Lesson 2 : Message passing

Xavier Juvigny

ONERA/HPC

January 2012

Sommaire

Parallel programming options

Programming a message–passing multicomputer can be achieved by

- Designing a special parallel programming language (e.g., OCCAM for transputers);
- Extending the syntax/reserved words of an existing sequential high–level language to handle message passing (e.g.: FORTRAN M, Unified Parallel C);
- Using a existing sequential high–level language and providing a library of external procedures for message passing (e.g. MPI, PVM).

Another option will be to write a sequential program and use a parallelizing compiler to produce a parallel program to be executed by multicomputer.

Message passing parallel programming

MPI is a norm for libraries for C/C++ or Fortran managing parallel code. Some wraps exists for Python/Ruby and other high level languages.
Any way, we have to say explicitly :

- Number of processes are to be executed;
- When to pass messages between concurrent processes;
- What to pass in the messages.

Two methods are needed for this form of message–passing systems :

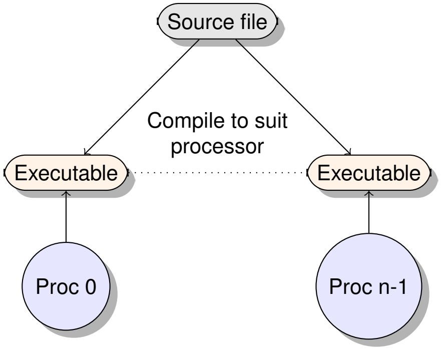
- A method of creating separate processes for execution on different computers;
- A method for sending and receiving messages.

SPMD model

In this case, *different processes execute the same program*. Within the program, there are **control statements** that will customize the code, i.e select different parts for each process.

Basic features :

- Usually, static process creation at execution;
- Using only basic features of MPI is enough;



MPMD : Multiple Program Multiple Data model

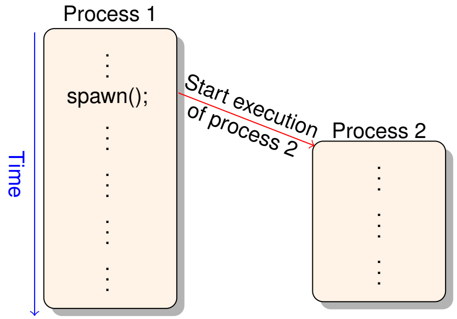
In this case, *separate programs are executed for several groups of processors*.

Several strategies possible :

- master–slave : a single processor executes a master process and the other processes are started from within the master process.

  Basic features :

  - Usually dynamic process creation
  - Possible only with MPI version 2 .



- Concurrent system : All programs are independant. Some synchronizations are required from time to time. Option available with all MPI versions.

Identification of a processus

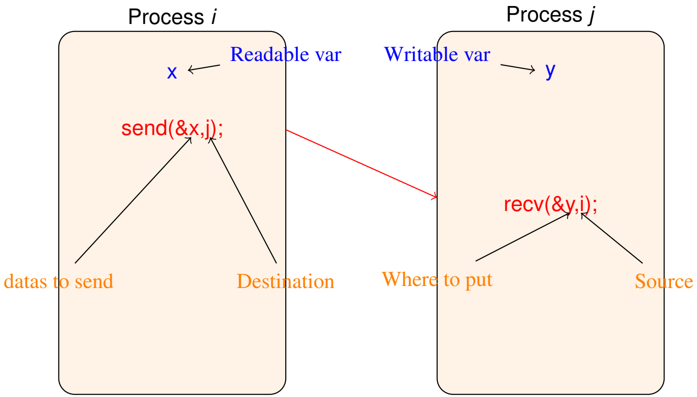
To manage a part of data or for control statements, an id must be affected for each process. Ever more, to split data accordingly, the number of processes launched for the application must be known.

Example : let's consider a parallel sum of two vectors of length *n*. Each process must sum the same amount of data than other processes (for load balancing).

- Amount of data to sum in each vector for a process ? Retrieve the number of processes nbProcs executing our code. Number of elements in each vector to sum : $n_l = \frac{n}{\mathrm{nbProcs}}$.
- Which elements in each vector a process must sum ? Each process has an unique identifiant r : $0 \leq r < \mathrm{nbProcs}$. For process r : caring about coefficients with indices i as $n_l r \leq i < (n_l + 1)r$: we done a **partition** of the data of each vector.

Basic send and receive routines

Passing datas between processes using `send()` and `recv()` library calls to send/receive a message.



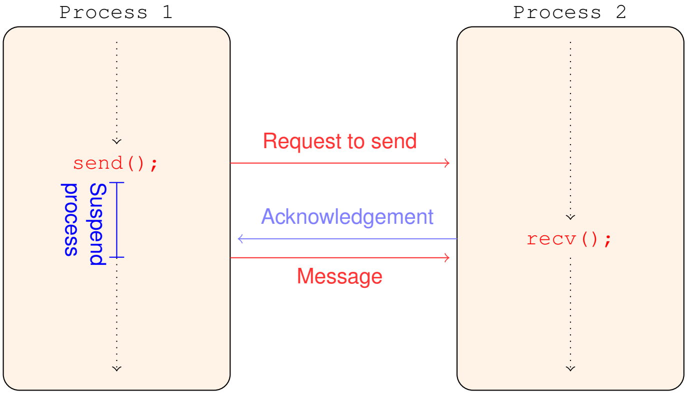
Synchronous message–passing

- Synchronous message–passing routines return when the message transfer has been completed;
- There is no need for message buffer storage.
  - The synchronous send routine could wait until complete message can be accepted by the receiving process before sending the message;
  - The synchronous receive routine wait until the message it is excepting arrives.
- Synchronous routines perform two basic actions : They
  - transfer data and
  - synchronize processes.

. . . Synchronous message–passing

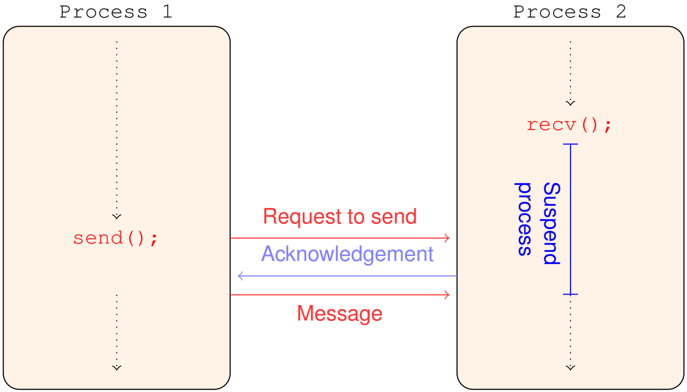
A three–way protocol is actually used here :

- **Case 1** : `Process 1` arrives to `send()` before `Process 2` arrives to `recv()` :



. . . Synchronous message–passing

- **Case 2** : `Process 1` arrives to `send()` after `Process 2` arrives to `recv()` :



Deadlock

A tricky problem in synchronous message–passing is the deadlock : it may occur when *messages cannot be sended because they are blocked by other messages waiting to be sended and these packets are blocked in a similar way such that actually none of the messages can be send*.

Example :

- Node 1 wishes to send to 2;
- Node 2 wishes to send to 3;
- Node 3 wishes to send to 1;

A famous example was introduced by Dijkstra : 5 philosophers may think or eat spaghetti using a circular table with 5 plates and 5 forks when hungry. Eating spaghetti requires two forks (!?). A philosopher release his forks when he is no more hungry. A deadlock appears when all philosophers are present and all held the right–hand side fork.

Blocking and nonblocking message–passing

Blocking

- This term is used to describe routines that do not return until the transfer is completed.
- More precisely, the routines are blocked from continuing the process code;
- Generally speaking, the terms synchronous and blocking are synonymous.

Non–blocking

- This term is used to describe routines that return whether or not the message had been received

Warning : The general terms were redefined in MPI, see below...

MPI definition of blocking and nonblocking

blocking – Return after their local actions are finished, though the message transfer may not have been completed (E.g., for `send()` it may return after the data are put in buffer to be send).

Nonblocking – Return immediately. In such a case, it is assumes that the data storage to be used for the transfer is not modified by the subsequent statements before the transfer is completed and it us the programmer duty to ensure this.

Notice : This type of message is based on the use of message buffers between the source and destination processes. As the buffers are of finite length, it may happend that the `send()` routine is blocked because the available buffer space has been exhausted.

Message tag

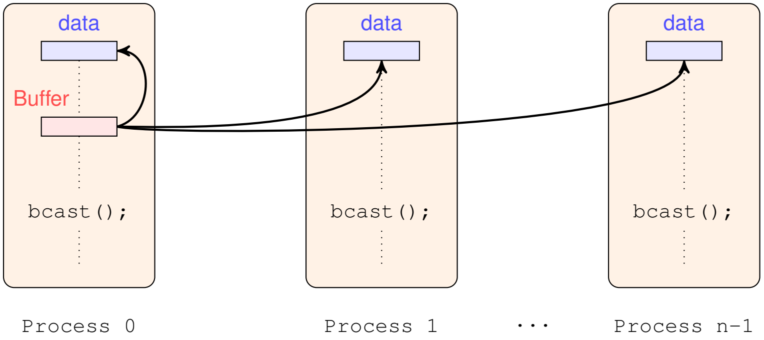
A message tag is an extra information put in the message to differentiate between different messages being sent.
Example :

⋮	⋮
`send(&x,2,5);`	`recv(&x,1,5);`
⋮	⋮
(Process 1)	(Process 2)

The tag 5 is used to match the send statement in `process 1` to the receive statement in `process 2`.
Notice : If such a special type matching is not required, then a wild card message tag is used, so that `recv()` will match any `send()`.

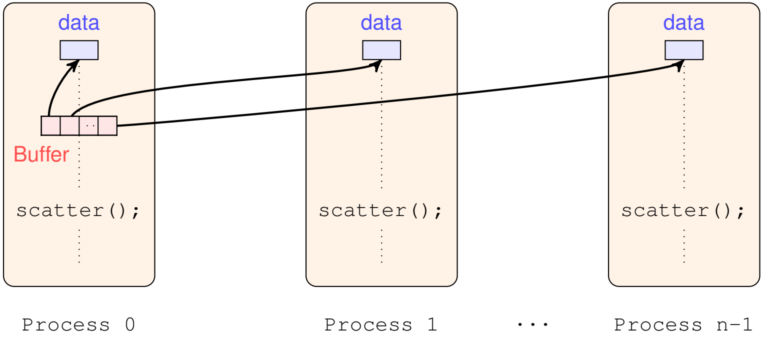
Broadcast

Broadcast is used to send the same message to all processes concerned with the problem
Multicast is similar, but it is used to send a message to a defined group of processes.



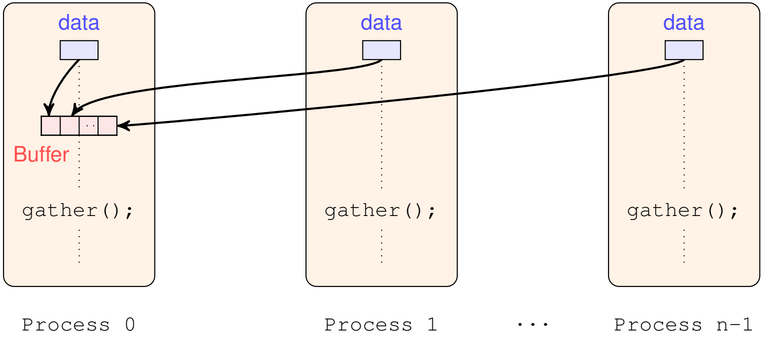
Scatter

Scatter is used to send each element in an array of data of the sending process to corresponding separate processes (datum from the i^{th} location goes to the i^{th} process).



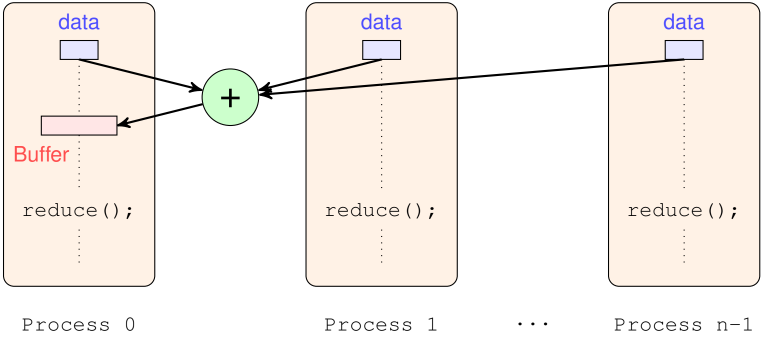
gather

Gather is the opposite operation : the receiving process collect in an array the data sent by separate processes (datum from the i^{th} process goes to the i^{th} location).



Reduce

Reduce combines `gather()` routine with an arithmetic or logical operation : the receiving process collects the data, applies the operation and saves it in its own memory.



Software tools

PVM (Parallel Virtual Machine)

- A first wildly accepted attempt to use a workstation cluster as a multicomputer.
- It may be used to run programs on both homogeneous or heterogeneous multicomputers.
- It has a collection of library routines to be used with C or FORTRAN programs.
- It's free.

. . . Software tools

MPI (Message Passing Interface)

- It is a standard developed by a group of academies and industrial people to increase the use and the portability of message passing.
- Several free implementation exists
- It has a collection of library routines to be used with C, C++ and Fortran programs. Some interface exist to use MPI with Python programs.
- It may be used to run programs on both homogeneous or heterogeneous multicomputers.

MPI

General : MPI is a standard with various implementation
Process creation and execution : Defined at the running of the programs. The set of computers used for the problem can be defined prior the running of the programs (A convenient way to do this is by using a machinefile listing the names of the computers available. This machinefile is then read by MPI.
Communications : One defines the scope of the communication operation; the set of all involved processes may be accessed using the predefined variable `MPI_COMM_WORLD`; each process has an unique rank, a number from 0 to $n-1$ (where n is the number of processes); other communication groups may be defined.

. . . MPI

Shape of an MPI program

```c
int main(int argc, char *argv[])
{
  int myrank, nbprocs;
  MPI_Init(&argc, &argv);  /* Initialize MPI context */
  /* Find the rank of the process */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  /* Find the number of processes running the program */
  MPI_Comm_size(MPI_COMM_WORLD, &nbprocs);
  ...
  if (myrank == 0)
  {
     /* Do something if first processor */
  }
  else
  {
     /* Do something different for others processors */
  }
  MPI_Finalize();/*Delete MPI context */
}
```

. . . MPI

Point–to–point communication : Message tags and wild cards may be used (`MPI_ANY_TAG` or `MPI_ANY_SOURCE`)

Blocking routines : Return when they are locally complete, i.e, when the location used for the message can be used again without affecting the message being sent; general format :

```
MPI_Send(  void* buf, int count, MPI_Datatype datatype,
           int dest, int tag, MPI_Comm comm)
```

where `buf`–address of send buffer, `count`–number of items to send, `datatype`–Data type of each item, `dest`–Rank of destination process, `tag`–Message tag, `comm`–Communicator

```
MPI_Recv(  void* buf, int count, MPI_Datatype datatype,
           int source, int tag, MPI_Comm comm, MPI_Status *status)
```

where `buf`–address of receive buffer, `count`–Maximum number of items to receive,

`datatype`–Data type of each item, `dest`–Rank of source process, `tag`–Message tag,

`comm`–Communicator, `status` – Status after operation

...MPI

Example (Blocking communication) : To send an integer x from process 0 to process 1 :

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0){
  int x;
  MPI_Send(&x, 1, MPI_INT, 1, 73, MPI_COMM_WORLD);
  } else if (myrank == 1){
  int x;
  MPI_Status status;
  MPI_Recv(&x,1,MPI_INT, 0,73, MPI_COMM_WORLD,&status);
  }
```

. . . MPI

Non–blocking communication : `MPI_Isend()` and `MPI_Irecv()` – return “immediately”, even if the communication is not safe; to be used in combination with `MPI_Wait()` and `MPI_Test()` in order to ensure a complete communication.
Example (non–blocking communication) – same example

```
MPI_Status status;
int x;
if (myrank == 0){
  MPI_Request req1;
  MPI_ISend(&x, 1, MPI_INT, 1, 73, MPI_COMM_WORLD,&req1);
  compute();
  MPI_Wait(&req1, &status);
  } else if (myrank == 1){
  MPI_Recv(&x,1,MPI_INT, 0,73, MPI_COMM_WORLD,&status);
  }
```

. . . MPI

Send communication modes : Four basic modes are

1. Standard mode send – It is not assumed that the corresponding receive routine has started (buffer space is not defined here; if buffering is provided, send can be complete before the corresponding receive was reached)
2. Buffered mode – Send may start and return before a matching receive was reached (here it is necessary to specify buffer space)
3. Synchronous mode – Send and receive have to complete together (however, they may start at any time)
4. Ready mode – Send can only start if a matching receive was already reached (use it with care...)

. . . MPI

Collective communication : This applies to processes included in a communicator. The main operations are :

- `MPI_Bcast()` – Broadcast from root to all other processes
- `MPI_Gather()` – Gather values from processes in the group
- `MPI_Scatter()` – Scatter parts of the buffer to processes
- `MPI_Reduce()` – Collect and combine values from processes
- `MPI_Allreduce()` – Combine reduce and broadcast

Barrier : May be used to synchronize processes by stopping each process until all have reached the barrier call

- `MPI_Barrier(MPI_Comm comm)`

Example of MPI program

We illustrate MPI programming style with a simple question :
add the numbers from a file using multiple processes
A master–slave approach is used :

- A master process (process 0) detects the number of processes from communicator, reads data from the file and sends them to all processes (by broadcast).
- Each process (including the master) identifies its portion of data and adds them.
- The master collects the partial sums and adds them (using `reduce` statements) and prints the final result.

MPI program

```
# include <mpi.h>
# include <stdio.h>
# include <math.h>
# define MAXSIZE 1000
void main(int argc, char *argv) {
  int myid, numprocs;
  int data[MAXSIZE], i, x, low, high, myresult, result;
  char fn[255];
  char *fp;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if ( myid == 0 ) {
    if ( (fp = fopen("rand_data.txt", "r")) == NULL ) {
      printf("Can't open the input file\n");
      MPI_Abort(MPI_COMM_WORLD,1);
    }
```

. . . MPI program

```
    for (i=0; i<MAXSIZE; i++) fscanf(fp,"%d",&data[i]);
  }
  /* Broadcast data */
  MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);
  /* Add my portion of data */
  x = MAXSIZE/numprocs;
  low   = myid*x;
  high = low + x;
  myresult = 0;
  for ( i = low; i < high; i++ )
    myresult += data[i];
  printf("I got %d from %d\n",myresult,myid);
  /*Compute global sum */
  MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (myid == 0) printf("The sum is %d.\n",result);
  MPI_Finalize();
}
```

Parallel program evaluation

Program evaluation :

- Both theorical and empirical techniques may be used to determine the efficiency of (parallel) programs.
- The ultimate goal is to discriminate between various parallel processing techniques; a fine tune is also necessary to find the best number of processes and a good balance of the computation time and of the communication time of each process.
- An extra goal to find out if a parallel processing approach is actually better suited than a (usually simpler) sequential one.

Parallel execution time

Theorical evaluation of parallel programs is based on :

- Parallel execution time, t_p, consists of the computation time t_{comp} and the communication time t_{comm}, namely :

$$t_p = t_{comp} + t_{comm}$$

- Computation time, t_{comp}, is estimated similarly as in the case of sequential algorithms (Supposed here that all processes use identical computers)
- Communication time, t_{comm}, consists of the startup time $t_{startup}$ (also called “message latency”) and the time to send data $n.t_{data}$, where t_{data} is the time to transfer a data word, namely :

$$t_{comm} = t_{startup} + n.t_{data}$$

Latency hiding

- A general method to overcome the significant message communication time is to overlap communication with subsequent computations
- Nonblocking send routines are particulary useful to enable latency hiding
- Another technique is to use multi–threading
- Beware updating of sended datas !

Cost–optimal algorithms

A cost–optimal (or processor–time optimal) algorithm is one such that

(parallel time complexity) $\times$ (number of procs.)
=
(sequential time complexity)

Example :

- Suppose the best know algorithm for problem P has time complexity $O(n \log n)$
- A parallel algorithm solving the same problem using n processes and having the time complexity $O(\log n)$ is cost–optimal, while a parallel algorithm which uses n^2 processes and has time complexity $O(1)$ is not cost-optimal.

Evaluating programs empirically

Measuring execution time : To measure the execution time between point L_1 and L_2 in the code, one may use the following construction :

```
L1: double t1 = MPI_Wtime();
    ...
L2: double t2 = MPI_Wtime();
    double ellapsedTime = t2-t1;
    printf("Elapsed time : %5.2g seconds\n",elapsedTime);
```

. . . Evaluating programs empirically

Communication time by Ping–Pong methods : To empirically estimate the communication time from a process P_1 to a process P_2 one may use the following method : Immediately after receiving the message P_2, send the message back to P_1.

```
double t1 = MPI_Wtime();
if ( myrank == P1 )
{
  send(&x,P2); recv(&x,P2);
}
else if ( myrank == P2 )
{
  recv(&x,P1); send(&x,P1);
}
double t2 = MPI_Wtime();
elapsedTime = 0.5*(t2-t1);
```

Remark : Another method is to use `MPI_Barrier` after sending or receiving

Debugging strategy

An useful three–step approach to debugging message–passing programs is :

1. If possible, run the program as a single process and debug as a normal sequential program.
2. Execute the program using two to four multi–tasked processes on a single computer. Now, examine actions such as checking that messages are indeed being sent to correct places. It is very common to make mistakes with message tags and have messages sent to wrong places.
3. Execute the program using two to four processes but now across several computers. This step help to find the impact of netword delays on synchronization and timing constraints of your program.