02616 Fall 2025: Labs for week 02

Today's menu

1. extend "Hello World" example with hostname, run on multiple nodes
2. extend last example with PID, run on multiple nodes
3. Bandwidth measurement
4. Latency measurement
5. Process mapping/binding and performance

Note:

All the labs are designed and tested on the DTU HPC system.

1. "Hello world" showing the host name

Take the hello_c.c example from last week (the one that both prints rank and size), and also add the printing of the host name, where a rank is executing. This can be useful in a debug situation, where we might want to know, on which machine a rank is executed. There are two ways of doing this:

a. The standard Unix/Linux way: There is a library function in Unix, that returns the host name of the machine a process is running on: `gethostname()` . It has the prototype

```
     #include <unistd.h>
     int gethostname(char *name, size_t len);
```

To use it, you have to declare a string (char array) of a certain size, e.g. 64, and pass this and the size in the function call, like this:

```
    #define HOST_NAME_MAX 64
    char hostname[HOST_NAME_MAX];
    gethostname(hostname, HOST_NAME_MAX);
```

and then you can print the host name in your printf() statement.

b. The MPI way: MPI has an library function that can do the same: `MPI_Get_processor_name()` . It has the prototype

```
    int MPI_Get_processor_name(char* name, int* resultlen);
```

To use it, you have to declare a string (char array) of length `MPI_MAX_PROCESSOR_NAME`, and you also need to pass a reference to an integer variable, that can hold the length of the returned name-string. Like this:

```
    char pname[MPI_MAX_PROCESSOR_NAME];
    int pnamelen;
    MPI_Get_processor_name(pname, &pnamelen);
```

and then you can print the host name in your printf() statement.

Implement either a. or b. above, and then submit a batch job, that starts the program with a total of 8 ranks, and 4 ranks per machine. Check the host names in the output, and compare with the host names you get from LSF (hint: use the `-w` option of `bjobs`)!

2. "Hello world" showing the host name and the process id (PID)

The next extension of our little example is to add also process id (PID) information to the output. There is no intrinsic MPI function to do this, so we need to use the library function of Unix, i.e. `getpid()`. The prototype for this is

```
    #include <unistd.h>
    pid_t getpid(void);
```

In your implementation, declare

```
    pid_t mypid;
    mypid = getpid();
```

and then print this information in the program output. Hint: `pid_t` is an integer type, so you can use the `%d` format in your `printf()` statement.

Implement the above, and then submit a batch job, that starts the program with a total of 8 ranks, and 4 ranks per machine (as before).

Another way to get information about the host name and PID of an MPI rank is implement in `mpirun`. It provides the option `--report-bindings`, which then prints this information at the beginning of the execution (to stderr). This could look like this:

```
[n-62-21-46:749664] Rank 0 bound to ...
[n-62-21-46:749664] Rank 1 bound to ...
[n-62-21-46:749664] Rank 2 bound to ...
[n-62-21-46:749664] Rank 3 bound to ...
[n-62-21-75:752930] Rank 4 bound to ...
[n-62-21-75:752930] Rank 5 bound to ...
[n-62-21-75:752930] Rank 7 bound to ...
[n-62-21-75:752930] Rank 6 bound to ...
```

Add the `--report-bindings` to your job script, and submit again!

3. Testing the bandwidth of different networks

In the next two lab exercises, we want to take a closer look on the difference between a *"fast"* and a *"standard"* network. Our machines in the cluster are equipped with two network interfaces: a high-bandwidth, low-latency Infiniband adapter (called `ib0` , and a standard Ethernet adapter (called `eth0` , which can be either 1 Gbit/s or 10 Gbit/s. The Infiniband network supports up to 56 Gbit/s.

To measure the two metrics of interest, i.e. bandwidth and latency, we will use an often used benchmark, developed and maintained at Ohio State University (OSU): the so called 'OSU MPI Benchmark Suite'. More information on the benchmarks can be found on their webpages: https://mvapich.cse.ohio-state.edu/benchmarks/. In the ZIP file on DTU Learn, we provide you two pre-compiled executables, `osu_bw` and `osu_latency` , as well as some pre-made job scripts, to conduct the tests. The executables have been compiled on the HPC cluster, with the MPI version we are using here.

Your first exercise is to measure the bandwidth of the two different networks in the cluster nodes. In the job scripts, we have set up the necessary parameters to achieve that, e.g. the selection of the *"correct"* network when calling `mpirun` , and we also provide information on the network speed of the two different networks.

Your tasks:

1. Take a look at the job script `bandwidth_test.sub` , and try to understand what it will do.
2. Select one of the two machine types (they have different Ethernet speeds), and submit a job.
3. Take a look at the bandwidth measurements in the job output. What can you observe,
   - when it comes to message size?
   - when it comes to the max. bandwidth of the two networks? How do the max. values measured compare the max. speed of the network?

4. Now change the machine type, re-submit, and repeat step 3. above.

4. Testing the latency of different networks

The second exercise using the OSU benchmarks is to measure the latency of the two different networks. As in the lab above, we provide a job script, `latency_test.sub`, where everything is set up for the measurements.

Note: Unlike the bandwidth, where we have some well-defined maximum speeds, a network can support, the latency can depend on many factors. However, we expect the Infiniband network to have a lower latency than the Ethernet network. Do a search on the internet, which factor you can expect between the latency of the two network types.

Your tasks:

1. Take a look at the job script `latency_test.sub`, and try to understand what it will do.
2. Select one of the two machine types (they have different Ethernet speeds), and submit a job.
3. Take a look at the latency measurements in the job output. What can you observe,
   - when it comes to message size?
   - when it comes to the latency of the two networks? How do the values measured compare your expectations (the findings of your search)?
4. Now change the machine type, re-submit, and repeat step 3. above.

5. Process mapping/binding and performance

In this last exercise for today, you will take a look at the effect of mapping MPI processes (ranks) to pre-defined locations of processing units, which are typically CPU cores. To make it easier for you, we provide a pre-compiled executable for a 2D Poisson problem, implementing a Jacobi solver in MPI. We also provide you with the LSF batch scripts, to investigate the different scenarios below. You can find the files in a ZIP archive on DTU Learn.

Scenario 1: Here we submit an MPI job with 8 ranks, where we ask LSF to place the ranks on a single node, and a single socket. Submit script: `jacobi.sub`. Make yourself familiar with the contents of the script, and submit it. We will take the timings of this run as a reference.

Scenario 2: In this case, we ask LSF for a full node (24 cores in the hpcintro queue), and run again with 8 MPI ranks. Now we use the MPI mapping options to place the ranks distributed over both sockets. Submit script: `jacobi_fullnode.sub`. Check the additional options in this script, and then submit the jobs. What can you observe in the measured timings, when comparing to the scenario above?

Scenario 3: In this, and the next scenario, we will use two nodes, instead of one single machine, which means 4 ranks/node. The first case places the ranks 'compact', i.e. all on the same socket. Submit script: `jacobi_twonodes_compact.sub`. Check both the LSF options, and the MPI mapping flags in the script, to get used to the syntax. You might need them later in this course. Submit the script, and record the timings. How do they compare to the single node timings?

Scenario 4: Now we use again 2 nodes, 4 ranks per node, but then we make use of both sockets in the machine. This should give us a better performance (why?). Submit script: `jacobi_twonodes_spread.sub`. Check the syntax of the script, and submit the job. Compare the timings of this job with the other jobs.

Note: the scenarios above are a typical benchmark situation, where we do not care about 'wasted' resources (cores we reserve, but do not use)!