MPI (I)

Distributed-memory programming

Juan Ángel Lorenzo del Castillo

CY Cergy Paris Université

ING2-GSI-MI Architecture et Programmation Parallèle

2023 - 2024

juan-angel.lorenzo-del-castillo@cyu.fr

Table of Contents

Table of Contents

Message Passing programming model

- Used in distributed memory multicomputers and computer networks (heterogeneous systems)
  - Private variables
- Primitives for:
  - Accessing other processors local memory (communications).
  - Synchronisation.
- Flexible: total control of your program.
- Downside: the developer is responsible for optimising the code.
  - Distribution of data and computation, synchronisation, communications. . .
- Typically, better performance than OpenMP (although it depends on the degree of optimisation).

Message Passing programming model

- Standard message passing libraries:
  - PARMACS, PVM (Parallel Virtual Machine) and MPI (Message Passing Interface)
- Programs in C/C++ or Fortran with communication routines.

Comparison

Shared Memory

- Programming is easier.
- It is more difficult to make mistakes
- Parallelisation can be done incrementally

Message passing

- Easier to get better performance
- Hard to detect and correct errors

Table of Contents

MPI

- MPI (*Message Passing Interface*) is a library of communication functions for inter-process message delivery and reception.
  - Current standard (since 1993) for distributed-memory, message-passing programming.
  - Can be also used in shared-memory systems, (heterogeneous or not) clusters, computer networks, grids, etc.
  - Portability and efficiency oriented.
- Versions for C, C++ (deprecated), Fortran 77 and Fortran 90.
- Inter-process communications must be explicitly defined.
  - Data movement
  - Synchronisation
- Messages contain *precise instructions*:
  - Process who sends the message, data sent, type of data, number of data elements, message receiver, variable where the data received will be stored, etc.

MPI

Standards

MPI-1 (1994)

- Manufacturers and researchers consortium

MPI-2 (1997)

- Dynamic process management, *1-side* communications, parallel I/O...

MPI-3 (2012)

- *1-side* extensions, non-blocking collective operations

- Current implementations:
  - MPICH (MPI-3), OpenMPI, HP, Intel, pyMPI (∼MPI-1.2)...
  - We will utilise OpenMPI

MPI: Processes

- MPI implements the SPMD (*Single Program Multiple Data*) programming model.

```
if (pid == 1)
   SEND to pid2
else if (pid == 2)
   RECEIVE from_pid1
```

  - Private address space
- MPI assumes static process management (number and allocation)
  - Each process has an unique identifier (*pid* or *rank*)
  - MPI-2 introduces dynamic process management
- MPI groups in *communicators* all processes involved in a parallel execution
  - A *communicator* groups processes that can exchange messages
  - The `MPI_COMM_WORLD` communicator is created by default and groups all running processes

MPI: communication types

- Point to point
  - Only two processes are involved (transmitter-receiver)
  - They must belong to the same *communicator*
- Colectives
  - Communication routines where more than two processors are simultaneously involved
  - Can be built from point-to-point communications
  - They must belong to the same *communicator*

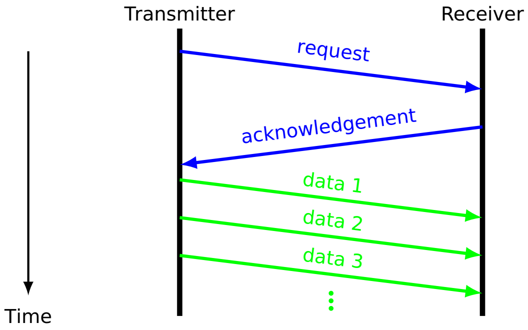
Communication strategies

- Inter-process communication strategies

  - Synchronous (vs. asynchronous)
  - Blocking (vs. non blocking)
  - With buffer (vs. without buffer)

Synchronous communication
Communication strategies

- Transmission request (wait)
- Transmission approval
- Data sending

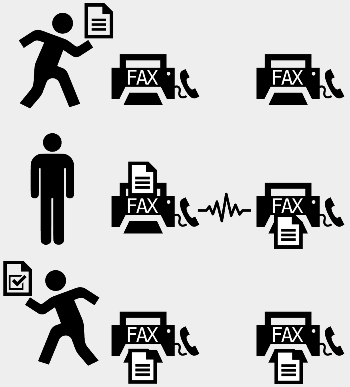


Synchronous communication

Communication strategies

Synchronous sending

- The sender needs an acknowledgement on the message reception
- The communication process will not end until the whole message has been received



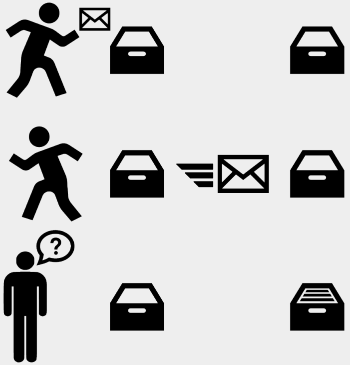
http://iconmonstr.com/

Asynchronous communication

Communication strategies

Asynchronous sending

- The sender knows only that the message was sent
- The communication process will end as soon as the message was sent



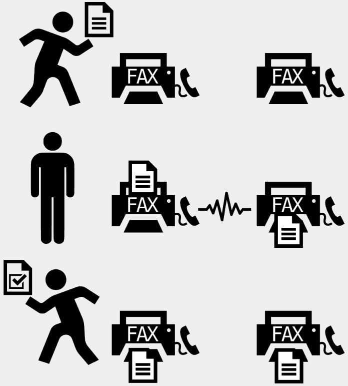
http://iconmonstr.com/

Blocking communication

Communication strategies

Blocking operation

- The communication subroutine only ends when the communication operation is completed



http://iconmonstr.com/

Blocking communication

Communication strategies

- We wait for the communication to happen (or for the user buffer to be available again)
- Execution continues (using a non-blocking communication) and we check later whether it has finished

> Non blocking + wait function = blocking

- Synchronous communications are always blocking ones.

Blocking communication Communication strategies

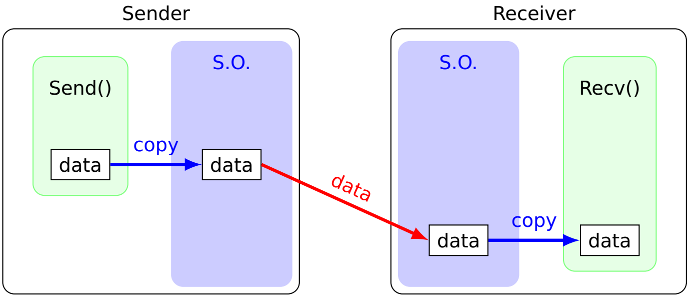
Non-blocking operation

- The sending process starts and the sender keeps its execution
- There exist functions that check the reception or wait for the message to be received

http://iconmonstr.com/

Buffered communication
Communication strategies

- The message is copied to a buffer



- They can be either blocking or non-blocking operations

Communication strategies

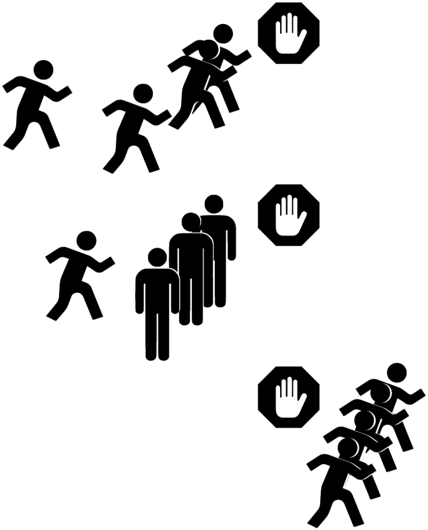
- Each strategy has its advantages and disadvantages
  - Synchronous: faster if the receiver is ready to receive data
    - No need to copy data into a buffer
    - Deadlock risk (because it is a blocking operation)
  - Buffered: the sender is not blocked if the receiver is not available
    - We need to make a copy of the message
- The efficiency of the communication will determine the performance

Collective communications

- Communications between more than two processes
  - They can be built from point-to-point communications or using specific hardware

- Barriers
- Radiation
- Reduction

Barriers
Collective communications

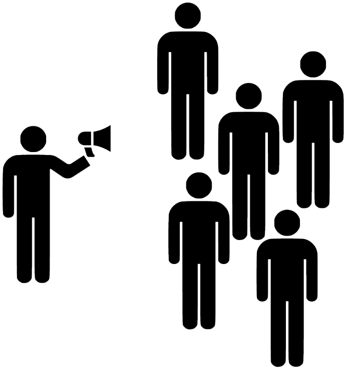


http://iconmonstr.com/

- To synchronise processes
- No data exchange
- The program stops until all processors arrive to the barrier

Radiation
Collective communications

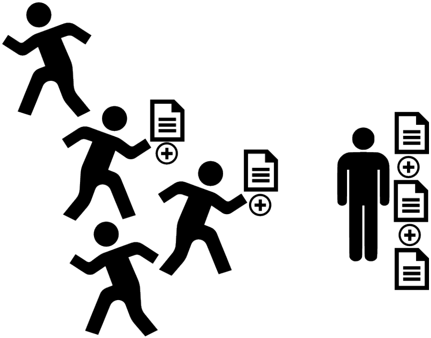
- Communication from one to all
- One of the processes sends a message to multiple receivers



http://iconmonstr.com/

Reduction
Collective communications

- They take values from different processors and perform an operation that reduces to a single value
- The result can be radiated to the rest of processes



http://iconmonstr.com/

Basic MPI

- MPI is a library of communication routines
- Header files
  - In C (sometimes, "mpi.h")

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

  - In Fortran

```
include 'mpif.h'
```

- Format of the MPI functions
  - In C

```
int error;
error = MPI_Xxxxx(parameters);
```

  - In Fortran

```
CALL MPI_XXXXX(parameters, IERROR)
```

Initialisation/finalisation Basic MPI

Initialise MPI

```
MPI_Init(int *argc, char **argv[]);
```

- It is the first MPI function that must be executed in the program
- It defines (for all processes) a communicator that includes all processes: MPI_COMM_WORLD

Finalise MPI

```
MPI_Finalize( );
```

- The last MPI function that must be executed in the program

Process identification Basic MPI

- Processes involved in a MPI execution are sorted and numbered consecutively starting from 0.

Identification of an MPI context

```
MPI_Comm_size(MPI_Comm comm, int *np)
```

- Returns in `np` the number of processes of the `comm` communicator
- `MPI_Comm_size(MPI_COMM_WORLD, &np);`

```
MPI_Comm_rank(MPI_Comm comm, int *myid)
```

- Returns in `myid` the identifier of the process on the `comm` communicator
- `MPI_Comm_rank(MPI_COMM_WORLD, &myid);`

Basic MPI

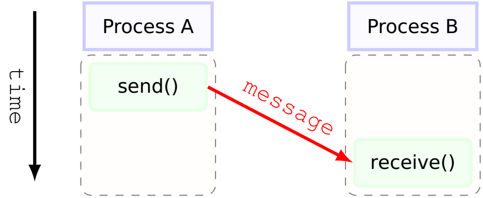
A simple example

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int myid, np;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  printf("I am the node %d of %d\n", myid, np);

  MPI_Finalize();
  return 0;
}
```

Compilation: `mpicc -o example1 example1.c`
Execution: `mpirun -np 4 example1`

Communication functions Basic MPI

- Communication between two processes, A and B, is performed by a couple of functions:



  - Process A must execute a **send** function
  - Process B must execute a **receive** function
  - If any of the functions is not executed, the communication will not take place (**deadlock** risk!)
- To send or receive a message we must specify:
  - To whom the message is sent (or from whom it is received)
  - Data to send (start position and length)
  - Data type
  - A message identifier (`tag`)

Communication functions Basic MPI

- Different implementations according to the synchronisation and buffering type.
  - Standard (depends on the implementation)
  - Synchronous or asynchronous
  - Blocking or non-blocking
  - Buffered or not
  - Ready
  - Persistent

send and receive Basic MPI

send: basic function

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

`<buf, count, datatype>` Message to send

- Standard types: `MPI_CHAR`, `MPI_FLOAT`...

`<dest, comm>` Receiver

`<tag>` message tag

- to associate types to messages (classes, order...)

- System-dependent implementation

send and receive Basic MPI

receive: basic function

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

<buf, count, datatype> Message to receive

- Standard types: MPI_CHAR, MPI_FLOAT...

<src, comm> Sender

<tag> message tag

<status> State

- Control information about the received message

- MPI_Recv is a blocking operation

`send` and `receive` Basic MPI

Some considerations

- The message size in `MPI_Recv()` (`count`) must be equal or higher than the size of `MPI_Send()`
  - Can be found out with `MPI_Get_count(status, datatype, &count)`
- In `MPI_Recv()`
  - The source (`src`) can be `MPI_ANY_SOURCE`, if we want to receive from anybody
  - The tag can be `MPI_ANY_TAG`
- `status` is a struct with information about the message
  - `status.MPI_SOURCE`: shows the message sender
  - `status.MPI_TAG`: shows the tag of the received message
  - `status.MPI_ERROR`: error code

send and receive Basic MPI

Process 0 sends message msg to the rest of processes

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#define size 20

int main(int argc, char *argv[]){
  int myid, np, i;
  int tag = 0;
  char msg[size] = "";
  MPI_Status stt;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if(myid == 0){
    strcpy(msg,"Hello MPI");
    for(i=1; i<np;i++)
      MPI_Send(msg, size, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }else
      MPI_Recv(msg, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stt);

  printf("I am node %d: %s\n", myid, msg);
  MPI_Finalize();
}
```

Input/Output Basic MPI

- Typically, only one process will access the keyboard and screen
  - This process will read the data and will distribute it
  - At the end of the computations, this process will gather all data

Read and distribute data

```
if (myid == 0){
  read_data();
  distribute_data();
}
else
  receive_data();
```

Data types

Basic MPI

- The following data types are defined for MPI in C language:

```
MPI_CHAR
MPI_UNSIGNED_LONG
MPI_SHORT
MPI_FLOAT
MPI_INT
MPI_DOUBLE
MPI_LONG
MPI_LONG_DOUBLE
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_BYTE
MPI_PACKED
```

- They match those from the C language, and they add the `byte` type and the `packed` type, which allows sending different data types simultaneously

Execution time Basic MPI

Measuring execution times

```
double MPI_Wtime()
```

- Time elapsed in seconds from a given timestamp

```
double MPI_Wtick()
```

- Returns the resolution of `MPI_Wtime()` in seconds

Example

```
T1 = MPI_Wtime();
...
...
T2 = MPI_Wtime();
Runtime = T2 - T1;
```