HPC - MPI - introduction & point-to-point communication

Table of Contents

1 Introduction

MPI is an API for distributed programming based on exchange of messages. The first version of the standard was proposed in 1993. MPI is updated every few years to cover new use-cases and new architectures. It is used extensively in High Performance Computing.

MPI programs are executed as multiple processes (usually running on different machines). MPI standard defines functions for communication between these processes. An MPI implementation (such as MPICH) implements these library functions and also provides tools to compile and launch MPI programs conveniently.

Files for today: mpi-lab-01.zip

Google Colab Notebook: draw-bandwidth.ipynb

2 Program structure

```
#include <mpi.h>

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

   MPI_Init(&argc,&argv); /* intialize the library with parameters caught by the runtime */

   /* parse program arguments */
   /* do something: */
   /* MPI communication goes here */

   MPI_Finalize(); /* mark that we've finished communicating */

   return 0;
}
```

`MPI_Finalize()` cleans up MPI state, so it must be run on all execution paths.

- extend `hello-world-seq.c` with these functions and try to compile it on one of lab machines `green`*, `red`* or `yellow`*. Warning: the MPI version at `students.mimuw.edu.pl` is incompatible with the version the lab machines have. Thus, you need to compile and run directly on one of the lab machines.

3 Compiling and running

3.1 Distributed, ssh-based environments

3.1.1 Compilation

An MPI implementation provides its own wrapper for a C compiler. The wrapper is usually called mpicc.

- change the makefile and compile `hello-world-seq.c` on one of the lab machines `red`* or `yellow`*.
- homework: install mpich on your machine. There are easy to install packages for most linux distributions and for mac osx.

3.1.2 Running

```
mpiexec -n <N> <program_path>
```

Where `<N>` is the number of processes to launch. `mpiexec` will launch all the processes on a single machine.

To launch processes on different hosts, you have to be able to login to each of these hosts through ssh from the host where you start the executable. Here, assume that we use `yellow01` and `yellow02`:

```
students$ ssh yellow01
yellow01$ ssh-copy-id yellow02
yellow01$ ssh yellow02 # you have to log in manually once every 24h or so
yellow02$ logout
yellow01$ ssh yellow02 # now you shouldn't have to type the password
yellow02$ logout
```

Now, your environment is prepared to run MPI via SSH:

```
yellow01$ mpiexec -n 2 -host yellow01,yellow02 <program_path>
```

3.2 Compiling and running on a supercomputer

Warning: remember that a supercomputer is an expensive machine normally used for serious scientific problems. Be reasonable. We will have a reservation of a number of nodes for our labs; we all share machines in this reservation. If you take too many machines for too long, others won't be able to use them (and they'll know who you are!). Be reasonable.

3.2.1 Accessing the supercomputer

Before first login, setup 2FA following instructions on page: https://kdm.icm.edu.pl/Tutorials/Logowanie/ssh.en/

```
ssh username@login.icm.edu.pl
ssh okeanos
```

`okeanos1` is the head node for our supercomputer (to confuse users, you login to `okeanos`, but work on `okeanos1`).

With emacs, you can use tramp mode to remotely edit files on your local machine. To open a connection to okeanos, use the following filename: `/ssh:account@login.icm.edu.pl|ssh:account@okeanos:/`.

For other IDEs, like VSCode, we suggest adding the following to your `.ssh/config` file:

```
Host okeanos
        User <okeanos-login>
        HostName okeanos
        ProxyCommand ssh <okeanos-login>@login.icm.edu.pl -W %h:%p
```

Providing <okeanos-login> twice is required when it differs from your local name, as otherwise it is ignored during proxy jump. Now you should be able to login to the "service node" directly by using `ssh okeanos` command.

From now on you may use common solutions for working with remote machines, for example `Remote - SSH` plugin for VSCode.

3.2.2 Compiling

You compile the code on the head node.

Hint: for development, have a git repository that you clone on the head node, but develop the code somewhere else.

Okeanos' environment wraps the default C compiler `cc` to `mpicc` provided by okeanos' vendor, Cray. The C++ compiler is invoked by `CC`. To change the actual compiler (from cray to intel, or gnu) use: `module swap PrgEnv-cray PrgEnv-intel` (or `PrgEnv-gnu`).

3.2.3 Running

Supercomputers use sophisticated scheduling software to force fair sharing of too little resources among too many users. Okeanos uses Slurm https://slurm.schedmd.com/, a standard, open-source scheduler. Each time you want to use some of the nodes, you submit a **job** by declaring its runtime, the number of requested nodes and (usually) the command to run. Slurm queues your job. Once your job is at the top of the queue and there are enough free nodes, your job is allocated to concrete nodes, and then Slurm runs the provided executable.

There are three main usage scenarios. For **long running jobs**, you typically provide a batch script. For **semi-interactive work**, you ask for an allocation and then run your code inside this allocation. Finally, for **interactive work** you submit a job and the terminal blocks until your job is executed.

Batch script:

1. Create a file describing your computation:

   `hello-world.batch`

   ```
   #!/bin/bash -l
   #SBATCH --job-name mim-hello          # this will be shown in the queueing system
   #SBATCH --output "mim-hello-%j.out"   # stdout redirection
   #SBATCH --error "mim-hello-%j.err"    # stderr redirection
   #SBATCH --account "g96-1905"          # the number of our grant
   #SBATCH --nodes 2                     # how many nodes we want
   #SBATCH --reservation=rzadca_12       # our reservation: rzadca_calendardate
   #SBATCH --tasks-per-node 24           # each node is 2 socket, 12 core, so we want 24 tasks on each node
   #SBATCH --time 00:05:00               # if the job runs longer than this, it'll be killed

   srun hello-world-seq.exe              # what command to run
   ```

2. Submit a job using the description:

   ```
   sbatch hello-world.batch
   ```

3. (optional) Nervously check the state of the cluster (`sinfo`) and the queue (`squeue`), maybe filtering your jobs only (`squeue -u your_user_id`).
4. Once the job is completed, the stdout will be in `mim-hello-$jobid.out`.

Allocate & run:

1. Ask for an allocation:

   ```
   salloc --nodes 2 --tasks-per-node 24 --account g96-1905 --time 00:05:00 --reservation=rzadca_12 # use reservation name rzadca_calendardate
   ```

2. Wait until your job starts to execute (it goes through the same scheduler as a batch job).
3. Once `salloc` returns, it launches a shell subprocess with environment variables that describe the current allocation (e.g.: `$SLURM_JOB_NODELIST` contains the host names on which your job executes). But you're still on the head node.
4. To launch a program, use `srun`, try:

   ```
   srun hostname
   ```

   or:

   ```
   srun ./hello-world-seq.exe
   ```

5. Once your job is over, exit the shell subprocess.

Execute a single command (here, `hostname`) without a batch script (but in a batch-like environment: a job executes a command, waits until it completes, then the job ends):

```
srun --nodes 2 --tasks-per-node 24 --account g96-1905 --time 00:05:00  --reservation=rzadca_12
```

4 Communicators

MPI abstracts a group of communicating processes into a **communicator**. A communicator enables processes to address each other by ranks: integers between 0 and $n-1$, where n is the number of processes in a communicator. A communicator also supports collective communication (such as broadcast) - more on this in the next lab.

Once MPI is initialized, it creates a default communicator, `MPI_COMM_WORLD`, grouping all the processes launched with a single `mpiexec` call.

An application can define its own communicators that group a subset of processes (http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node156.htm#Node156).

The following example shows how to query a communicator for the total number of processes and the rank of the current process.

```
int numProcesses, myRank;
MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

- extend `hello-world-seq.c` so that each process prints its rank; and process 0 prints the number of processes.

5 Point to point communication

We start with the default communication mode: blocking receive, potentially blocking send (i.e., it might be a rendez-vous communication, see the slides from the lecture).

```
int MPI_Send(const void* buf,  /* pointer to the message */
            int count, /* number of items in the message */
            MPI_Datatype datatype, /* type of data in the message */
            int dest, /* rank of the destination process */
            int tag, /* app-defined message type */
            MPI_Comm comm /* communicator to use */
           );
```

A message is an array consisting of `count` items of the same type `datatype`. MPI defines standard datatypes, such as `MPI_CHAR`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_BYTE` (full list: http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node48.htm). MPI also allows to define derived types http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node77.htm#Node77.

```
int MPI_Recv(void* buf, /* where the message will be saved */
            int count, /* max number of elements we expect */
            MPI_Datatype datatype, /* type of data in the message */
            int source, /* if not MPI_ANY_SOURCE, receive only from source with the given rank  */
            int tag, /* if not MPI_ANY_TAG, receive only with a certain tag */
            MPI_Comm comm, /* communicator to use */
            MPI_Status *status /* if not MPI_STATUS_IGNORE, write comm info here */
           );
```

You can read communication data from status fields: `status.MPI_SOURCE`, `status.MPI_TAG` and `status.MPI_ERROR`. To read the number of received items use: `MPI_Get_Count(const MPI_Status* status, MPI_Datatype datatype, int* count)`.

- extend `hello-world-seq.c` so that each process of rank $[1, n)$ picks a random integer $r \in [0, 10]$, prints it and then sends a pair $(rank, r)$ to rank 0. Rank 0 prints all the received data.

6 Exercises

6.1 Ring

Organize n processes into a ring: rank 0 sends to rank 1, rank 1 sends to rank 2, rank $n-1$ sends to rank 0. A message contains a single `int64` number. First message is 1; then each process receives the number, multiplies it by its current rank and sends its to the next rank. Rank 0 prints the received message.

6.2 Benchmarking your platform

Simple benchmarking in MPI:

```
double startTime;
double endTime;
double executionTime;

startTime = MPI_Wtime();

// the code to benchmark goes here

endTime = MPI_Wtime();

executionTime = endTime - startTime;
```

Your goal is to compute the throughput (in MB/s) and the round-trip latency (in ms) on Okeanos and on the computers in our labs. For each, you should make N (e.g. 30) experiments, discard 1-2 minimal and maximal values and then average the remaining ones. For throughput, send large messages (millions of bytes); for latency, send short messages (1-10-100 bytes).

After completing this, extend your code to compute throughput in function of the length of the message. You can use our jupyter notebook `draw-bandwidth.ipynb` to display results (the data format is: `experiment_sequence_number message_size communication_time`).

7 Bibliography

- MPI standard: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- Designing and Building Parallel Programs, Ian Foster http://www.mcs.anl.gov/~itf/dbpp/text/node94.html
-

Date: 2024/04/12
Author: Krzysztof Rządca
Created: 2024-04-12 Fri 11:14
Emacs 25.3.50.1 (Org mode 8.2.10)
Validate