Message Passing Parallel Programming Paradigm
Cluster Programming with MPI

Parallel Programming and High Performance Computing

Master's Degree in NTCS
University of Murcia

2022/2023

Contents

Contents

1. Introduction

- Distributed Computing
  - Computational elements are interconnected by using a network (local or distributed) → clusters of heterogeneous computing nodes.

- Programming Model: Message Passing
  - Several computational components work together to solve a problem <u>by exchanging data messages.</u>
  - Each process is assigned to a physical processor:
    - Memory positions are not shared among processes.
    - To access the memory blocks of another process, it has to communicate with them.

  - But, it also can also be used in shared memory systems…

1. Introduction

- What is MPI ?
  - Specification (API) for message-passing programming.
  - Standard for computing in distributed-memory environments.
  - Versions: MPI2, HMPI, FT-MPI...

- What does it offer?
  - Portability across platforms: shared-memory multiprocessors, message-passing multicomputers, heterogeneous systems...
  - Good performance (when using an efficient implementation)
  - Wide range of functionality: about 140 functions for the most common message-passing operations.
  - Free Open-Source Implementations: MPICH, LAM/MPI → OpenMPI.

1. Introduction

- SPMD Model
  - 1 program and several processes.
  - The code executed for each process could be different if compiled for different architectures.
  - Processes can be created statically (at the same time) or dynamically (one creates another during execution)
  - Each process has:
    - Its own memory.
    - Its own identifier (`rank`)
    - The value of the total number of processes (`np`)
  - In addition, process 0 (root)
    - Reads input data.
    - Sends input data to the other processes: `1,...,np-1`
    - Receive local data from the rest of processes
    - Combine the results and print them

1. Introduction

- Messages:
  - Depending on the number of processes:
    - Point-to-Point: one process sends and another receives
    - Globals: one process sends/receives to/from all others
  - Depending on timing:
    - Synchronous: processes wait until communication takes place
    - Asynchronous: processes do not block and continue working (while data are not available yet)
  - Data are sent/received using:
    - Elementary data types: `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_PACKED`...
    - Derived data types (created by the own developer)

1. Introduction

- Parallelization:
  - Use of functions provided by the MPI implementation:

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

- Installation (for OpenMPI)

```
sudo apt-get install openmpi-bin libopenmpi-dev
```

Includes compiling support for standard programming languages (Fortran, C/C++)

- Compilation:

```
mpicc -O3 program.c -o exec
```

- Execution:

For instance, with 2 nodes:

```
mpirun -np <num_proc> -host <node> ./exec : -np <num_proc> -host <node> ./exec
```

Contents

2. MPI: Functions

- **Environmental**

Initialize MPI:

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

Get the identifier (`MPI_Comm`: communicator (it identifies the whole group of processes)):

```
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Get the number of members in the communicator:

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

Cleanup MPI:

```
MPI_Finalize();
```

Blocks until all processes in the communicator have reached it:

```
MPI_Barrier();
```

Get the current time (in seconds)

```
MPI_Wtime(void);
```

Contents

2. MPI: Functions

- **Synchronous (Point-to-Point)**

Sending: non-blocking.

```
MPI_Send(void *buf,int count, MPI_Datatype dtype,int dest, int tag, MPI_Comm _comm);
```

Reception: blocking. Ends when the message has been received.

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

- send_recv.c

2. MPI: Functions

- **Synchronous (Point-to-Point)**

Sending: non-blocking.

```
MPI_Send(void *buf,int count, MPI_Datatype dtype,int dest, int tag, MPI_Comm _comm);
```

When does it end?:

`MPI_Send:`

when reception starts (`MPI_Ssend`)

or before reception starts (`MPI_Bsend`)

`MPI_Rsend:`

without taking into account the reception

(the receptor is supposed to be ready before sending)

Reception: blocking. Ends when the message has been received.

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

To check the number of received data elements:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype dtype, int *count)
```

When the order/source of messages does not matter:

`MPI_ANY_TAG, MPI_ANY_SOURCE`

➤ **send_recv.c**

2. MPI: Functions

- **Synchronous (Point-to-Point)**

send_recv.c

```
if (rank > 0)
{

                snprintf(mssg, LENGTH, "Hello from Process %d", rank);
                MPI_Send(mssg, strlen(mssg) + 1, MPI_CHAR, 0, 10, MPI_COMM_WORLD);
                printf("process %d sending message to process 0\n", rank);

}

else
{

        for(int i=1; i<np; i++)
        {
                MPI_Recv(messages[i], LENGTH, MPI_CHAR, MPI_ANY_SOURCE, 10, MPI_COMM_WORLD, &status);
                printf("process 0 receiving message \"%s\"\n", messages[i]);

        }

}
```

2. MPI: Functions

- **Synchronous (Point-to-Point)**

send_recv.c

Makefile:

```
N =                         # program parameter (if necessary)
NP = 1                      # number of processes per node
H1 = marte                  # first node
H2 = marte                  # second node
FILE =                      # file to compile
EXEC_DIR = exec             # directory where executable are stored

$(FILE): $(FILE).c
$(CC) $(CFLAGS) $(OMPI_INC) $(FILE).c -o $(EXEC_DIR)/$(FILE) $(DEF)

test_mpi:
mpirun -np $(NP) -host $(H1) ./$(EXEC_DIR)/$(FILE) $(N) : -np $(NP) -host $(H2) ./$(EXEC_DIR)/$(FILE) $(N)
```

2. MPI: Functions

• Synchronous (Point-to-Point)

send_recv.c

Compiling:

```
$ make FILE=send_recv
        mpicc -O3 -std=gnu99 -I/include send_recv.c -o exec/send_recv -DTIME -DDEBUG
```

Executing:

```
$ make test_mpi FILE=send_recv
        mpirun -np 1 -host marte ./exec/send_recv  : -np 1 -host marte ./exec/send_recv
        < marte >: process 0 of 2
        < marte >: process 1 of 2
        process 1 sending message to process 0
        process 0 receiving message "Hello from Process 1"
        …
```

Executing selecting parameters in command line:

```
$ make test_mpi FILE=send_recv H1="marte" H2="mercurio" NP=2
        mpirun -np 2 -host marte ./exec/send_recv 1000 : -np 2 -host mercurio ./exec/send_recv
        < marte >: process 0 of 4
        < marte >: process 1 of 4
        < mercurio >: process 2 of 4
        < mercurio >: process 3 of 4
        …
```

2. MPI: Functions

- **Synchronous (Point-to-Point)**

send_recv.c
script.pbs

```
#PBS -S /bin/bash

#PBS -V
#PBS -q batch
#PBS -N test_mpi
#PBS -l walltime=00:10:00
#PBS -l nodes=marte:ppn=6+mercurio:ppn=6

source /etc/profile.d/modules.sh
module load openmpi/1.6.4-gcc

cd $PBS_O_WORKDIR

make FILE=send_recv
make test_mpi FILE=send_recv H1="marte" H2="mercurio" NP=1 N=1000
```

2. MPI: Functions

- **Synchronous (Point-to-Point)**

send_recv.c
script.pbs

```
$ qsub script.pbs
107650.luna



$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
107650.luna                test_mpi         javiercm               0 R



$ cat test_mpi.o107650

mpicc -O3 -std=gnu99 -I/usr/local/openmpi/1.6.4/gcc/include send_recv.c -o exec/send_recv -DTIME -DDEBUG

mpirun -np 1 -host marte ./exec/send_recv 1000 : -np 1 -host mercurio ./exec/send_recv 1000

< marte >: process 0 of 2
< mercurio >: process 1 of 2

process 0 receiving message "Hello from Process 1"
process 1 sending message to process 0
```

Contents

2. MPI: Functions

- **Asynchronous (Non-blocking)**

Sending:

```
MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest,
int tag, MPI_Comm comm, MPI_Request *request);
```

`*request` allows to know if the operation has finished.

Reception:

```
MPI_Irecv(void *buf, int count, MPI_Datatype dtype,
int source, int tag, MPI_Comm comm, MPI_Request *request);
```

```
MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Waits until the requested operation has finished.

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

Returns a flag indicating whether the operation is complete.

➢ **async.c**

Contents

2. MPI: Functions

Collective

Send/Receive messages to/from all processes.
Cost of sending data could be reduced:

- Synchronous SendReceive: $(p - 1) * (t_s + n*t_w)$
- Collective: $(t_{cs} + n*t_{cw})$

```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Send a message to all group members.

```
MPI_Barrier(MPI_Comm comm);
```

Block all processes.

➢ bcast.c

2. MPI: Functions

Collective

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

root process distributes the data in separate parts (`sendcount=recvcount` elements each part) to all group members.

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

root process receives data in separate parts (`sendcount=recvcount` elements each part) from all group members.

- bcast_gather.c
- scatter_gather.c

2. MPI: Functions

Collective

`MPI_Scatterv / MPI_Gatherv`:

Distributes and collects, respectively, the data by using variable displacements between elements.

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype dtype, MPI_Op
op, int root, MPI_Comm comm);
```

Combine values from all group members and apply one of the following `MPI_Op` operators: `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD`...

➢ scatterv_gatherv.c

➢ reduce.c

➢ bcast_scatter_reduce.c

Contents

3. MPI: Other Features

- **Derived Datatypes**
  - To define new data types in MPI.
  - Useful to group the data defined in a C struct.
  - The data are arranged using an MPI function:

```
MPI_Type_struct(int count, int *array_block_lengths, MPI_Aint
*displ_array, MPI_Datatype *array_of_types, MPI_Datatype
*newtype);
```

  - ...and saved before using:

```
MPI_Type_commit(MPI_Datatype *newtype);
```

  - Created in execution time.

3. MPI: Other Features

- **Derived Datatypes**
  - Using Example:

```
typedef struct {
    float a, b;
    int n;
} IN_TYPE;
```

```
void get_data(IN_TYPE *in, int my_rank)
{
    MPI_Datatype msg_type;

    if(my_rank == 0) {
        printf("Enter a, b, n: \n");
        scanf ("%f %f %d", &(in->a), &(in->b), &(in->n));
    }

    build_derived_type(in, &msg_type);
    MPI_Bcast(in, 1, msg_type, 0, MPI_COMM_WORLD);
}
```

3. MPI: Other Features

- **Derived Datatypes**
  - Example:

```
typedef struct {
    float a, b;
    int n;
} IN_TYPE;
```

```
void build_derived_type(IN_TYPE *in, MPI_Datatype *msg_type)
{
    int blocklen[3] = { 1, 1, 1 };
    MPI_Datatype typelist[3] = { MPI_FLOAT, MPI_FLOAT, MPI_INT };

    MPI_Aint addrs[4];
    MPI_Address(in, &addrs[0]);
    MPI_Address(&(in->a), &addrs[1]);
    MPI_Address(&(in->b), &addrs[2]);
    MPI_Address(&(in->n), &addrs[3]);

    MPI_Aint disps[3];
    disps[0] = addrs[1] - addrs[0];
    disps[1] = addrs[2] - addrs[0];
    disps[2] = addrs[3] - addrs[0];

    MPI_Type_struct(3, blocklen, disps, typelist, msg_type);
    MPI_Type_commit(msg_type);
}
```

3. MPI: Other Features

- **Derived Datatypes**
  - Other Constructors:

Data of new type is made up of adjacent data of the old type:

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                       MPI_Datatype *newtype);
```

Data of new type consists of equally-spaced identical blocks:

```
MPI_Type_vector(int count, int block_lenght, int stride,
                   MPI_Datatype elem_type,
                   MPI_Datatype *newtype);
```

Data of new type are in different blocks not equally-spaced:

```
MPI_Type_indexed(int count, int *array_block_lengths,
                    int *array_disps, MPI_Datatype elem_type,
                    MPI_Datatype *newtype);
```

Contents

3. MPI: Other Features

- **Packaging**
  - To pack a set of data (of the same or different type)
  - Alternative to create data types.

  The data are packed into a buffer:

```
MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *buffer,
int size, int *position, MPI_Comm comm);
```

- The data are taken from `inbuf` copied into `buffer+position`
- `position` is an input/output parameter whose value is managed internally by MPI.
- As output, `position` references to the next position in the buffer.
- `size` must be expressed in bytes (`sizeof...`)

3. MPI: Other Features

- **Packaging**

To pack a set of data (of the same or different type)
Alternative to create data types.

And unpacked in the same order they were packed:

```
MPI_Unpack(void *buffer, int size, int *position,
void *outbuf, int outcount, MPI_Datatype datatype,
MPI_Comm comm);
```

As output, `position` references to the next position in the buffer.

The data are taken from `buffer+position` and copied into `outbuf`

➢**pack_unpack.c**

Contents

3. MPI: Other Features

- **Communicators**
  - `MPI_COMM_WORLD` represents all MPI processes.
  - It is also possible to create communicators with a lower number of processes.
  - An MPI communicator consists of:
    - Group: sorted collection of processes with an assigned rank between 0 and p-1
    - Context: identifier assigned to a group by the system

3. MPI: Other Features

- **Communicators**
  - Example of Creation of an MPI Communicator:

`MPI_COMM_WORLD` = q^2 processes grouped in a *qxq* grid

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group row_group;
MPI_Comm row_comm;

int row_size;
int *process_ranks = (int *) malloc(q*sizeof(int));

for(int proc=0; proc<q; proc++) { process_ranks[proc] = proc; }

MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, &row_group);
MPI_Comm_create(MPI_COMM_WORLD, row_group, &row_comm);
```

`MPI_Comm_group` + `MPI_Group_incl` are local functions → No communications

`MPI_Comm_create` is collective. All processes in `MPI_COMM_WORLD` must execute it even if they are not part of the new group.

3. MPI: Other Features

- **Communicators**
  - Communication Types:
    - Intra-Communicators:
      - To send messages between processes within the same communicator.
    - Inter-Communicators:
      - To send messages between processes located in different communicator.

Contents

Contents

4. References

Chapter 3 from Almeida, F., Giménez, D., Mantas, J-M., Vidal, A. *Introducción a la Programación Paralela*, Paraninfo, 2008.

The MPI Forum:
https://www.mpi-forum.org/

The Message-Passing Interface Standard:
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

The Open MPI Project:
https://www.open-mpi.org/

The OpenMPI API Documentation:
https://www.open-mpi.org/doc/current/

Contents

5. Appendix: Hybrid Programming

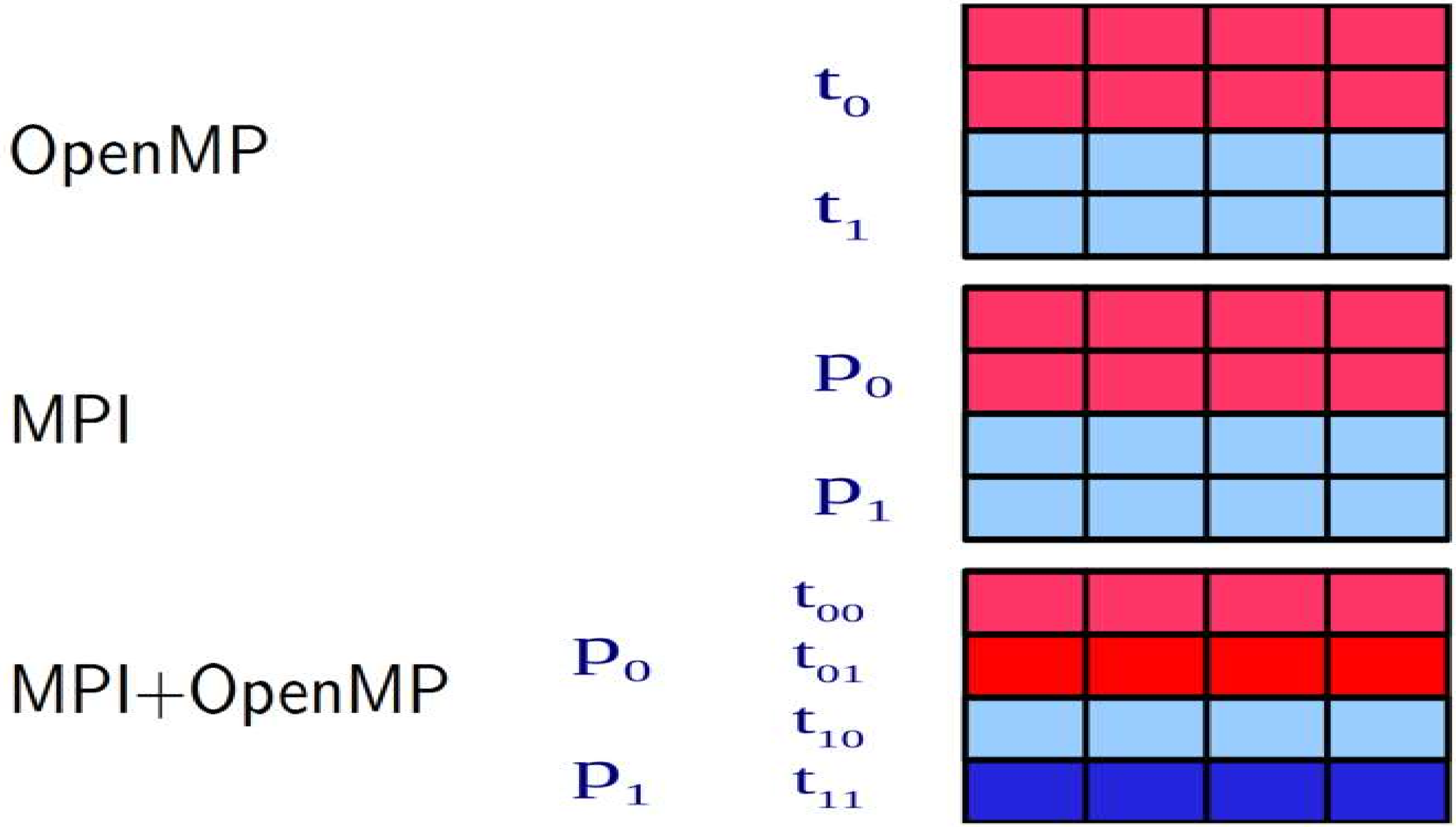
- MPI + OpenMP:
    - Example of Multilevel Parallelism.
    - MPI processes are assigned to compute nodes (Distributed Memory) with one or more processes per node.
    - Each MPI process creates a set of OpenMP threads inside the node (Shared Memory)
        - Can also be combined with OpenMP nested parallelism to efficiently exploit the memory hierarchy levels.

5. Appendix: Hybrid Programming

```
int main(int argc, char *argv[]) {

    int nthr, tid, nproc, rank, len;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Get_processor_name(name, &len);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    omp_set_num_threads(omp_get_max_threads()/nproc);

    #pragma omp parallel private(tid) firstprivate(rank, nproc)
    {
        tid = omp_get_thread_num();
        printf("thread %d in process %d of node %s\n", tid, rank, name);

        if (tid == 0) {
            nthr = omp_get_num_threads();
            printf("running %d processes with %d threads\n", nproc, nthr);
        }
    }

    MPI_Finalize();
    return(EXIT_SUCCESS);
}
```

5. Appendix: Hybrid Programming

- **Matrix Multiplication**



5. Appendix: Hybrid Programming

```
int main(int argc, char *argv[]) {

    int len, rank, np;

    MPI_Init(&argc, &argv);
    MPI_Get_processor_name(name, &len);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        N = atoi(argv[1]);
        NUM_THREADS = atoi(argv[2]);

        // Send N and NUM_THREADS to each process (from 1 to np-1)
           ...
    }
    else {
        // Receive N and NUM_THREADS from Process 0
           ...
    }

    // Each MPI process sets its own number of OpenMP threads
    omp_set_num_threads(NUM_THREADS);
```

5. Appendix: Hybrid Programming

```
   ...
// Allocate size for matrices A, B and C
   ...

if (rank == 0) {
     // Send matrix B to all processes
        ...

     // Send the portion of matrix A to each process
        ...
}
else {
     // Receive matrix B and the portion of matrix A
        ...
}

MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();

    mm_par(N, my_A, my_B, my_C);      // Parallelized with OpenMP

MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
```

5. Appendix: Hybrid Programming

```
        ...

    if (rank == 0) {
        // Receive each portion of matrix C from process 1 to np-1
           ...
    }
    else {
        // Send the portion of matrix C to process 0
           ...
    }
        ...

    if (rank == 0) {
        // Print Elapsed Execution Time
           ...
    }

    // Free Allocated Memory
           ...

        MPI_Finalize();
        return(EXIT_SUCCESS);
}
```