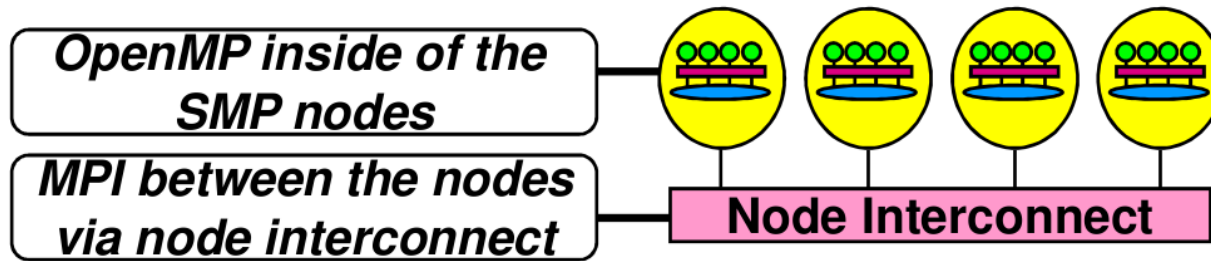


## Parallel programming

Modern parallel programming has two pillars: **open-MP** and **MPI**.

- **openMP:**
  - is used only for parallelization across multiple cores on the same node.
  - is part of the compiler, and is typically invoked by "-openmp" (or "-fopenmp") option when compiling. The code needs to contain "#pragma" statements.
- **MPI :**
  - is primarily used for parallelization across multiple nodes (although it works also on a single node).
  - is implemented as add-on library of commands. A proper set of MPI calls to the library will allow parallel code execution.
  - **Hybrid:** This is the combination of **MPI** and **openMP** parallelization whereby **openMP** is used inside a single node, and **MPI** across different nodes.



A fundamental difference between **MPI** and **openMP** is that in the latter all threads can access the same memory (so called shared memory available to all threads), while in **MPI** every thread can access only its own memory. To communicate between threads, **MPI** command needs to be issued, which is typically very expensive.

On the hardware level, every **MPI** commands results in the network traffic between nodes, which is expensive (latency problem). In **openMP** all threads are running on the same motherboard, and hence access the same RAM.

## M P I = Message Passing Interface.

It is a standardized collection of routines (functions) which is implemented for each programming language (fortran, C, C++, Python).

It was first standardized in 1994 (MPI-1.0) and second in 1997 (MPI-2.0) and (MPI-3.0) after 2012. Currently MPI-2.0 is most widely used.

Standard is available at <http://www.mpi-forum.org/docs/docs.html>

Many implementations of the standard are available (see [http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface))

The two most widely used implementations are MPICH

<http://www.mpich.org>

and open-MPI

<http://www.open-mpi.org>

(Note that open-MPI has nothing to do with openMP)

I will demonstrate examples using open-MPI (<http://www.open-mpi.org>) and for Python implementation Pypar (<https://code.google.com/p/pypar/>).

If you want to follow, you might want to install both.

There is a lot of literature available ("google MPI").

- <http://mpitutorial.com>
- <http://www.llnl.gov/computing/tutorials/mpi/#What>
- <https://www.youtube.com/watch?v=kHV6wmG35po>

## Brief history:

- 1980s - early 1990s: Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose. MPI Evolution
- April, 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.
- November 1992: - Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. MPIF eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.
- November 1993: Supercomputing 93 conference - draft MPI standard presented.
- Final version of draft released in May, 1994 - available on the at:

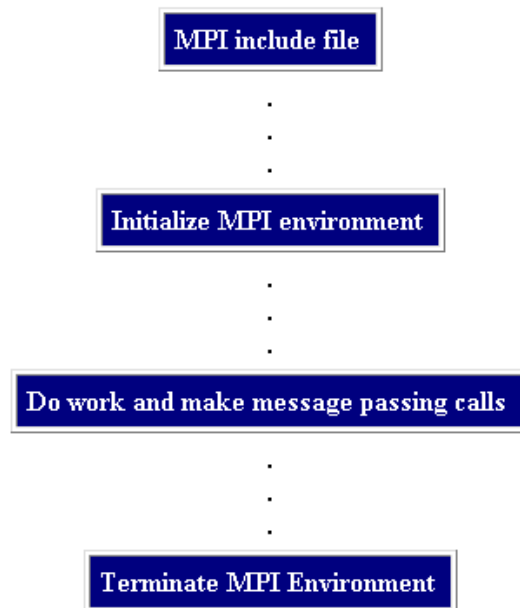
<http://www-unix.mcs.anl.gov/mpi>.

- MPI-2 picked up where the first MPI specification left off, and addressed topics which go beyond the first MPI specification. The original MPI then became known as MPI-1. MPI-2 is briefly covered later. Was finalized in 1996.
- Today, MPI-2 is widely used.

MPI is available for **Fortran**, **C** and **C++** and Python... We will present examples for **C++** and **Python**. Commands have the same name in all languages, but calls to routines differ slightly.

- MPI is large! It includes 152 functions.
- MPI is small! Many programs need only 6 basic functions.

Typical structure of a parallel code is organized as follows:



minimal C++ code (see

[http://www.physics.rutgers.edu/~haule/509/MPI\\_Guide\\_C++.pdf](http://www.physics.rutgers.edu/~haule/509/MPI_Guide_C++.pdf)  
for details)

```
#include <mpi.h> // must include mpi.h to acces functions from the library
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    MPI::Init ( argc, argv ); // Initializes communication. Mandatory call.
    int mpi_size = MPI::COMM_WORLD.Get_size ( ); // number of all threads / cores
    int my_rank = MPI::COMM_WORLD.Get_rank ( ); // each processor gets unique integer number (rank)
    cout<<"Hello world from core "<<my_rank<<endl; // every processor prints this message.
    MPI::Finalize ( ); // Mandatory call to finish the MPI
    return 0;
}
```

minimal Python code using Pypar:

```
#!/usr/bin/env python
import pypar # The Python-MPI interface

mpi_size = pypar.size() # number of all threads
my_rank = pypar.rank() # unique rank for each thread
my_name = pypar.get_processor_name() # name of the processor

print "I am proc %d of %d on node %s" %(my_rank, mpi_size, my_name)
pypar.finalize()
```



To compile C++ code, we need to execute

```
mpic++ -o example1 example1.cc
```

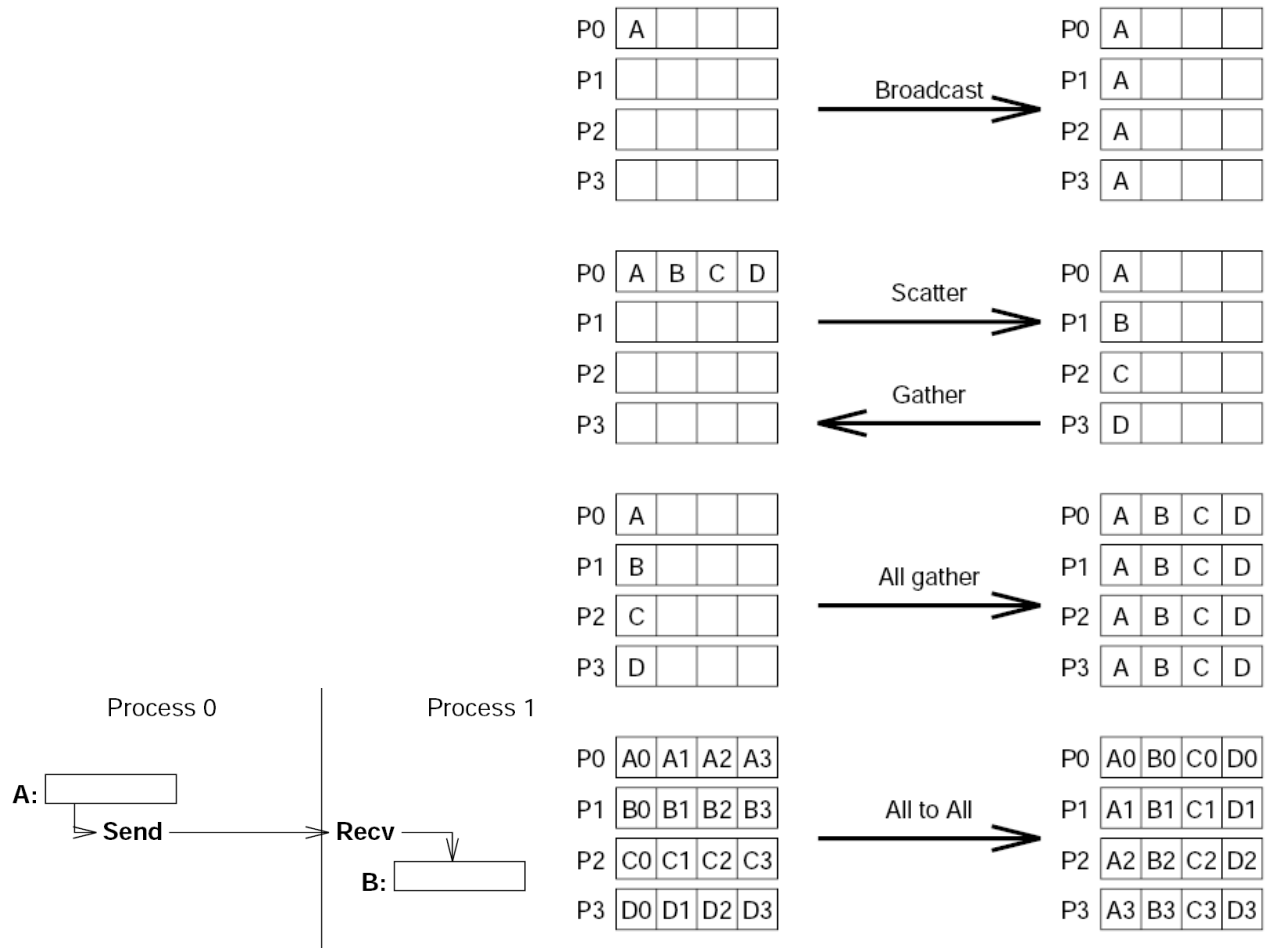
To execute the example, we should issue

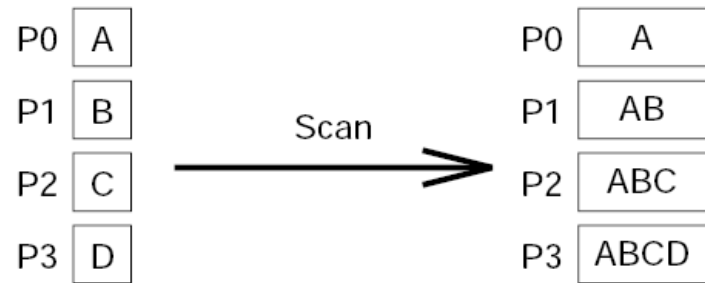
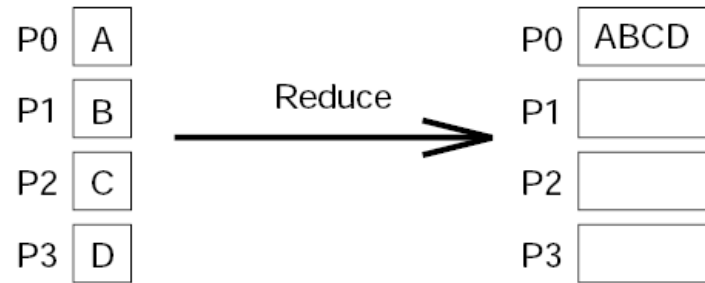
```
mpirun -n 4 example1
```

This will run the code on 4 cores on the local node. If one wants to run on multiple nodes, these nodes need to be properly configured into the common network. User needs to provide "machine file" (list of machine names) and the following option to mpirun

```
mpirun -hostfile <hostfile> -n 4 example1
```

Most usefull library calls include:





## Example of Monte Carlo integration on multiple cores using MPI

```

#include <cmath>
#include <iostream>
#include <mpi.h>
using namespace std;

double g(double* x) // 3D function to be integrated
{return 1./(1.0 - cos(x[0])*cos(x[1])*cos(x[2]))/(M_PI*M_PI*M_PI);}

int main(int argc, char *argv[])
{
    MPI::Init ( argc, argv );
    int mpi_size = MPI::COMM_WORLD.Get_size ( ); // How many cores?
    int my_rank = MPI::COMM_WORLD.Get_rank ( ); // Which processor is this?

    int iseed = time(0)+10*my_rank; // Each processor should start from different random sequence.
    srand48(iseed); // Set iseed

    int N = 10000000;
    double ave=0.0; // average
    double av2=0.0; // average^2
    for (int i=0; i<N; i++){
        double kv[3] = {drand48()*M_PI, drand48()*M_PI, drand48()*M_PI};
        double f = g(kv);
        ave += f;
        av2 += f*f;
    }
    ave=ave/N; // now we have average
    av2=av2/N; // and average^2
    double Vol = M_PI*M_PI*M_PI; // Volume of the region
    double Int = Vol*ave; // Integrals is Vol*<f>
    double Int2 = Vol*Vol*av2; // For error we also need Vol^2 * <f^2>
    double err = sqrt((Int2-Int*Int)/N); // This is standard deviation sigma^2 = ( (<f>*Vol)^2 - <f^2>*Vol^2 )/N
    cout<<"Integral="<<Int<<" Error="<<err<<endl;

```

```
double res[2]={Int,Int2}; // Packing results into array to communicate it via MPI
double res_sum[2];      // Reserving memory to store results of MPI call
MPI::COMM_WORLD.Reduce(res, res_sum, 2, MPI::DOUBLE, MPI::SUM, 0); // MPI CALL
// Results from each processor are summed up and the sum is now available on master
if (my_rank==0){
    cout<<"*** Average over all processors ***"<<endl;
    double Int = res_sum[0]/mpi_size;      // Average just sums up
    double Int2 = res_sum[1]/mpi_size;     // Also average^2 just sums up
    double err = sqrt((Int2-Int*Int)/(N*mpi_size)); // New error from all cores
    cout<<"Total Int="<<Int<<" Total err="<<err<<endl;
}
MPI::Finalize ( );
return 0;
}
```

## The same algorithm in Python using Pypar:

```
#!/usr/bin/env python
from scipy import *
import pypar      # The Python-MPI interface
import random

def g(x):
    return 1./(1.0 - cos(x[0])*cos(x[1])*cos(x[2]))/pi**3

if __name__ == '__main__':

    mpi_size = pypar.size()
    my_rank = pypar.rank()
    my_name = pypar.get_processor_name()

    random.jumpahead(my_rank)    # different random number start on each processor

    N = 100000
    ave=0.0
    av2=0.0
    for i in range(N):
        f = g(array([random.random(), random.random(), random.random()])*pi)
        ave += f
        av2 += f*f

    ave=ave/N    # now we have average
    av2=av2/N    # and average^2
    Vol = pi**3 # Volume of the region
    Int = Vol*ave    # Integrals is Vol*<f>
    Int2 = Vol*Vol*av2    # For error we also need Vol^2 * <f^2>
    Error = sqrt((Int2-Int*Int)/N) # This is standard deviation  sigma^2 = ( (<f>*Vol)^2 - <f^2>*Vol^2 )/N
    print "Integral=", Int, "Error=", Error

    res=array([Int, Int2])
    sum_res=zeros(2)
```

```
pypar.reduce(res, pypar.SUM, 0, buffer=sum_res) # This is the crucial MPI call

if my_rank==0:
    Int = sum_res[0]/mpi_size
    Int2 = sum_res[1]/mpi_size
    Error = sqrt((Int2-Int*Int)/(N*mpi_size))
    print "Final Integral=", Int, "Error=", Error

pypar.finalize()
```

We again compile C++ code with the command

```
mpic++ -o example2 example2.cc
```

and execute by

```
mpirun -n 4 example2
```

The Python code does not need compilation. Make the Python script executable (chmod a+x example2.py). Then execute by

```
mpirun -n 4 example2.py
```

Some useful tips:

- Always parallelize the most outside loop. Do not parallelize inside loops! This will minimize the communication.
- Avoid using many MPI calls. Try to combine MPI calls. Do not use multiple 'send-receive' calls if you can use Broadcast or Gather,....
- Every MPI call takes some minimum amount of time (it is expensive) and typically all processors need to wait at the point of MPI call. MPI call slows down all processors.
- Always develop and test serial code first. Parallel job is very hard to debug!
- Some algorithms are easy to parallelize. Some impossible. Test if more processors gives you better performance. Sometimes gives you even worse!
- In parallel programming, the "minimum amount of work" strategy does not apply. The amount of communication has to be minimized because communication is usually slow.



For example.

- Master reads some data. Other processors wait for the master
- MPI::Broadcast is used to transfer the data to slaves
- Each processors performs part of the work
- The common part of the work could be performed on master only. Let's call MPI::Gather or MPI::Reduce and perform common part of the computation on Master. When finished, master distributes the result by MPI::Broadcast
- Each processor continues with its own task
- Finally the results are merged together with MPI::Gather or MPI::Reduce

Much more efficient sheme is

- Every processor reads the data and immediately starts with work.
- Each processors performs part of the work
- Each processor performs the common part of the work. It does not take more time. All processors repeat the same calculation, but the MPI call can be skipped.
- Each processor continues with its own task
- Finally the results are merged together with MPI::Gather or MPI::Reduce