# Exercise 5 - Multicore/multinode

Michela Venturini

November 26, 2018

**Abstract**

The task of the exercise is to understand commands as hwloc and numactl and use them on Ulysses architecture to test performances (latency and bandwidth) in multicore and multinode cases.

## 1 Ulysses structure

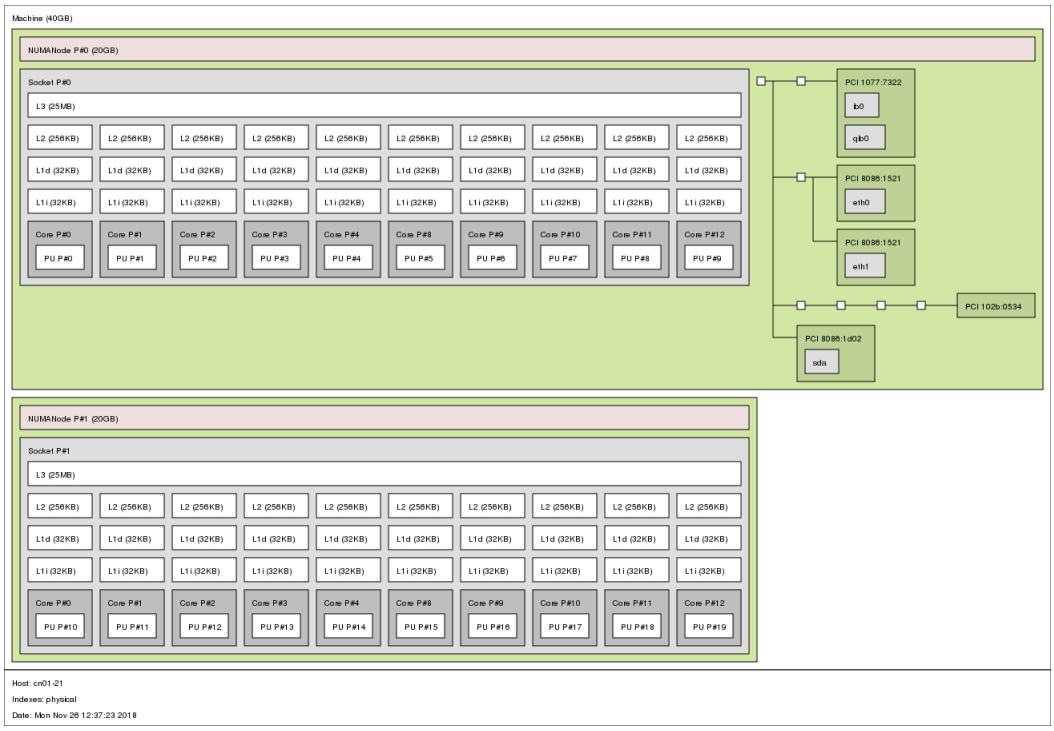


Figure 1: Ulysses node

The figure 1 shows the structure of a Ulysses' node: it is composed by two sockets, each of them provided by 20 GB of memory and 10 cores. The structure is important to understand the exercises belows.

## 2 MPI benchmark

**Latency and Bandwidth** The task of this part of the exercise is to run Intel MPI ping-pong benchmark among two processors within the same node and between two different nodes. MPI ping-pong benchmark open a connection and send and receive messages of increasing size for a number of iteration predefined, by using the functions `MPI_Recv` and `MPI_Send`. In particular is important to measure latency of the program and bandwidh of transmission.

**Latency** to measure latency (time taken to open the transmission) we consider the time taken to ping/pong small messages, in particular the plateau of times constant to sent messages of size from 0 to 16 bytes. We use those messages because they are too samll to affect the total time.

**Bandwidth** to measure bandwidth we consider the largest messages when bandwidth tends to flat.

In order to obtain measures we repeat the commands
`mpirun -np 2 /u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong`
and `mpirun -np 2 /u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 iter 10000 PingPong` three times. We use 2 MPI processes and by `-iter 10000` we define the number of repetitions in sending messages. The table 1 and table 2 show the results:

| Bytes | Run 1 [us] | Run 2 [us] | Run 3 [us] | min [us] |
|---|---|---|---|---|
| 0 | 0.67 | 0.53 | 0.53 | 0.53 |
| 1 | 0.66 | 0.57 | 0.52 | 0.52 |
| 2 | 0.65 | 0.55 | 0.52 | 0.52 |
| 4 | 0.65 | 0.52 | 0.53 | 0.52 |
| 8 | 0.65 | 0.53 | 0.52 | 0.52 |
| 16 | 0.65 | 0.52 | 0.52 | 0.52 |

Table 1: Latency 1000 iterations

As we can see from the tables the latency is independent from the number of repetitions of the transmission because it indicates only the time taken to open the connection.

| Bytes | Run 1 [us] | Run 2 [us] | Run 3 [us] | min [us] |
|---|---|---|---|---|
| 0 | 0.49 | 0.48 | 0.54 | 0.48 |
| 1 | 0.54 | 0.531 | 0.61 | 0.53 |
| 2 | 0.54 | 0.52 | 0.60 | 0.52 |
| 4 | 0.52 | 0.52 | 0.60 | 0.52 |
| 8 | 0.52 | 0.52 | 0.61 | 0.52 |
| 16 | 0.52 | 0.53 | 0.60 | 0.52 |

Table 2: Latency 10000 iterations

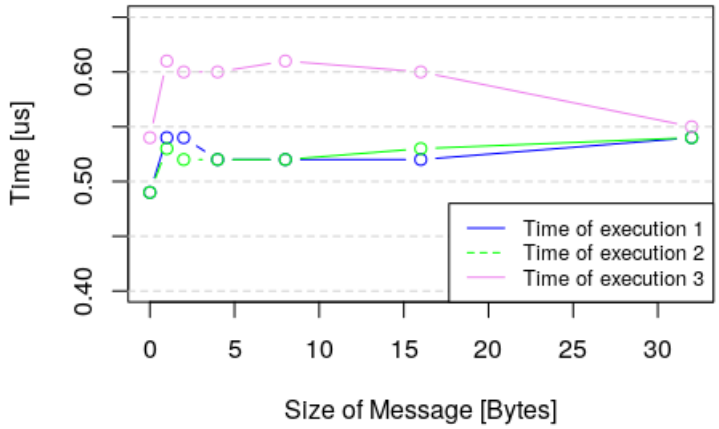


Figure 2: Latency for 10000 ierations

Also the bandwidth, shown in Tables 3 and 4, is not affected from the nuber of repetitions of the ping pong and reach the maximum when the message is large about 4 MB.

**Intranode/Internode performances** The last part of this section consists of measuring the default performances in intranode and internode communications by using Ping-pong MPI benchmark.

**Intranode** First we measure the latency and bandwidth among cores in the same node and in the same socket. To do this we bound the 2 MPI processes among the cores by using `hwloc-bind core:0 core:7`.

| Bytes | Run 1 [MB/s] | Run 2 [MB/s] | Run 3 [MB/s] | max [MB/s] |
|---|---|---|---|---|
| 2000000 | 6053.81 | 6421.04 | 5253.47 | 6421.04 |

Table 3: Bandwidth 1000 iterations

| Bytes | Run 1 [MB/s] | Run 2 [MB/s] | Run 3 [MB/s] | max [MB/s] |
|---|---|---|---|---|
| 2000000 | 5635.80 | 5941.36 | 6250.48 | 6250.48 |

Table 4: Bandwidth 10000 iterations

Then we repeat the measure but between cores in the same node and different sockets by the command `hwloc-bind core:0 core:13`.

**Internodes** To get the measures in this case we ask Ulysses for two entire nodes and we use the commands `--may-by ppn:1` that allows to bind the two MPI processes in two different nodes.

The table 5 shows the comparison between the obtained latency and bandwidth. As we expected, the latency increase when the connection is between sockets in the same node and between cores in dofferent nodes. Moreover this affects also the bandwidth that decreases. The biggest change in latency and bandwidth is when we run the benchmark in the same node but the connection is among different sockets. In this case the latency is higher of 160%. the worst senario is of course when the connection is between differen nodes but with respect to the former case latency increment is about 21%.

| Same Node | Same socket | Latency [usec] | Bandwidth [MB/s] |
|---|---|---|---|
| Y | Y | 0.20 | 11080.29 |
| Y | N | 0.52 | 5949.26 |
| N | N | 0.63 | 5326.95 |

Table 5: Intranode/Internode performances

# 3 Stream

**Stream** Stream is a benchmark that measure sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. This program is designed to work with datasets much larger than the available cache on any given system, so that the results should be more indicative of the performance of very large, vector style applications. In this part of the exercise we use the script stream.c to measures memory transfer

rates in MB/s in different cases. First we buid the program using the given Makefile and then run the script changing the number of threads from one to 10 considering two cases:

1. memory and cpu bounded on the same socket on the same node
   `numactl --cpunodebind 0 --membind 0`
2. memory and cpu bounded on different sockets in the same node
   `numactl --cpunodebind 0 --membind 1`

The figure 3 shows the results. As we can see the bandwidth in both cases increases as the number of threads until reaching a plateau arount 6 threads; moreover, in the case of different socket for memory and cpu the bandwidth remains always less than the first case: we expected this result because the we are in a NUMA configuration.

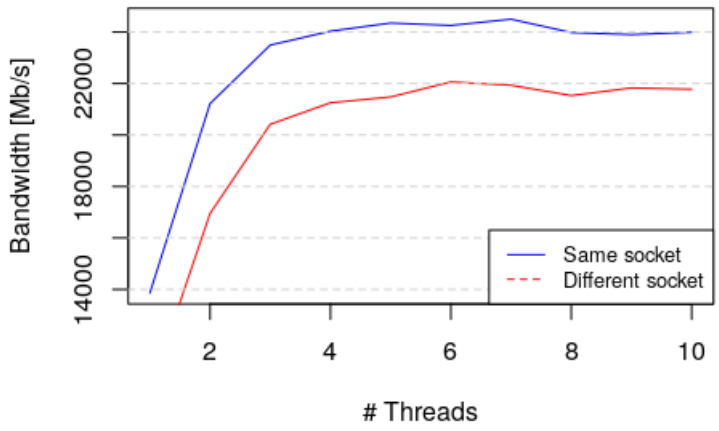


Figure 3: Bandwidth vs # of threads

## 4 Nodeperf

Nodeperf.c is a sample utility that tests the `DGEMM` speed across the cluster and it is a part of Intel MP LINPACK Benchmark for clusters. `DGEMM` is a routine that calculates the product of a double precision matrix:

$$C = \alpha A * B + \beta C \tag{1}$$

**Compilation with Intel compiler** First we compile the `nodeperf.c` using the Intel MPI compile wrapper script for Intel compiler with the optimizations enabled to at level -O2, tuning for the instruction set supported on the build machine by using the -xHost, enabling OpenMP support (-qopenmp), and linking with the MKL library for the optimized version of the DGEMM routine (-mkl): `mpiicc -O2 -xHost -qopenmp -mkl nodeperf.c -o nodeperf`

**Environmental variables** Then we set the number of OpenMP threads equal to the number of processes of Ulysses' node by using `export OMP_NUM_THREADS=20` and we indicate cores as the available hardware with `OMP_PLACES=cores`.

**Performance** The teorethical performance of one Ulysses' node are calculated as:

P = # of cores * # of FP units * CPU frequency = 20*8*2.6 = 448 GFlops

The results we obtain from the execution is P = 451070.733 MFlops that is 440.499 GFlops. This number is 98% of theoretical peack performance of the node. The performance of this program could also overcome the teorethical peack performance thanks to Intel turbo boost that allows cores to increase frequency over limits for a small time.

**Compilation with gcc compiler** In this case we use `mpicc -fopenmp -o3 nodeperf.c -m64 -I$MKLROOT/include -o nodeperf.x -L$MKLROOT/lib/intel64 -Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl`. The performance of the execution were drastically reduced to 26997.223 MFlops: this is due to the fact that nodeperf.c is highly optimized for the Intel and built to take advantages of Intel triks.