Assignment #3 - MPI & Computation Graphs

This assignment aims to make you experience the basic MPI routines for sending and receiving messages among processes and the concepts we have seen in class about the Computation Graphs.

NOTES FOR WINDOWS USERS

This assignment is designed for OpenMPI and Linux-based machines. As an alternative, there is a Microsoft version of MPI (MS-MPILinks to an external site.) but I cannot guarantee the two implementation are fully compatible and I cannot provide you assistance with the installation (in addition, you probably will not able to use straightforwardly the makefile provided). Another alternative is to install OpenMPI from Windows WSL (as described below for the Linux-based systems). Mind that, once you have installed WSL you need even a compiler. You should get a notification during the installation process of OpenMPI. If not, you can do it manually:

sudo apt install g++

You can use environments like vscodeLinks to an external site. to edit the code.

MPI'S DESIGN FOR THE MESSAGE PASSING MODEL

MPI is a library providing parallel programs with the ability to communicate through the message passing paradigm. The first MPI concept we introduce, is the *communicator* which defines a group of processes that are allowed to exchange messages among them. Within a communicator, each process is assigned to a unique rank which falls in the range from *0* to *N*-1 where *N* denotes the number of processes belonging to the communicator.

Let's start now with the MPI version of the classic "hello world" program:

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

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv); // init the MPI environment
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size); //get the size of the communicator
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get the rank of the process
  printf("Hello world from processor with rank %d out of %d processors\n", rank, size);
  MPI_Finalize(); // finalize the MPI environment
}
```

Save the program above in the file "hello.cpp" and compile it as it follows (Window users: please read the note at the end of this page):

```
mpicxx hello.c -o hello
```

If you still have to install the MPI packages, you can do it with the following command:

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

Once you are done with the installation and the compilation, you can launch the parallel program with the following command:

```
mpirun -q --oversubscribe -n 4 ./hello
```

Let's start with the explanation... In the file "hello.cpp", during "MPI_Init", all the MPI variables, like for example MPI_COMM_WORLD and rank of the processes, are initialized. As a consequence this function must be called before any other MPI function. MPI_COMM_WORLD is the default communicator (instantiated by MPI for us) that includes all the processes created when the parallel program is launched. The initial number of processes is set by means of the option "-n" which follows the "mpirun" command. The processes created by "mpirun" are copies of the same program: the "only" thing which differs is the rank assigned to each process. Based of its own rank, a process can differentiate its behavior with respect to the others (see the next example) and identify its partner in the communication routines . In this case, since the number of processes is 4, the output of the "Hello World" program looks like:

```
Hello world from processor with rank 0 out of 4 processors
Hello world from processor with rank 1 out of 4 processors
Hello world from processor with rank 2 out of 4 processors
Hello world from processor with rank 3 out of 4 processors
```

"MPI_Comm_size" and "MPI_Comm_rank" return, respectively, the size of a communicator and the rank of a process in a communicator passed as parameter. Ranks are mainly used for identifying sender and receiver of a message.

MPI SEND AND RECEIVE

MPI's send and receive are two coupled functions which means that to each send corresponds one receive and vice-versa, otherwise the program stalls. When a process, let it have the rank #0, needs to communicate some data to the process with rank #1, all necessary data and meta-data are packed into a buffer to be sent to the process #1. Let's see a simple program implementing a basic schema of communication:

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


int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int rank, size, message;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const int tag=16, sender=0, receiver=1, count=1;
  if (rank==0) {
    message = 10;
    MPI_Send(&message, count, MPI_INT, receiver, tag, MPI_COMM_WORLD);
  }
  if (rank==1) {
    MPI_Recv(&message, count, MPI_INT, sender, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received %d from process 0\n", message);
```

```
  }
  MPI_Finalize();
}
```

For both "MPI_Send" and "MPI_Recv" the first parameter is a pointer to the memory location that store the message content (in some special cases, it may also be NULL with the associated counter equal to zero). The next two parameters respectively count the number and specify the type of the message and are used, for example, to calculate the amount of memory needed by the buffer. In the case above, we exchange one MPI_INT. Other basic types are:

MPI datatype	**C equivalent**
MPI_SHORT	short int
MPI_INT	int
MPI_LONG_LONG	long long int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	char

Apart for them, programmers can define their own type for more complex data structure. Mind that since each process has its own address space, we are not allowed to send pointers. The next parameter in the functions identifies the partner of the communication that is the receiver rank for "MPI_Send" and the sender rank for "MPI_Recv". After that tag and communicator are specified. The tag is needed to correctly match a pair of send and receive when more messages are sent between a couple of processes in the same communicator. Communicators have been already introduced. For "MPI_Recv" there is a further parameter denoting the status of the message (e.g., the status stores the rank of the sender when the receive routine uses the wild-char MPI_ANY_SOURCE) that we can ignore for the moment. As a final remark a brief explanation of "MPI_Send". It implements a communication which is not strictly synchronous as expected. Depending on the size of the message, the function may work asynchronously, i.e., the message is buffered somewhere and the function returns before the message has definitively left the sending process. In any case, it is always allowed to reuse the message buffer as soon as "MPI_Send" returns.

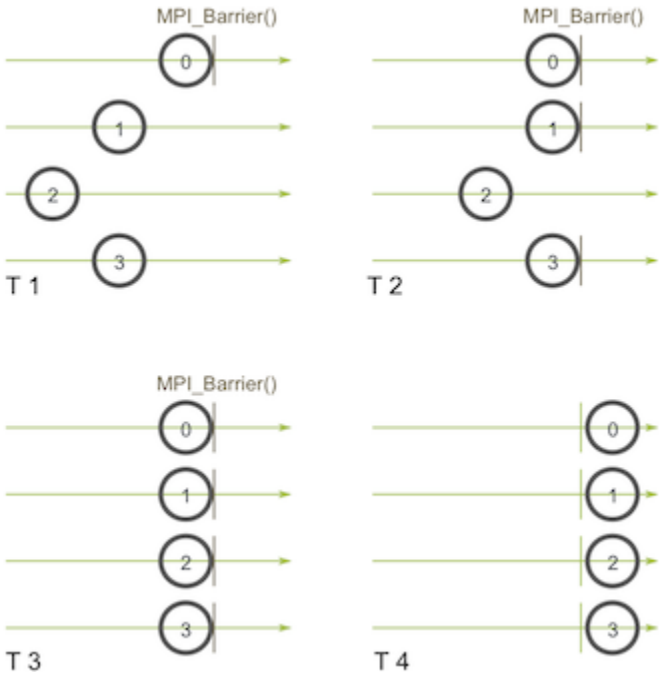
This assignment is designed to make you practice with the *message passing* paradigm. As a consequence, it's not allowed to use OpenMP to parallelize a solution, but exclusively the MPI library.

First of all download and extract lab3.zip Download lab3.zip. To run each task, you can use the make utility relying on the makfile you found in the zip file (you can use the option -B to force the make to work even when the target file is up to date). For example:

make -B seq

TASK #1

We have seen in class what a collective communication is and how to calculate the related *depth.* In general all collective communications implicitly requires a synchronization point, so-called *barrier*. The MPI provides even a function that implements this functionality: MPI_Barrier(). A barrier is an operation that performs a synchronization among the processes belonging to the given communicator. In other words, it guarantees that the involved processes proceed with their execution only when all of them have reached the barrier.



Implement the barrier function in the file "barrier.cpp" by means of point-to-point communications and write a comment in your code reporting the related depth and a brief explanation of it. If your barrier works properly, the output should be like: ***n*-process barrier test passed**.

Now you should be enough familiar with the MPI to be able to solve the next tasks dealing with the implementation of compute graphs. Look at the code in the file "seq.cpp" and "common.hpp". Before you start, read carefully all the instructions and understand how the given program works. This system consists of one module (called *M* in the code) which is connected to a pair of external processes *in* and *out*. The overall computation consists in a set of "fake functions" having a nearly null variance (as a consequence, we can implement the *communication channels* by means of synchronous *send* and *receive* calls without the aid of message buffers). Analyze the code and the service time of each process then calculate the expected completion time for the given input stream. Once you are done, run the code and check if the elapsed time reported on the screen is almost the same as the expected one.

For the next tasks, you have to parallelize **efficiently** (i.e., by getting the ideal performance in each case without wasting the number of processes implied) the given system in three different ways. For each task, create a new file and remember that neither the file

"common.hpp" nor the semantic of the computation can be modified, i.e., your solution has to be implemented by means of the data structures and communication functions defined in "common.hpp". It's also worth noting how the different *metrics* (i.e., completion time, average service time, and average latency) vary with respect to the parallelization schema used in each task.

TASK #2

Now you have to parallelize the given program in order to speedup the computation. To this end redesign *M* as a *pipeline* and, when you are done, calculate the expected *metrics* (i.e., completion time, average service time, and average latency). Finally implement your solution, run it, and check if the measurements reported as output confirm with your expectations.

Notes, hints, and requirements:

- **1 process runs exactly 1 module-function**. Modify the value of the variable "required_comm_size" according to the optimal number of processes your solution needs (e.g., in the initial case it's 3 since you need three processes for the functions *in*, *M*, and *out*). The makefile will use that information to create the corresponding number of processes.

TASK #3

The pipeline implemented in TASK #2 improved the performance of the parallel computation: still we haven't reached yet the optimal one. Hence, instead of the pipeline, parallelize the initial program by means of a *farm.*

In addition to the notes and restrictions of TASK #2:

- The implementation of the collector for a farm requires that it is able to accept messages from any workers in any order. The function is already implemented in the "common.hpp" file with the name *recv_any* which returns the process id of the sender.
- For the emitter of the farm you can implement a simple round-robin approach for dispatching the stream elements to the workers or a on-demand policy.
- A typical problem of compute graphs is how to propagate the termination signal to all modules. Termination signal stands for "end of the input stream" (look at the modules in "seq.cpp"). For farm, once the Emitter has received the termination-message, it forwards the same message to each worker then returns. Even, workers forward the termination-message then return. Collectors, instead, have to use a counter variable that keeps track of the number of termination-messages received: when the number is equal to the number of workers, even the collectors can propagate the termination-message and return.
- Define the value of the variable "required_comm_size" by using a formula based on the given service time of each function. In "common.hpp", you can find the macro functions DIV and CEILDIV that you can use to write such a formula.

TASK #4

Re-do TASK #3 by means of a *map* instead of a farm.