Arquiteturas de Alto Desempenho 2024/2025

Practical class AAD_P05 (2024-10-21 and 2024-10-22)

Tomás Oliveira e Silva

1 Objectives

Learn how to parallelize code across multiple machines the easy way (Message Passing Interface, `MPI`, the *de facto* standard for high-performance parallel computing on large supercomputers). List of things you will learn about:

- how to compile and run a `MPI` program on a single compute node and on multiple compute nodes, and
- how to use the basic `MPI` communication functions and the pitfalls of some of them.

2 The Message Passing Interface standard

In previous practical classes we explored the possibility to running parts of our program in parallel on the same computer. We used `pthreads` (hard) and `OpenMP` (easy) to do that. The programs written with them are called shared memory programs because their parallel parts share the same address space.

In the previous practical class we explored the possibility of running parts of our program on a different process, possibly running on a different computer. We only explored a server/client paradigm, in which one of the processes is the server and the others are clients. We used sockets to handle the communication between the server and each one of the clients, and we had to manually start each process and tell the clients the internet protocol address of the server. Each connection to the server was ephemeral (not persistent) and client–client communications were not allowed. The server/client paradigm is sometimes useful, but there exist many problems where it is not useful because the communication patterns between compute nodes are decentralized. Indeed, the communication graph in the server/client paradigm is a star, with the server at the center, and with each client connected only to the center.

When more complex communication patterns are required, setting up all communication links by hand can be cumbersome. Furthermore, using sockets may not be the most efficient way to handle the communication links (for example, many supercomputers use InfiniBand to interconnect compute nodes). Because (super)computers of different vendors may use different communication technologies to handle inter-node and even inter-process communications, to write portable programs it became necessary to devise a standardized way to

- launch a parallel program on one or more compute nodes, with one or more processes on each node, and to
- handle the explicit communication between processes, on the same node and between nodes.

All relevant players eventually agreed to a standard, called The Message Passing Interface, `MPI`. Nowadays, it is the *de facto* standard for high-performance parallel computing on large supercomputers.

In `MPI`, each process may have multiple threads (program counters and associated stacks) that share a single address space. So, a `MPI` program can still use `pthreads` or even `OpenMP`. `MPI` is used for communication among processes, which have separate address spaces. Inter-process communication handles synchronization (when needed) and movement of data between address spaces. `MPI` handles the movement of data via messages. All parallelism is explicit: the programmer has to devise a way to split the work among nodes and then has to specify all communication messages that are necessary to get the work done.

When a MPI program is launched to be run using N processes (N copies of the program), each process gets its own number, the rank R, with $0 \leqslant R < N$. The rank can be used to define which part of the computation the process will work on. Ranks are also used in communication messages, to tell the destination of each message.

There exist two mostly compatible MPI implementations: openmpi and mpich. We will use openmpi in this class. The following links are useful:

- general information about openmpi (clickable link).

  URL: https://docs.open-mpi.org/en/v5.0.x/
- manual pages, section 1, of the commands used to launch MPI programs (clickable link).

  URL: https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man1/index.html
- manual pages, section 3, of all MPI functions (clickable link).

  URL: https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man3/index.html

The manual pages are sometimes terse but are better than nothing. The following book is also a very useful source of information:

> William Gropp, Ewing Lusk, and Anthony Skjellum
> Using MPI: Portable Parallel Programming with the Message-Passing Interface
> second edition, 1999, MIT press

3 Our first MPI program: hello world

Before we can compile and run our first MPI program we need to install openmpi. On a Debian-based GNU/Linux OS, this can be done using the command:

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

You should now have in your system the mpicc and mpirun commands, and the manual pages of the MPI functions:

```
$ mpicc --version
$ mpirun --version
$ man MPI_Init
```

It is now time to take a good look at the contents of the hello_world_mpi.c file. It contains the code of the hello world program. Observe how the message passing interface is initialized and terminated. All MPI program must do this. Observe also how the number of processes and the rank of the process are retrieved. In this example, we also retrieve the host name where the process is running, but that is just for information; a typical MPI program does not need to do this. Each process prints the usual “hello world” message, after which it prints the number of processes, the rank of the process, and the host name.

Take also a good look at the makefile, to see how a MPI program is compiled and how to run it.

To compile and run it, do

```
$ make run_hello
```

You should see an output like the following:

```
mpicc -Wall -O2 hello_world_mpi.c -o hello
mpirun -n 4 ./hello
Hello world from 1/4 [tosh]
Hello world from 3/4 [tosh]
Hello world from 2/4 [tosh]
Hello world from 0/4 [tosh]
```

In the `mpirun` command, `-np 4` specifies that four processes should be launched. Since no `--host` nor `--hostfile` command line options were given, the four processes are to be launched on the local host.

3.1 Optional: launching processes in another computer

If you and a friend are up to it, and if you have a mobile hot spot, you can try to launch the processes spawned by `mpirun` on your computer and on your friend's computer. To do that, however, you need to install the `sshd` server on your friend's computer, and you will have to generate a private key and corresponding public key to allow `ssh` operations without entering a password. After installing the `sshd` server on your friend's computer

```
$ sudo apt install openssh-server
```

generate the private and public keys (if you already have the files `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub` use another file names and adjust what follows accordingly):

```
$ ssh-keygen
```

Just press return three times to give the default answer to the `ssh-keygen` questions. You should now have the files `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`. Append the contents of the `id_rsa.pub` file to `~/.ssh/authorized_keys` on your friend's computer. This can be done using, for example,

```
$ cat ~/.ssh/id_rsa.pub | ssh [remote_username]@[remote_ip_address] 'cat >> .ssh/authorized_keys'
```

To make things easier, on your computer edit the file `~/.ssh/config` (create it if it does not yet exist), and place there something like the following (replace names where appropriate):

```
Host friend
  HostName      your_friend_ip_address
  User          your_friend_login_name
  IdentityFile  ~/.ssh/id_rsa
```

After this, to see if all is set up properly, you can do, for example,

```
$ ssh friend hostname
```

This should give the true hostname of your friend's computer. Because you do not have a shared file system, you will need to copy the hello program to the home directory of your friend's computer. After that, you are now, **finally**, ready to try `mpirun` out. Just do

```
$ mpirun -np 8 --host localhost:4,friend:4 ./hello
```

This should create and run 8 processes, 4 on your computer and 4 on your friend's computer.

On a supercomputer, when you launch a `MPI` program, a set of free compute nodes is selected, the `sshd` configuration files are adjusted to reflect the selected node names, and only then the program is actually launched. The description of the previous chain of events assumes that processes are launched using the `ssh` program. That may not be true, but if not similar actions are performed using whatever program is being used to run the program in each compute node.

In the end, clean up (undo the changes in the `~/.ssh` directory, remove the `hello` program from the home directory, and uninstall the `sshd` server). Your friend will almost certainly not want you to be able to run whatever program you like on her/his computer :-)

4 One-to-one communication

The one_to_one_communication_mpi.c program does three things (after the MPI initialization and before its termination):

1. In the first part:
   - The process with rank 0 sends a message (10 integers) to the process with rank 1. Besides the data type and number of items, each message has to specify a so-called tag (message id, chosen by the programmer), and a so-called communicator. A communicator specifies the set of processes that can be recipients. In this class this set will always be all processes, and that is specified in MPI using MPI_COMM_WORLD.
   - The process with rank 1 receives a message send by any process (even itself), with any tag, with at most 10 integers. It prints information about the received message (who send it, with which tag, and the data received).
   - All other processes do nothing.
   - Because there is only one sender, of course the rank 1 process will receive the message send by the rank 0 process.
2. In the second part we apply a barrier. The processes stop at be barrier until all have done so. After that, execution continues. This was placed in the program just to show how this can be done. The program will also run without it.
3. In the third part, the process with rank R sends a message to the process with rank $(R+1)$ **mod** N, and after that, it tries to receive a message for the process with rank $(R-1)$ **mod** N. This is done for all processes. The communication graph for this is a ring. The way this is done in the code is a **common MPI programming error**. It may work, or it may give rise to a deadlock. This is so because the MPI_Send() function, used to send a message, may block; depending on the MPI implementation, if the message is too large MPI_Send() may return only after the message is received. In the code, all processes first send and only later receive, so this deadlock may happen. (Short messages may be handled differently, and that is why the code as is probably does not give rise to a deadlock.)

Study it with care, compite and run it.

Replace MPI_Send() with MPI_Ssend(), which is a synchronous send, which will definitely block until the message is received. Observe the deadlock. Follow one of the suggestions of the comments placed in the third part of the code to avoid the deadlock.

5 One-to-all communication

The one_to_all_communication_mpi.c program illustrates how data can be broadcast from one process to all processes.[1] Instead of doing this for one of the fundamental data type — MPI_CHAR, MPI_INT, or MPI_DOUBLE — this is done for the data stored in a C structure. Study the program to see how this is done.

6 Once again: parallelization of MandelbrotCount.c

It is now time to put all that you learned in this class in action. Modify the MandelbrotCount.c to do the computation in parallel using MPI.

[1]There exist other communication patterns, including reductions. In these practical classes we do not have time to describe them all. If you are curious, and the teachers hope that you are, look them up and study them.