Parallel programming

1 Overview

Most widely accepted technique for parallel programming is so called: **M P I = Message Passing Interface**.

This is not a package or program, but rather a standardized collection of routines (functions). It was first standardized in 1994 (MPI-1.0) and second in 1997 (MPI-2.0). This is the current version. Standard is available at
`http://www.mpi-forum.org/docs/docs.html`

Many implementations of the standard are available (see
`http://www-unix.mcs.anl.gov/mpi/implementations.html`).
One of the most widely used implementations is MPICH:
`http://www-unix.mcs.anl.gov/mpi/mpich/index.htm#docs`.

This implementation *is a library of functions* which can be used for communication between processors.

One can download and test it on any machine, especially on dual-core machines which support two simultaneous running processes.

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

- `http://www.lam-mpi.org/tutorials/nd/part1/part1.pdf`
- `http://www.llnl.gov/computing/tutorials/mpi/#What`
- `http://www-unix.mcs.anl.gov/mpi/tutorial/mpiintro/`
- `http://www.mpi-forum.org/docs/docs.html`

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:

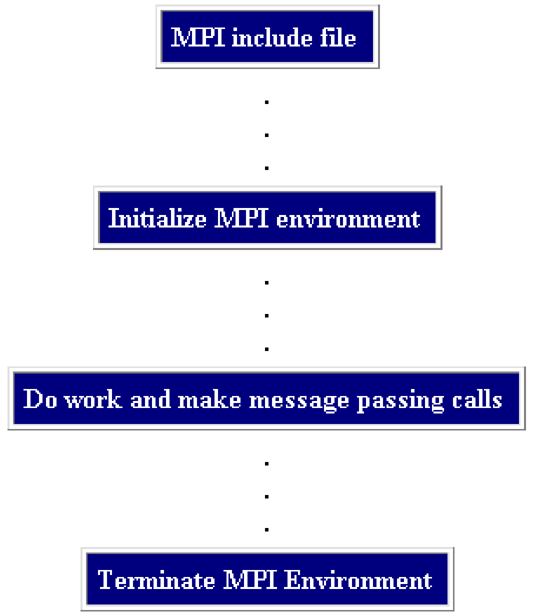
http://www-unix.mcs.anl.gov/mpi.

- 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 implementations are a combination of MPI-1 and MPI-2. A few implementations include the full functionality of both.

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

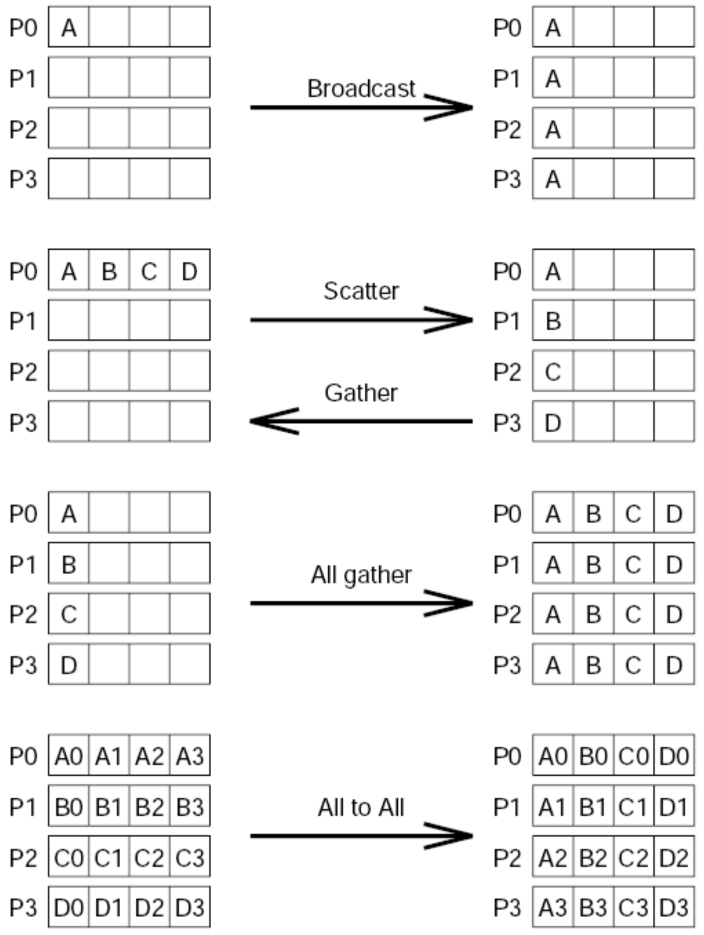
- MPI is large! It includes 152 functions.
- MPI is small! Many programs need only 6 basic functions.

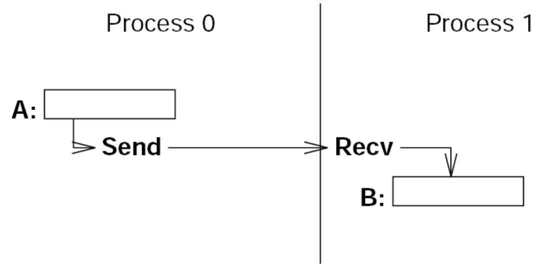
Typical structure of a parallel code is organized as follows:

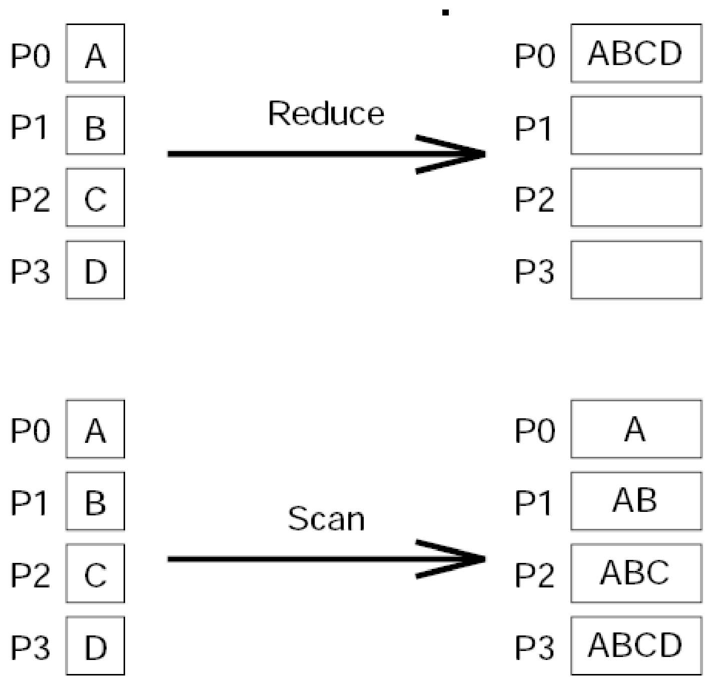

MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

```
#include <mpi.h> // must include mpi.h first and latter iostream!
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) // Must use argc and argv in main
{

  MPI::Init(argc, argv);// Initializes communication. Mandatory call.

  // each processor gets unique integer number (rank)
  int rank = MPI::COMM_WORLD.Get_rank();
  int size = MPI::COMM_WORLD.Get_size(); // number of all processors

  // every processor prints this message. Can result in a messy output.
  cout << "Hello world! I am " << rank << " of " << size << endl;

  MPI::Finalize(); // Mandatory call to finish the MPI
  return 0;
}
```

Most usefull calls include:






P0
A
P1
B
P2
C
P3
D
Reduce
P0
ABCD
P1
P2
P3
P0
A
P1
B
P2
C
P3
D
Scan
P0
A
P1
AB
P2
ABC
P3
ABCD

Example of usage:

```
#include <mpi.h>
#include <iostream>

using namespace std;
// mpd-demon mast be started on each machine for parallel execution.
// To start mpd-demon execute:
// > mpdboot -n xxx
// to start demon on xxx processors in mpd.hosts file.
// To run the code, execute:
// > mpdrun -n xxx lab3
//

int main(int argc, char *argv[]) // Must use argc and argv in main
{
  int master=0;    // We choose master to be equal to 0

  MPI::Init(argc, argv); // initializes communication. Mandatory call.
```

```
// each processor gets unique integer number (rank)
int rank = MPI::COMM_WORLD.Get_rank();
int size = MPI::COMM_WORLD.Get_size(); // number of all processors

// every processor prints this message. Can result in a messy output.
cout << "Hello world! I am " << rank << " of " << size << endl;

char data[100]; // array will contain data read by only one  processor
                // and distributed to all other processors
if (rank==master){
  // only master initializes the input data (reads input files....)
  strcpy(data, "This is an example input data which could be read from
}

// Broadcast call: Broadcasts a message from master to all processors.
// MPI::COMM_WORLD.Bcast(void *buffer, int count, MPI_Datatype datatyp
MPI::COMM_WORLD.Bcast(data, 100, MPI::CHAR, master);
```

```
// Now all processors have the same data to start some calculation
cout<<"Processor "<<rank<<" contains data: "<<data<<endl;

// Some computation is done here on each processor
// At the end we need to collect data from each processor and master w
// further process the data

srand48(rank+time(0));
// each processor will contain its unique real number
double my_number = drand48();
// each processor prints its number
cout<<"number on processor "<<rank<<" is "<<my_number<<endl;

// Reduce call: Reduces values on all processes to a single value
// MPI::COMM_WORLD.Reduce(void *sendbuf, void *recvbuf, int count, MPI
// MPI_Op can be one of:
//       MPI::MAX  maximum
//       MPI::MIN  minimum
//       MPI::SUM  sum
//       MPI::PROD  product
//      MPI::LAND  logical and
//      MPI::BAND  bit-wise and
//      MPI::LOR  logical or
//      MPI::BOR  bit-wise or
//      MPI::LXOR  logical xor
//      MPI::BXOR  bit-wise xor
//      MPI::MAXLOC  max value and location
//      MPI::MINLOC  min value and location
// User defined operators are also supported.
double receive;
MPI::COMM_WORLD.Reduce(&my_number, &receive, 1, MPI::DOUBLE, MPI::SUM,
// Only master has receive buffer set!
if (rank==master){
  // Master now further process the data
  cout<<"MPI Gather received "<<receive<<endl;
}
// Mandatory call to finish the MPI
MPI::Finalize();
return 0;
}
```

Some useful tips:

- Alwyas 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 inpossible. 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