Practice 2
Parallelism in distributed memory systems

Objectives:

- Learn how to parallelize an application in a distributed memory system using the "*message passing*" parallel programming style.
- Study the MPI API and learn how to apply **different** parallelism strategies with it.
- Apply methods and techniques specific to CE to estimate the maximum speed-ups and efficiencies of the parallelization process.

Introduction

In this practice we are going to study and implement **parallelism in distributed memory** systems, using the **MPI** standard. Therefore, we will learn how to parallelize a sequential application to improve its performance in distributed memory parallel machines.

MPI specification:

MPI (*Message-passing interface*) is a specification that establishes the functions of a library for message passing between multiple processors/processes. This allows communication via message-passing between nodes in a cluster of computers that constitute distributed memory multicomputer systems[1] . In this scenario, the data used by one process is moved from its address space to that of another process, in a cooperative manner, by means of the instructions contained in the MPI standard.

We can distinguish **4 types of instructions** defined in the MPI specification or API:

1. To open, manage and close communications between processes running on different nodes.
2. To transfer data between 2 processes (one to one).
3. To transfer data between multiple processes.
4. Instructions that allow the user to define data types.

The main advantages of using MPI are that it allows the establishment of a standard for message pass ing in portable multicomputer architectures, easy to use and that allows the abstraction of the low-level details involved in the management of this type of communications.

There are many libraries that implement the MPI standard (mpich, OpenMPI, ...) in different languages such as C, C++, Rust, Fortran, Python, Matlab, etc... In this link you can find a tutorial with examples of MPI usage from the Lawrence-Livermore lab.

1 Refer to Unit 1 slides on Flynn's Taxonomy.

Task 2.1: Pre-study and MPI training (1 session)

In this task you should look for information about the different implementations of the MPI standard on C/C++/Fortran and other languages such as Rust, Python, Matlab, etc. There are important companies, such as IBM, Cray or SGI, that have their own MPI implementation to be used in their parallel machines. There are also different free software projects that implement MPI and for which its use is widespread in both academia and industry (mpich, OpenMPI, ...).

Task 2.1.1 Comment on the differentiating characteristics of each implementation.

In these links you can find tutorials with examples of MPI usage.

- https://www.youtube.com/watch?v=c0C9mQaxsD4
- https://www.youtube.com/watch?v=OGez4VNYhJA
- https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php *(University of Granada)*
- https://www.youtube.com/watch?v=c0C9mQaxsD4 *(UPV)*
- https://www.youtube.com/watch?v=0FxDKhRxQqU *(UPV)*
- https://www.youtube.com/watch?v=y1yB7LTn6oA *(UPV)*

Task 2.1.2 We are going to make a simple program to test the correct operation of MPI between at least 2 nodes in the laboratory. The program should perform some kind of **interactive communication** between the two nodes. We propose the following program, although it can be complicated/improved as you deem appropriate:

Example program: A node, which we will call N1, requests a value of type integer from another node (N2) in the network. For them, N1 will report in the standard output the following "Requesting the length of the vector to node N2...". N2, at this point, should ask the user something like "Message from N1: Enter vector size: ". After N1 successfully receives that number (N), it should calculate the sum of the first N natural numbers and send the result back to N2. N2, in turn, must print the result on the screen and notify N1 that it has completed its task. N1 terminates the parallel program.

Task 2.1.3 Implement the following programs in sequential and parallel with MPI on a distributed memory parallel machine:

- Calculation of PI as the definite integral between 0 and 1 of the derivative of the arctangent. See the link: https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=03_pi
- Calculation of PI using the Monte Carlo Method. See the link https://www.geogebra.org/m/cF7RwK3H

Answer the following questions:

- Is the problem fully parallelizable?
- What is the maximum expected speed gain?
- What about efficiency?

Calculate the following times for each program:

- Sequential time of the program without parallelization.
- Parallel time of the parallelized program executed in only 1 node (N=1) and assuming that P=1,2,3,4,5,6 ... That is, the program is parallelized, but we are simulating in the laboratory

machine the behavior it will have once deployed in several nodes of the network. Plot these times and comment on the results. Is speed gain observed? Why?
- Parallel time of the program executed in parallel on several nodes in the lab. Is speed gain observed? Why?

Task 2.1.3 Review the slides of **Unit 3** that you already have in Moodle (from number 28 onwards). Specifically, the ones that talk about the different **communication alternatives** between various processes.

Find how MPI (specifically mpich, the version they have installed in the lab) implements the following communication alternatives. Also explain what is done in each of them:

- one-to-all:
    - Broadcast
    - Dispersion
- all-at-once:
    - Reduction
    - Accumulation
- all-to-one, etc.

Development of one or more **simple** programs to test the correct operation of these functions. At least **6** of the **communication alternatives** provided by the mpich API should be used.

Task 2.1.4 Develop 2 parallel programs following the **SPMD** and **MPMD programming mode**, respectively. Review the Unit 3 slides to recall what a parallel programming mode is. These programs must calculate, given a vector of N elements of type `double` or `float`, the maximum, minimum and average of all elements in the vector. Check on **real nodes** that the program works correctly.

Task 2.2: Study and parallelization of an application with MPI (1 session)

Attached to this practice you will find the file *lab2_files.zip*, which contains 3 files:

- `main.c`: Code to be parallelized
- `processing.h`: C header file using *main.c*
- `libprocessing.so`: dynamic library with the implementation of several functions.

In the `main.c` file you will find a simple code that calls 4 functions, sequentially, that process a vector of 100 elements (you can change the length of the vector if you wish). The signature of each of the 4 functions called has the following form:

```
void process_block1 (int group, int* vector, int LEN, int start, int end);
```

where:

- `group` is the number of the group of practices to which they belong (e.g. for the group of Mondays from 15-17h group is 5, for the group of Mondays from 17-19h group is 6, etc.),
- `vector` and `int_elements` are the pointer to a 32-bit integer vector and its length, respectively.
- Finally, `start` and `end` are the indices (`0..LEN-1`) between which you can process the vector. For example, if I want to process only element 8, I would specify `process_block1(GROUP, vector, LEN, 7,7);`
- The processing of each element of the vector (which is hidden) does not depend on previous calculations, only on the vector element in question.
- The functions do not return anything, they only perform some processing that you should speed up.

NOTE: You must compile taking into account the dynamic dependency `libprocessing.so` (which will not be found by default in the system path for these libraries). You must update the `LD_LIBRARY_PATH` environment variable to **compile** and **run** correctly.

For example, try compiling like this (assuming the 3 files are in the same directory):

```
gcc -Wall main.c -L. -lprocessing -o main.elf
```

Perform the **profiling** you deem appropriate to study the time behavior of the program without parallelization (t_{seq}). Parallelize the code with MPI assuming that it consists of at least 2 and 3 nodes of the laboratory to calculate the *speedup*.

Tasks:

2.2.1. Calculate the sequential time of the unparallelized version (t_{seq}).

2.2.2. Propose an optimal alternative to optimize the speed using MPI. Calculate, with this implementation, the time for P=1, 2 and 3 nodes **on the same machine in the lab**. Is any gain observed?

2.2.3. Calculate the gain in speed (S_p (p)) for at least 2 and 3 different nodes in the laboratory.

2.2.4. Calculate the parallel efficiency in parallel execution with real nodes and using several processes within the same node (as you did in 2.2.2).

Note:
It is important to argue all the necessary changes to parallelize the starting sequential solution. The **performance analysis must be exhaustive and detailed** (different sizes of the problem, different number of nodes in the cluster...).

General notes to the practice:

- The implementation will have to be able to run **under the Linux operating system of the laboratory**, and will take 4 sessions (8h)
- **It is mandatory to** deliver a ***Makefile*** with the appropriate rules to compile and clean your program (`make clean`) in a simple way.
- The students will deliver, in addition to the application developed, a report, structured according to the professor's indications, with the information obtained.
- **Delivery**: the week of November 11 before the start of your corresponding practice session.
- The theoretical/practical work must be original. The detection of copying or plagiarism will result in a grade of "0" in the corresponding test. The direction of the Department and the EPS will be informed about this incidence. Repeated misconduct in this or any other subject will lead to the notification of the corresponding vice-rectorate of the faults committed so that they can study the case and sanction according to the legislation.

Help: Steps to compile and run in the lab (this information may vary from course to course):

1. All machines must have the **same user**. All executables must have the **same name** on each machine and be in the **same folder**.
2. Compile using mpicc or mpic++:
   ```
   mpicc program.c -o program
   ```
   Remember that the program, along with its dependencies, must be copied and accessible on each node of your multicomputer.
3. Create the **hosts file**[2] with the IPs or names of each node in the network and copy it to the rest of the machines in the same working directory where the executables are located.
4. Create an ssh public key certificate on one of the nodes:
   `ssh-keygen -t rsa` *(creates the hidden folder ~/.ssh)*
5. Copy the `~/.ssh/id_rsa.pub` file to the `~/.ssh` folder on all other machines and rename it to `authorized_keys`. If the hidden `~/.ssh` folder does not exist on a machine, create it beforehand by running the command `ssh-keygen -t rsa`.
6. Run the program on the machine where the certificate was created:
   ```
   mpirun -mca plm_rsh_no_tree_spawn 1 -hostfile <file_hosts>
        -n <process_number> ./program
   ```

Remarks:

- Use the same directory on all machines: e.g. Personal Folder (`$HOME`)
- Login with the **same user on all nodes** in the cluster
- It may be necessary to modify file permissions:
  - Hosts file: `chmod 600 <hosts_file>`.
  - Executable file to allow remote execution: `chmod o+x program`

MPI Hello world:

```
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>
#include <stdlib.h>

int sched_getcpu();

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);
  printf(" PID: %d Hello from process %d out of %d on %s processor %d\n",
           getppid(), rank, numprocs, processor_name, sched_getcpu());
  if (rank==2)
  {
     printf("Hello! from processor %d\n", rank);
  }
  MPI_Finalize();
}
```

2**Hosts file**: plain text file that specifies the name or IP of the machines that make up the supercomputer. Each name/IP on one line.