Programming of parallel computers
Computer Lab no. 2: Running MPI on a cluster

IT, Uppsala University

March-May, 2019

Getting started

During this lab we use the UPPMAX system Rackham.

Briefly about Rackham

Rackham is a cluster, one of the SNIC resources.
It has 9720 cores, residing in 486 nodes with two 10-core Intel Xeon V4 CPU's each. The majority of the nodes has 128 GB memory. There are four nodes with 1TB memory and 32 nores - with 256 GB memory,
The interconnection network is Infiniband FDR with a maximum bandwidth of 56 Gb/s and a latency of 0.7 microseconds. The topology is described as a fat tree with 14 leaf switches and 6 core switches. Each leaf switch connects 24 servers and reserves 12 ports for uplinks. Information about the network can be found at `http://www.mellanox.com/page/performance_infiniband`.
Some more details about Rackham can be found at `http://uppmax.uu.se/resources/systems/the-rackham-cluster/`.

Accesing Rackham

To access Rackham you need an UPPMAX account, which you should have already gotten by following the instructions from 'How to access UPPMAX resources' 'Course material' at the student portal.
Once you have your account set up, you can login to the Rackham system using `ssh`:

```
$ ssh -Y your-user-name@rackham.uppmax.uu.se
```

In the lab, a number of test programs are to be studied. These are provided on the course webpage at the student portal, under 'Course Material/Labs'. The files are collected in an archive `lab2-code.zip`. Download this archive, and copy it to Rackham's file system using the `scp` (or `sftp`) command,

```
$ scp lab2-code.zip your-user-name@rackham.uppmax.uu.se:lab2-code.zip
```

The files can then be unpacked by `unzip lab2-code.zip`. The commands `ssh` and `scp` are typically available by default on Linux and Mac systems. On Windows, the `PuTTY` program suite can be installed, which has both an SSH client and a SCP utility. (Alternatively, a full Cygwin Unix environment can be set up.)
To edit files located on your UPPMAX account, you can use a command line editor such as `nano`, `emacs` or `vim`. Alternatively, if you have logged in with X11 forwarding (i.e. with the flag `-Y` to `ssh`), you may use a graphical editor like gedit. To start gedit from command line, simply type `gedit`, followed by the name of the file you want to edit. You can also edit copies of the files locally on your computer, and then use `scp` to transfer each time they are updated.

Using Rackham

On UPPMAX, a module system is used to enable various packages. Using the command `module avail` you can see all the available modules. For this course, we are going to need OpenMPI, which in turn requires a recent version of the GCC compiler. Run the following command to enable both of them:

```
$ module load gcc openmpi
```

You can now compile and run programs with `mpicc` and `mpirun` as before.

To use the Alinea DDT debugger, load the module `ddt`.

The batch system SLURM

When executing a program with mpirun on the command line, we are running it on the login node. This is a 'noisy' environment where the activity of other users affect the run time. Moreover, if your program needs a lot of resources, the node will appear slow for other users. To get accurate measurements, we run the code as a dedicated job on the compute nodes using the batch system SLURM. Specifically, a job is created using a batch script which is submitted using the command `sbatch`.
Consider as example the following script `alltoall.sh`, prepared for the code `alltoall`.

```
#!/bin/bash -l

#SBATCH -A g2019005
#SBATCH --reservation g2019005_1
#SBATCH -p core -n 2
#SBATCH -t 5:00

module load gcc openmpi
mpirun -np 2 ./alltoall
```

It first contains a number of comment lines with configurations to `sbatch`, followed by the actual commands to run.
The script is configured for allocation of two cores for a maximum runtime of 5 minutes, and also specifies the course project ID and the reservation for the lab.
The job is submitted by issuing

```
$ sbatch alltoall.sh
```

Don't forget to compile `alltoall` before submitting the job. The output of the job is written to the file `slurm-<jobid>.out`.
To see a list of your jobs in the queue, use thecommand `squeue -u <your-user-name>`. To cancel a running or queued job, use the command `scancel <job-id>`. Note that the '-reservation' line is only valid during the scheduled lab session, and will give an error otherwise. Remove it if you want to run outside of that time.
More at `http://uppmax.uu.se/support/user-guides/slurm-user-guide/`.

Report

Recall: The lab is a part of the examination. To pass you need to either attend the lab and work actively on the tasks (no need to finish all tasks), or write a short informal report summarizing your findings together with your source code for the different tasks, where some programming is required.

Exercises

Exercise 1 (Two-dimensional integral) Use your own code from Assignment 1. Alternatively, consider the code `integral2d.c`, which computes an integral equal to ≈ 2.558041407. The file contains a code for computing this integral, including time measurement.

- Set up a batch script by copying the one from the alltoall example. For the given example `integral2d.c`, setting the project id, the reservation, and the time limit is sufficient; other options are set on the command line. Also, remove any `-np` option to `mpirun` as we want to run on all available cores.
- Now submit some jobs for different numbers of processors, check the output to see the runtime and compute the speedup. For example,

```
$ sbatch  -n 1 integral2d.sh           # single process
$ sbatch  -n 4 integral2d.sh           # four cores on single node
$ sbatch  -n 20 integral2d.sh          #  20 cores (entire node)
$ sbatch  -p node -n 40 integral2d.sh  #  40 cores (two full nodes)
```

Exercise 2 Read through the following programs, which illustrate different types of communications in MPI. Compile and run the codes for varying number of PEs.

1. The test code `alltoall` is an example of using nonblocking communications.
2. The test code `IO_gather` illustrates gathering messages from all processes.
3. The test code `ring`, as the name suggests, simulates a ring architecture and sends a short message from one to the next PE, until the message comes back to the process which initiated the send-loop.

Exercise 3 (Testing the debugger - debugging) Follow the instructions at `https://www.uppmax.uu.se/support/user-guides/allinea-ddt-user-guide/` to get started with Allinea DDT. Then consider the program provided in `codetodebug.c`, which asks you for a number. Each process is supposed to print the sum of its rank and this number, but a bug has sneaked into this code. Compile the code and run it on more than 1 core and study the output. Thereafter, use DDT to step through the code and see if you can find the bug. Full documentation of the debugger can be found at `https://developer.arm.com/docs/101136/latest/ddt`.

Exercise 4 (Testing the debugger - communication patterns) You have two parallel implementations of one and the same algorithm that solves numerically a two-dimensional wave equation. The corresponding codes are `wave-persistent.c`, using persistent communications, and `wave-parallel.c`, which uses other send-receive constructions. Compile the codes and run them one after another via the debugger. Concentrate on the communication part (lines 130 to 182 in `wave-persistent` and lines 225 to 254 in `wave-parallel`). From the `Tools` menu choose `Message queue`. By stepping through the code and updating the system topology window, you are able to monitor the communications performed during the execution. You should be able to see the difference between persistent and `Isend-Irecv` communications. Note: the codes use defined data types, not covered in the course. However, these are not relevant for the exercise.

Exercise 5 (Testing the debugger - monitoring distributed data arrays) Consider the code `arrays.c`. It creates one ordinary (statically allocated) array and two three-dimensional arrays (one that is statically allocated and one that is dynamically allocated). Compile the program without optimizations. Set a breakpoint eg. on line 39 and run the code to the breakpoint. Now you will be able to monitor the distributed arrays and observe the values of the entries in the address space, local to each process: Use the `Multidimensional array viewer` from the `Tools` menu. Study the possibilities it offers - to evaluate array expressions, to slice and visualize the array etc. It is also possible to change array values if you add them to evaluations. See the documentation for details.
You are encouraged to extend the code by performing some arithmetic operations with the arrays and monitor the result. You may also change the size of the arrays by changing `N` in the c file.