`. Putting the received elements together with the sub-vector `p_s` already on the process, we get the local vector `p_l` ready for matrix-vector multiplication operation.

The matrix-vector multiplication operation is performed by calling the build-in function provided by `SciPy.sparse`.

```
1 vs=As._mul_vector(p_l)
```

Here `As` is the sub-matrix defined of CSR format in package `SciPy.sparse` and `vs` is a sub-vector for storing the matrix-vector multiplication results. For this computational part, the implementation is the same for all the six different versions, since no communication is involved. Refer to the source code for the full implementation details.

Vector inner product

Another important operation in Algorithm 2 is the vector inner product. The operation is performed three times in each iteration step. The procedure is almost the same with the example shown in Chapter 2. However, the only difference is here we need the global vector inner product results available on all processes instead of only on the root process. The reason is the results are needed for the upcoming matrix/vector operations. For instance, Line 15-18 in Algorithm 2, after calculating the inner product of two sub-vectors `p_s` and `v_s`, a MPI Allreduce function is called to sum the value `a2_s` from each process. The sum `a2` is returned to all processes. Then the global vector inner product result `a2` is available on each process for the oncoming vector addition and scale multiplication operations.

The Python implementation of vector inner product of two sub-vectors is the same of all six implementations.

```
1 a2_s=numpy.inner(p_s,v_s)
```

The six different implementations of summing the result on each process are presented next. For those modules who have not provided MPI Allreduce (*pypar*

and *MYMPI* [5]), we use a combination of MPI Reduce and MPI Bcast instead. The procedure is mainly the same, but we sum the value from each process to the master process first and broadcast it to all the other processes afterwards.

For some modules(*pypar*, *MYMPI* and *Scientific.MPI*), the collective operations do not support the arbitrary Python objects. The user needs to convert the value on each process to the type of numerical array first.

Python implementation of summing the global inner product result `a2`*, using pypar[10]*

```
1       a2s=array(a2s,'d')
2       a2=array(0.,'d')
3       pypar.reduce(a2s,pypar.SUM,0,buffer=a2)
4       pypar.bcast(a2,0)
```

For this case, the local value `a2s` contains only one element. We can use the reduction operation without specifying the pre-allocated buffer and taking the returned value instead. For instance,

```
1       a1=pypar.reduce(a2s,pypar.SUM,0)
```

Python implementation of summing the global inner product result `a2`*, using mpi4py[11]*

```
1         comm.Allreduce([a2s,mpi.DOUBLE],[ a2,mpi.DOUBLE], op=mpi.SUM)
```

In module mpi4py, this Allreduce is defined for transmitting numerical arrays. Alternatively, we can use the collective operations defined for arbitrary Python objects and take the returned values instead of specifying the pre-allocated buffer.

```
1         comm.allreduce(a2s,a2,op=mpi.SUM)
```

Python implementation of summing the global inner product result `a2`*, using MYMPI[12]*

```
1       a2s=array(a2s,'d')
2       a2=array(0.,'d')
3       a2=mpi.mpi_reduce(a2s,1,mpi.MPI_DOUBLE,mpi.MPI_SUM,0,comm)
4       a2=mpi.mpi_bcast(a2,1,mpi.MPI_DOUBLE,0,comm)
```

For module *MYMPI*, We remark that both the values returned by mpi_reduce and mpi_bcast are array type. For the upcoming operations, the user need to convert them to desired data types.

Python implementation of summing the global inner product result `a2`*, using pyMPI[13]*

```
1           a2=mpi.allreduce(a2s, mpi.SUM)
```

[5] see Table 4.7 for more information

For the pyMPI version, the syntax is quite simple and clear. The user don't need to pre-allocate a buffer or convert the types. We remark that the returned value has to be taken while using pyMPI.

Python implementation of summing the global inner product result `a2`*, using Scientific.MPI[14]*

```
a2s=array(a2s,'d')
a2=array(0.,'d')
comm.allreduce(a2s,a2,mpi.sum)
```

The user has to convert the local value `a2s` to numerical array and prepare another numerical array for received value while using collective communications provided by Scientific.MPI.

Python implementation of summing the global inner product result `a2`*, using boostmpi[15]*

```
        a2=boostmpi.all_reduce(comm,a2s,lambda x,y:x+y)
```

No collective operation is implemented in module boostmpi. The user has to define a function used as a collective operation for the collective communication routines in boostmpi.

Vector addition and scalar multiplication

For the computation part of vector addition and scalar multiplication, no communication is needed. We simply multiply a scalar with a sub-vector or add two sub-vectors.

6.2.3 Measurements and performance comparison

We choose a test problem, solving the Poisson's equation on two-dimensional space, for measure the parallel performances of six Python MPI modules.

$$\nabla^2 u(\mathbf{x}) = f(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in R^2 \tag{6.9}$$

The coefficient matrix of the Poisson's equation is constructed by applying central difference method on two dimensions. After performing some linear transformations to the coefficient matrix, we have a symmetric positive-definite matrix `A`. The matrix/vector partitioning scheme comes from the subdomain-based division on two-dimensional mesh with size $N \times N$. Therefore, the vector `x` and the vector `b` are of size N^2. The size of the matrix `A` is $N^2 \times N^2$. Figure 6.8 gives an example of the row partitioning scheme with mesh size 4×4 on four processes. The elements of one vector are partitioned on corresponding processes with the same marked colors and stored in a one-dimensional array. Then the rows of the matrix are assigned to different processes corresponding to the distribution of the

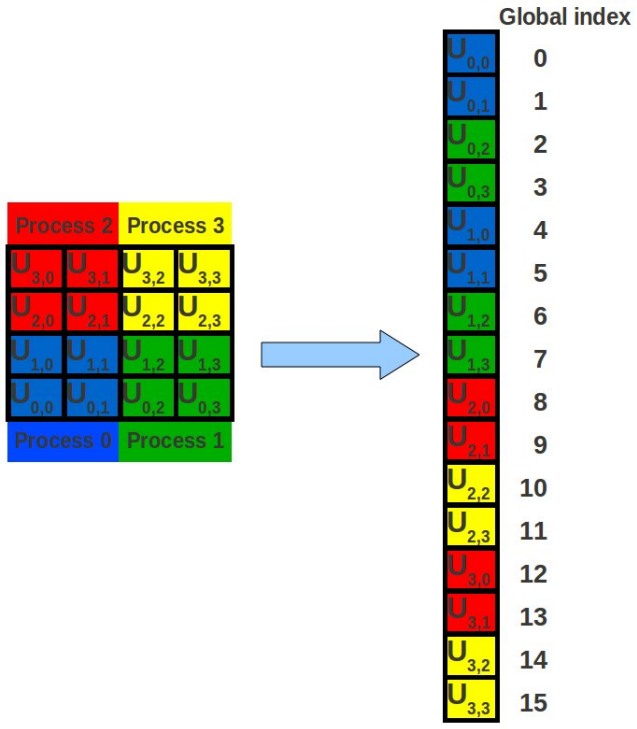


Figure 6.8: 2D row partitioning scheme with mesh size 4×4 on four processes

vector elements. Figure 6.9 shows an example of the matrix-vector multiplication operation on process 0. The blue entries are the non-zero entries of the sparse matrix. The blue elements in the local vector is stored originally on process 0, and the other four elements with green and red colors are obtained from process 1 and 2.

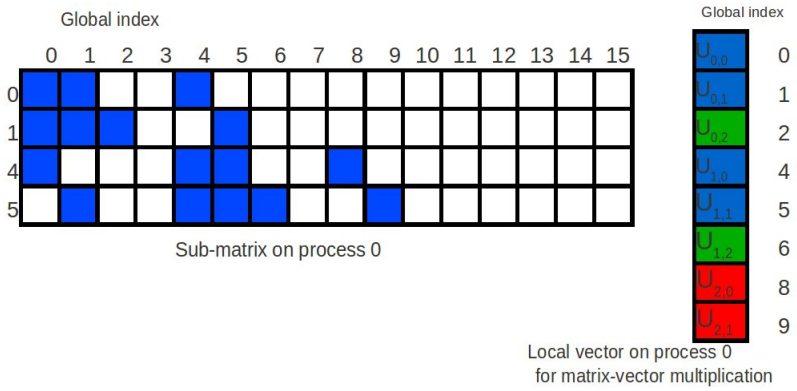


Figure 6.9: An example of matrix-vector multiplication on process 0

All the measurements in this section measures the wall-clock time (in seconds) of the iteration steps. The number of processes is varied between 1 and 64. The speedup results use the measurements on single core (free of communication) as the reference. The measurements are labeled respectively with different colors and point types. For the purpose of better illustrating, both x and y axes are transformed into log 2 scale.

We chose the mesh size as the power of 2 and one grid point in addition ($2^10 + 1 = 1025$ and $2^11 + 1 = 2049$) to reduce the roundoff error. Two meshes size 1025×1025 and 2049×2049 are defined for this two dimensional problem. We use 10^{-8} as the tolerance for the stop criteria of the while loop.

In Figure 6.10, the wall-clock time measurements show slightly different from

Figure 6.10: A comparison of the wall-clock time measurements (in seconds) of solving the Poisson's equation with parallel CG method on mesh 1025×1025 among six different implementations: pypar, mpi4py, MYMPI, pyMPI, Scientific.MPI and boostmpi

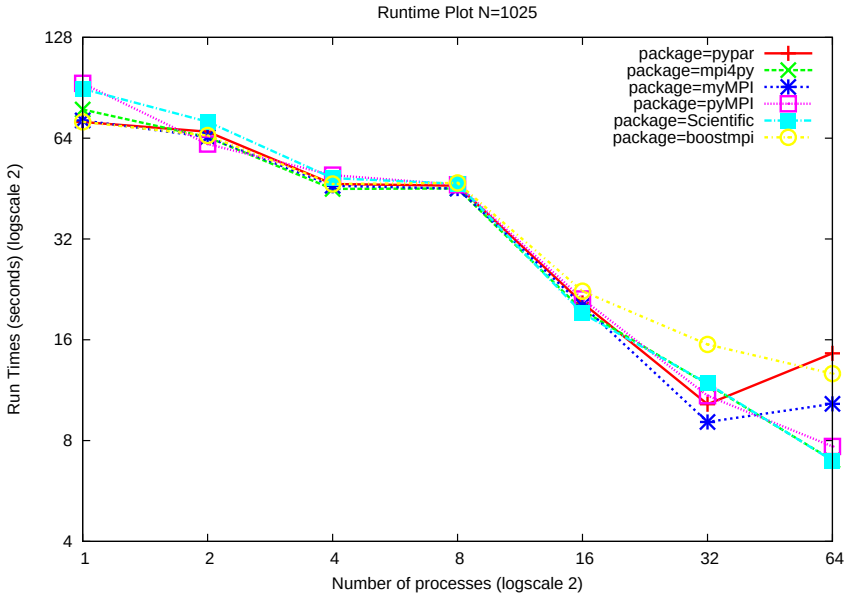


Figure 6.11: A comparison of the speedup results of solving the Poisson's equation with parallel CG method on mesh 1025×1025 among six different implementations: pypar, mpi4py, MYMPI, pyMPI, Scientific.MPI and boostmpi

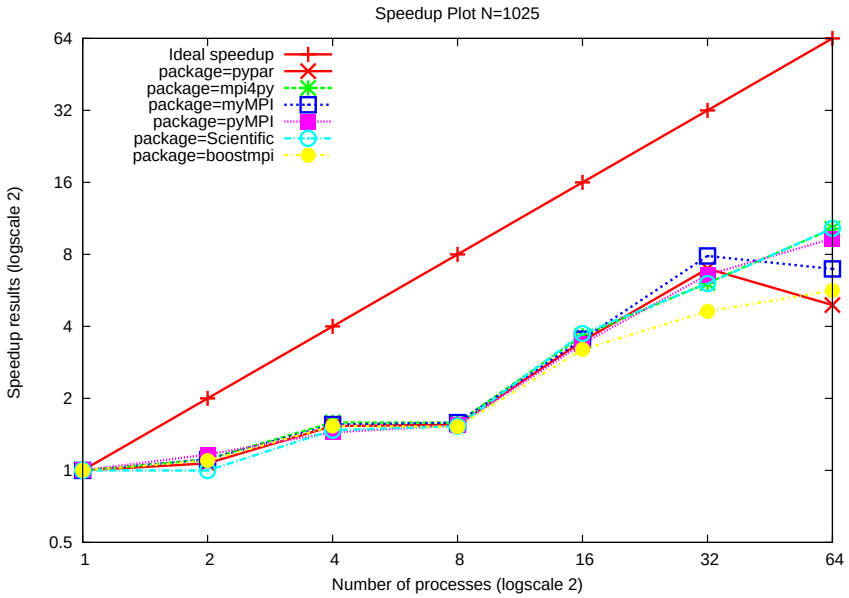


Figure 6.12: A comparison of the wall-clock time measurements (in seconds) of solving the Poisson's equation with parallel CG method on mesh 2049×2049 among six different implementations: pypar, mpi4py, MYMPI, pyMPI, Scientific.MPI and boostmpi

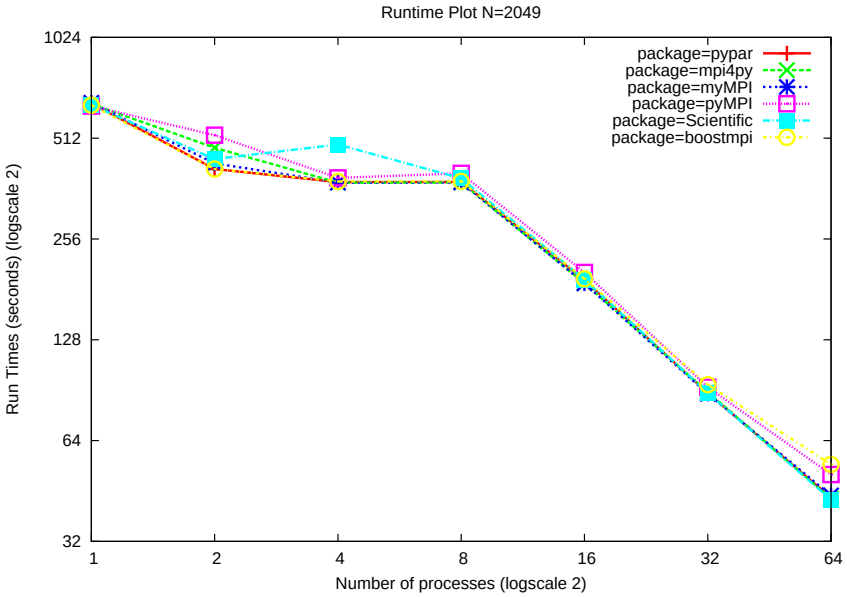
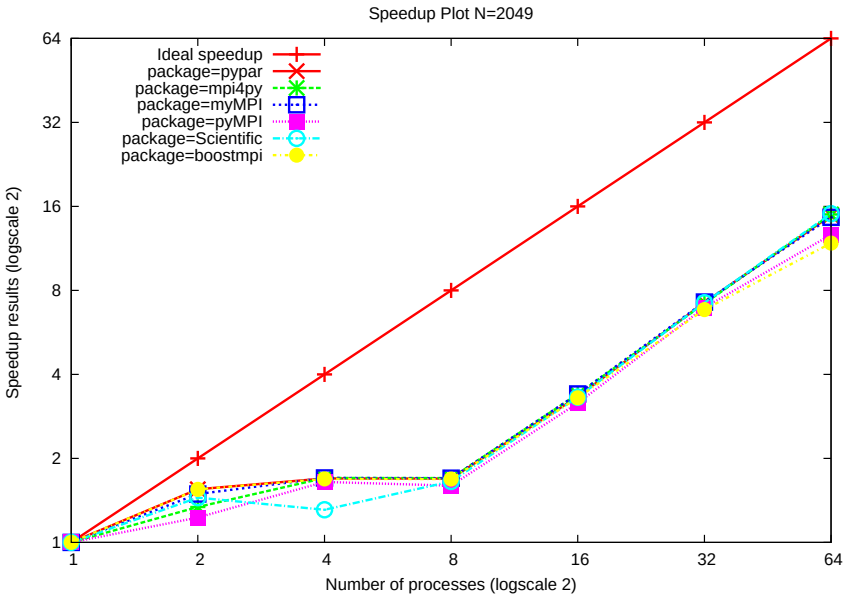


Figure 6.13: A comparison of the speedup results of solving the Poisson's equation with parallel CG method on mesh 2049×2049 among six different implementations: pypar, mpi4py, MYMPI, pyMPI, Scientific.MPI and boostmpi



each other, while the number of processes is as large as 16. The computation load on each process became comparatively smaller, then the difference may caused by the communication overhead. The communications are mainly on reduction operations. In connection with the performance comparison of the allreduce test in the previous chapter, we can see that the module with large latency have relatively bad performance in this case.

In Figure 6.11, the speed-up results show that the implementations are no longer efficient while the number of processes is larger than 32. The communication time took the most part of the overall time consumption for solving this problem.

In Figure 6.12, the wall-clock time measurements of six different versions almost draw on the same line. The large computation/communication ratio of this test case makes the differences of the communication overhead from different modules have very small influence on the overall performance. The differences while the number of processes smaller than 8 may be attributed to the cache efforts.

However, the speed-up results on the mesh size 2049 illustrates that all the six implementations have almost same speedup results. These Python MPI modules provide sufficient efficiency for MPI programming in real scientific applications. measurements show slightly different from each other, while the number of processes is as large as 16. The computation load on each process became comparatively smaller, then the difference may caused by the communication overhead. The communications are mainly on boundary exchanging operations. In connection with the performance comparison of pingpong test in the previous chapter, we can see that the module with large latency have relatively bad performance in this case. show that the implementations are no longer efficient while the number of processes is larger than 32. The communication time took the most part of the overall time consumption for solving this problem. $257 \times 257 \times 257$ illustrates that all the seven implementations have almost same speedup results. These Python MPI modules provide sufficient efficiency for MPI programming in real scientific applications.

6.3 Conclusion

After the experiments of solving the two scientific applications, we find that all the six Python MPI implementations give sufficiently good performance. Some modules have comparatively large communication overhead. However, when the computation/communication ratio is large, the communication overhead becomes a small part of the whole time consumption and have very little influence on the overall performance.

Regarding the programming issues, *mpi4py* clearly provides more MPI routines which is convenient for scientific application. Both `MPI_Sendrecv` and `MPI_PROC_NUll` are implemented, which greatly simplify the exchanging procedure in the first application. *pyMPI* and *pypar* provide flexible syntax and easy-to-use Python style function calls, which gives great convenience in developing parallel Python applications. *boostmpi* supports better C++ development style and nearly all the features of C++ library are available in this module. It is particularly suitable for C++ programmers or mixed programming users in Python and C++. As to the other two modules *MYMPI* and *Scientific.MPI*, although they provide less MPI routines and have some limitation of transmitting arbitrary Python objects, they are also suitable parallel programing specially in scientific computing.

Chapter 7

Conclusions and Future Work

In this thesis, we have compared the functionality, the implementation features and the parallel performance of six Python MPI modules. And through two scientific applications, we have compared the programming features and application performance of these modules.

Pypar is more flexible with different protocols for passing different Python objects, different options for using buffer and different options for returned status. Since it's easy to learn and sufficiently efficient, it is suitable for those Python users with less knowledge of MPI programming or tired of complicated syntax in C-MPI, but willing to try the great advantages of parallel programming.

On the other hand, *mpi4py* provides closely matched syntax with standard C MPI implementation and the most MPI routines, which is suitable for the users with some experiences on MPI programming. Furthermore, *mpi4py* has the best performance among the six Python MPI modules in our experiments.

Besides, *Scientific.MPI* has also very good performance in our tests, although only several MPI routines are provided. The messaging style is array-based instead of arbitrary Python objects. As a part of the package *ScientificPython*, this module is suggested for the usage of scientific computing.

As to the other three modules, the parallel performance we tested is not as good as the above three modules. *MYMPI* provides closely matched syntax with standard MPI implementation in C or Fortran, and supports only scalars and arrays. *pyMPI* supports the communication of any Python Objects and provides interfaces for over 120 MPI routines. It can be used for developing parallel applications for general purposes like password checkers and searching engines. About *boostmpi*, since it builds directly on top of a C++ interface of standard C MPI library, it works like the wrapper of wrapper. The communication overhead is quite large according to our performance tests with benchmarks. However, it supports better C++ development style and nearly all the features of C++ library are available in this *Boost.MPI* Python bindings. Therefore, *boostmpi* is particularly suitable for mixed programming users in Python and C++.

Future Work

A unified wrapper of MPI routines could be generated based on the studies of these six packages. This module allows a unified interface to these six packages. One reason for wanting the flexibility is that the different modules may be installed on different clusters. It enables writing scripts that are independent of the particular choice of the six Python MPI modules installed. The idea is that any of these modules can be replaced by one of the alternatives, and the same script should still work without modification. This will be a powerful tool for the MPI-based parallel programming in Python.

In order to have a more sophisticated comparison, more tools can be used to test the parallel performance of those MPI Python modules. TAU (Tuning and Analysis Utilities)[35] is one of them. Together with the profile visualization tool, *paraprof*, it provides a sophisticated and intuitive performance analysis of parallel programs written in Python, Fortran, C, C++ and Java.

Appendix A

Testing Environments

All the tests are done on an IBM multi-core based cluster with following hardware and software configurations.

- 84 compute nodes, 672 cores in total
- Each compute node has dual Intel(R) quad-core Xeon(R) E5420 2.50GHz processors, 8 cores per node
- nodes connected Gigabit Ethernets,
- 8 GB shared memory per node
- L2 cache: 12 MB + 12 MB
- Linux 2.6.24-25-server operating system
- Compilers: GCC 4.2.4
- MPI library: openmpi 1.2.5
- MPI compiler wrapper:

```
gcc -I/usr/lib/openmpi/include/openmpi
-I/usr/lib/openmpi/include
-pthread -L/usr/lib/openmpi/lib
-lmpi -lopen-rte -lopen-pal -ldl -Wl,
--export-dynamic -lnsl -lutil -lm -ldl
```

- Python Version: 2.5.2
- Numerical Python modules: Numpy 1.4.3 and Scipy 7.1

- Python MPI modules:

  Pypar 2.1.4

  mpi4py 1.2.1

  MYMPI 1.15

  pyMPI 2.5b0

  Scientific.MPI 2.9.0

  boostmpi 1.38.0.1

Appendix B

C-version MPI HelloWorld Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  int main(int nargs,char** args){
5    int numproc, myrank;
6    MPI_Init(&nargs,&args);
7    MPI_Comm_size(MPI_COMM_WORLD,&numproc);
8    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
9    printf("Hello, world! I'm process %d of %d.\n",myrank,numproc);
10   MPI_Finalize();
11   return 0;
12 }
```

Bibliography

[1] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer-Verlag New York, Inc., 2009.

[2] WikiPedia page of Message Passing Interface. http://en.wikipedia.org/wiki/Message_Passing_Interface.

[3] Python Docs for slicing. http://docs.python.org/release/2.3.5/whatsnew/section-slices.html.

[4] Python Numeric and Scientific Packages. http://wiki.python.org/moin/NumericAndScientific.

[5] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the Python programming language for serial and parallel scientific computations. *Sci. Program.*, 13(1):31–56, 2005.

[6] Xing Cai and Hans Petter Langtangen. *Parallelizing PDE solvers using the Python programming language*, volume 51. 2006.

[7] Parallel Processing and Multiprocessing in Python. http://wiki.python.org/moin/ParallelProcessing.

[8] Python Speed Performance Tips. http://wiki.python.org/moin/PythonSpeed/PerformanceTips.

[9] MPI Performance topics. http://www.mhpcc.edu/training/workshop2/mpi_performance/MAIN.html.

[10] Pypar Homepage. http://code.google.com/p/pypar/.

[11] mpi4py Homepage. http://mpi4py.scipy.org/.

[12] MYMPI Homepage. http://www.nbcr.net/software/doc/pydusa/.

[13] pyMPI Homepage. http://pympi.sourceforge.net/.

[14] ScientificPython Homepage. http://dirac.cnrs-orleans.fr/ScientificPython/ScientificPythonManual/.

[15] boostmpi Homepage. http://documen.tician.de/boostmpi/.

[16] Source codes for running the experiments and examples in the present thesis. http://folk.uio.no/wenjinli/master/source/codes.tar.

[17] Numpy Homepage. http://numpy.scipy.org/.

[18] MPI tutorial. https://computing.llnl.gov/tutorials/mpi/.

[19] First version of MPI. http://www-unix.mcs.anl.gov/mpi.

[20] WikiPedia page of Python programming language. http://en.wikipedia.org/wiki/Python_(programming_language).

[21] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 1998.

[22] Matlab vectorization guide. http://www.mathworks.com/support/tech-notes/1100/1109.html.

[23] Scipy Homepage. http://www.scipy.org/.

[24] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005.

[25] P Miller. pyMPI - An introduction to parallel Python using MPI. Available at: http://www.llnl.gov/computing/ develop/python/pyMPI.pdf, 2002.

[26] boost.MPI Homepage. http://boost.org/doc/libs/1_44_0/doc/html/mpi.html.

[27] LLCBench.MPBench Homepage. http://icl.cs.utk.edu/projects/llcbench/mpbench.html.

[28] Hans Petter Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*, pages 646–670. Springer-Verlag New York, Inc., 2003.

[29] WikiPedia page of Iterative methods. http://en.wikipedia.org/wiki/Iterative_method.

[30] WikiPedia page of Gaussian elimination. http://en.wikipedia.org/wiki/Gaussian_elimination.

[31] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[32] Explanation of Symmetric Matrix. http://mathworld.wolfram.com/SymmetricMatrix.html.

[33] Explanation of Positive Definite Matrix. http://mathworld.wolfram.com/PositiveDefiniteMatrix.html.

[34] Explanation of Sparse Matrix. http://mathworld.wolfram.com/SparseMatrix.html.

[35] TAU (Tuning and Analysis Utilities) Homepage. http://www.cs.uoregon.edu/research/tau/home.php.