Exercise 5 - Multicpu and Multicore

Alessia Paoletti
Matricola SM3500374

1 MPI intranode and internode

Message Passing Interface (MPI) is a standardized and portable message-passing designed to function on a wide variety of parallel computing architectures. MPI is used both for communication in the same node either for communication between different nodes.

Ping Pong is a simple benchmark created by Intel to measuring latency and throughput of a single message sent between two processes. We can use this single program to calculate both latency and bandwidth.

1.1 Latency measure

Latency is the time between initiating a request for a byte or a word in memory until it is retrieves by a processor. If the data are not in the cache it takes longer to obtain them, as the processor will have to communicate with the "external" memory. As the latency decreases the reading operation will be faster.

To measure latency we can use the time it takes to ping/pong the smallest messages. I have done 3 different runs with 1000 iterations and, as Table 1 reports, this time is almost constant for messages size from 0 to 32 bytes. The average latency time is about 0.53 μs.

bytes	**Run 1** [μs]	**Run 2** [μs]	**Run 3** [μs]	**Min** [μs]
0	0.53	0.53	0.54	0.53
1	0.57	0.59	0.55	0.55
2	0.57	0.59	0.55	0.55
4	0.54	0.54	0.53	0.53
8	0.52	0.54	0.53	0.52
16	0.51	0.54	0.52	0.51
32	0.53	0.56	0.54	0.53

Table 1: Estimate latency on 3 different runs

1.2 Bandwidth measure

Bandwidth is the maximum rate of data transfer across a given path. To measure bandwidth we can look at the largest messages when bandwith tend to flat. I have considered 3 different runs and, taking into account the largest messagges, I have observed the value of the bandwidth. As Table 2 shows, when the size of the message is almost 1048576 bytes, the bandwidth tends to flat with the value of almost 7200 MB/s. The results are also presented in Figure 1.

bytes	Run 1 [MB/s]	Run 2 [MB/s]	Run 3 [MB/s]	Max [MB/s]
262144	6802.72	6776.21	6777.99	6802.72
524288	7005.83	6776.21	7035.36	7035.36
1048576	7156.29	7106.73	7094.11	7156.29
2097152	7196.20	7234.52	7207.17	7234.52
4194304	7216.32	7236.55	7294.60	7294.60

Table 2: Bandwidth

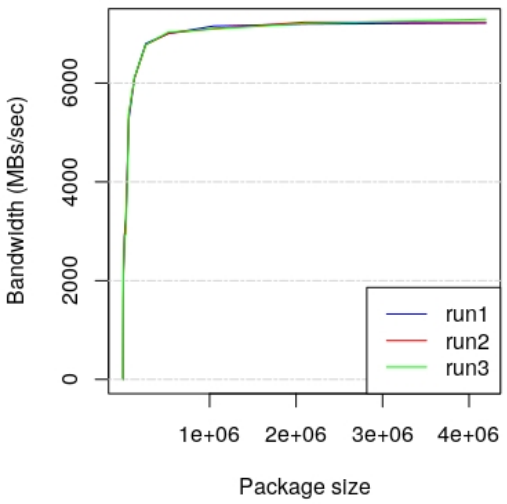

Figure 1: Bandwidth vs Number of threads

1.3 Internode and Intranode Communication Performance

Internode We would like now to evaluate the latency among cores belonging to the same socket and among cores which are on different sockets. Considering the Ulysses' architecture and using one node with 20 processors, we know that the cores with logical id from 0 to 9 belong to the socket 0 while the cores with logical id from 10 to 19 belong to socket 1. To evaluate the latency among cores belonging to the same socket I have done 2 diffent runs, the first among core 0 and core 7 that belong to socket 0 and the second among core 10 and core 17 that belong to socket 1. To evaluate the latency among cores belonging to different sockets I have done 2 different runs, the first among core 0 belonging to socket 0 and core 17 belonging to socket 1, and the second among core 9 belonging to socket 0 and core 19 beloning to socket 1.

As Table 3 reported, and as exepected, the latency among cores that belong to the same socket is less than the latency among cores belong to different socket, due of course to the fact that cores on the same socket can communicate in a faster way.

bytes	Same socket		Different socket	
	Run 1 [μs]	**Run 2** [μs]	**Run 1** [μs]	**Run 2** [μs]
0	0.20	0.19	0.53	0.51
1	0.21	0.20	0.51	0.53
2	0.21	0.21	0.52	0.52
4	0.21	0.20	0.53	0.52
8	0.20	0.20	0.54	0.52
16	0.21	0.20	0.53	0.52
32	0.26	0.22	0.59	0.54

Table 3: Estimate latency among cores belonging to the same socket and among cores on different sockets

Intranode Finally we would like to estimate latency between to cores which are on different nodes. The results are reported in the Table below.

bytes	**Run 1** [μs]	**Run 2** [μs]
0	0.61	0.60
1	0.66	0.70
2	0.65	0.68
4	0.64	0.64
8	0.62	0.63
16	0.65	0.62
32	0.57	0.58

Table 4: Estimate latency among cores in different nodes

Performance comparison In Table 5 are reported the values of estimate latency in all the different cases. As expected the latency between cores in the same socket and belonging to the same node is the lowest while the latency between cores on different nodes is the highest. An observation that can be done is that the greater increase of latency is when we use cores on the same node but beloning to different sockets (from 0.21μs to 0.53μs). On the other hand the increase of latency between cores on different nodes is lower (from 0.53μs to 0.63μs).

Same Node	Same Socket	Estimate latency [μs]
Yes	Yes	0.21
Yes	No	0.53
No	No	0.63

Table 5: Estimate latency among cores in different nodes

2 Stream Benchmark

The STREAM Benchmark is a simple synthetic benchmark program that meausures sustainable bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. We should care about that because computer CPUs are getting faster much more quickly than computer memory systems, so programs will be limited in performance by the memory bandwidth of the system rather than the computational performance of the CPU.

Considering a shared memory, where all processors have access to a pool of shared memory, Non-Uniform Memory Access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory.

Ulysses has a NUMA computer memory design so we can compare the bandwidth of one single core reading from memory associated to the socket against one core reading from memory associated with the other socket. The Figure 2 shows the results obtained doing it for all threads available in one socket. We can observe, as expected, that local access (*close memory)* is faster than no-local access (*distant memory*). In particular we notice that there is big gap when accessing to distant or close memory when we have a single threads, 10042 MB/s vs 13859 MB/s. This gap decreases as the number of threads increases but it never disappear, due to the nature of this computer memory design.

3 Nodeperf

`nodeperf.c` is a small program to test compute nodes performance. This is a simple MPI program that runs a highly optimized version of a double precision general matrix multiply (`DGEMM`) library routine from the MKL. This routine is also the core of the HPL test. As the MKL provides versions of `DGEMM`, it is sure that the test will be optimized for the Intel microprocessors.

I have compiled the `nodeperf` program using the `mpiicc` command, that means use the Intel C Compiler. The successful completation of the commmand produces an executable ready to run. Just before the run, we need to set the `OMP_NUM_THREADS` variable equals

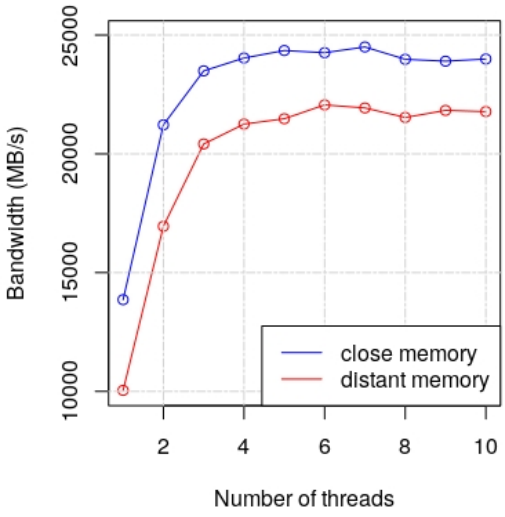

Figure 2: Bandwidth vs Number of threads

to 20, that is the number of physical processor cores in every node of Ulysses. Moreover we set the `OMP_PLACES` variable equals to *cores* in order to distribute threads between the cores in the system, so that two different threads will not run on one physical core.

Running the executable I obtained a performance equals to 449397.756 MegaFlops on the node $cn06-12$. That means a performance equals to almost 449 GigaFlops, that is more than the peak performance of the node, equals to 448 GigaFlops, so the node reported a performance higher than the theoritical peak. This happens because by default Interl Turbo Boost technology is enabled. This allows for processors running at a higher frequency than the nominal one when the CPU power consumption stays within the specification and the cooling system can cool the processor package belowe its critical temperature. So Turbo Boost can help achieve better performance results.

Then I have tried to compile the same nodeperf program using the `mpicc` command, that allows you to choose the compiler, defaulting to `gcc`. Running the executable on the same node $cn06-12$ I obtain a performance equals to 26985.272 MefaFlops, that means almost 27 GigaFlops, almost 6% of theoritical peak performance.

Comparing the two obtained results seems that the second executable does not work. This because the `nodeperf` program is highly optimized in order to use tricks that the Intel compiler provides. For this reason, of course, when we compile with a not Intel compiler (like `gcc`) we obtain bad results.