Exercise 05

Multicore/Multinode

Exercise 5.1: Ping Pong

The Ping Pong test is an elementary point-to-point exchange pattern, in which one MPI process sends a message to another and expects a matching response in return[1]. The MPI ping pong program uses "MPI_Send" and "MPI_Recv" in order to continually bounce messages off of each other until a stopping criteria is reached.

To compile the MPI PingPong program the following command line may be used:

```
[@cn01-08 ~]$
mpirun -np 2 /u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong
```

One output is presented on appendix A.

Latency measure

In order to compute the Latency measure, we ran three different times the last step. Considering 1000 and 10 000 iterations, the following results were obtained (tables 1 and 2):

Table 1: Latency Measure - PingPong output - 1 000 iterations

#bytes	#repetitions	Run 1	Run 2	Run 3	Mean	Std
		(µs)	(µs)	(µs)	(µs)	(µs)
0	1000	0,21	0,21	0,22	0,21	0,01
1	1000	0,21	0,22	0,24	0,22	0,02
2	1000	0,21	0,24	0,24	0,23	0,02
4	1000	0,22	0,21	0,24	0,22	0,02
8	1000	0,21	0,23	0,23	0,22	0,01
16	1000	0,2	0,24	0,25	0,23	0,03
32	1000	0,31	0,29	0,26	0,29	0,03
64	1000	0,29	0,29	0,31	0,30	0,01
128	1000	0,3	0,3	0,32	0,31	0,01
256	1000	0,32	0,32	0,33	0,32	0,01
512	1000	0,39	0,4	0,41	0,40	0,01
1024	1000	0,45	0,44	0,46	0,45	0,01
2048	1000	0,61	0,6	0,62	0,61	0,01
4096	1000	1,03	1	1,02	1,02	0,02
8192	1000	1,62	1,6	1,62	1,61	0,01
16384	1000	2,85	2,82	2,88	2,85	0,03
32768	1000	5,08	5,11	5,07	5,09	0,02
65536	640	8,07	8,02	7,97	8,02	0,05
131072	320	14,43	14,14	14,03	14,20	0,21
262144	160	26,86	26,86	26,67	26,80	0,11
524288	80	49,09	49,09	49,06	49,08	0,02
1048576	40	93,44	93,38	93,37	93,40	0,04
2097152	20	182,67	183,93	183,87	183,49	0,71
4194304	10	654,24	615,2	669,9	646,45	28,17

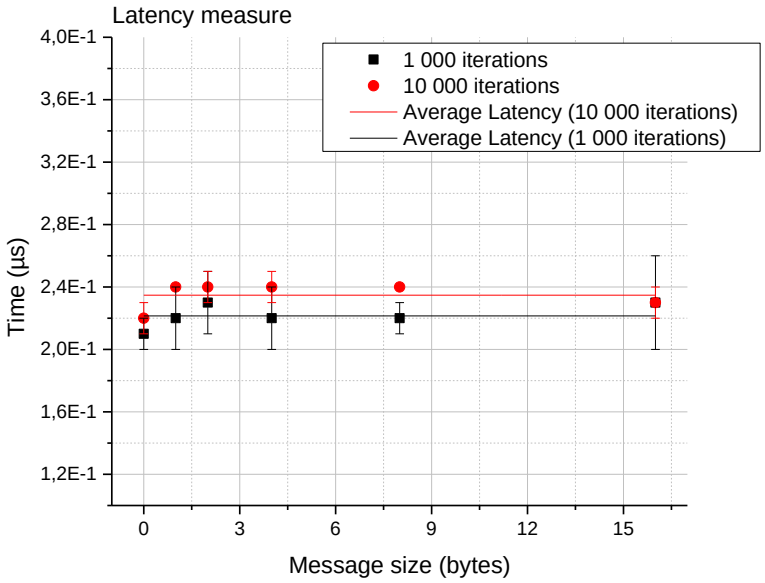
Table 2: Latency Measure - PingPong output - 10 000 iterations

#bytes	#repetitions	Run 1	Run 2	Run 3	Mean	Std
		(µs)	(µs)	(µs)	(µs)	(µs)
0	10000	0,21	0,22	0,23	0,22	0,01
1	10000	0,24	0,24	0,24	0,24	0,00
2	10000	0,23	0,24	0,24	0,24	0,01
4	10000	0,23	0,24	0,24	0,24	0,01
8	10000	0,24	0,24	0,24	0,24	0,00
16	10000	0,23	0,24	0,23	0,23	0,01
32	10000	0,29	0,3	0,3	0,30	0,01
64	10000	0,31	0,32	0,33	0,32	0,01
128	10000	0,32	0,33	0,34	0,33	0,01
256	10000	0,34	0,35	0,36	0,35	0,01
512	10000	0,4	0,4	0,42	0,41	0,01
1024	10000	0,46	0,47	0,48	0,47	0,01
2048	10000	0,63	0,62	0,65	0,63	0,02
4096	10000	1,02	1,03	1,06	1,04	0,02
8192	5120	1,61	1,62	1,67	1,63	0,03
16384	2560	2,85	2,86	2,93	2,88	0,04
32768	1280	5,1	5,09	5,26	5,15	0,10
65536	640	7,98	7,94	8,18	8,03	0,13
131072	320	14,2	14,2	14,24	14,21	0,02
262144	160	26,76	26,97	27,28	27,00	0,26
524288	80	49,51	49,76	49,09	49,45	0,34
1048576	40	93,35	94,11	94,4	93,95	0,54
2097152	20	184,48	184,8	183,95	184,41	0,43
4194304	10	666,36	662,9	471,95	600,40	111,26

[1] Optimizing HPC Applications with Intel Cluster Tools, Paperback – October 15, 2014 by Alexander Supalov, Andrey Semin, Michael Klemm, ISBN-13: 978-1430264965 ISBN-10: 1430264969 Edition: 1st.

Exercise 05

Analyzing tables 1 and 2, one can notice that the time varies in a non linear fashion with respect to the message size considered. To measure the latency it is necessary to identify the time it takes to ping pong the smallest messages. Considering that time is approximately constant in the interval from 0 to 16 bytes, it was possible to plot the graph 1 and compute the average latency.



Graph 1: Latency Measure (µs) - 1 000 and 10 000 iterations

One can notice that the latency for the case with 1 000 and 10 000 iterations are constant, regardless of the number of iterations.

Result table 1: Average Latency for Ping Pong 1 000 and 10 000 iterations, with the standard deviation of the mean.

iterations	Average Latency (µs)	Std (µs)
1 000	0,22	0,01
10 000	0,23	0,01

.

Exercise 05

Bandwidth measure

In order to measure the bandwidth, in contrast to the previous procedure to measure the latency, this time we need to consider the largest messages, when the bandwidth tends to flat. We ran the case considering 1000 and 10 000 iterations, as can be seen in tables 3 and 4:

Table 3: Bandwidth Measure - PingPong output - 1 000 iterations

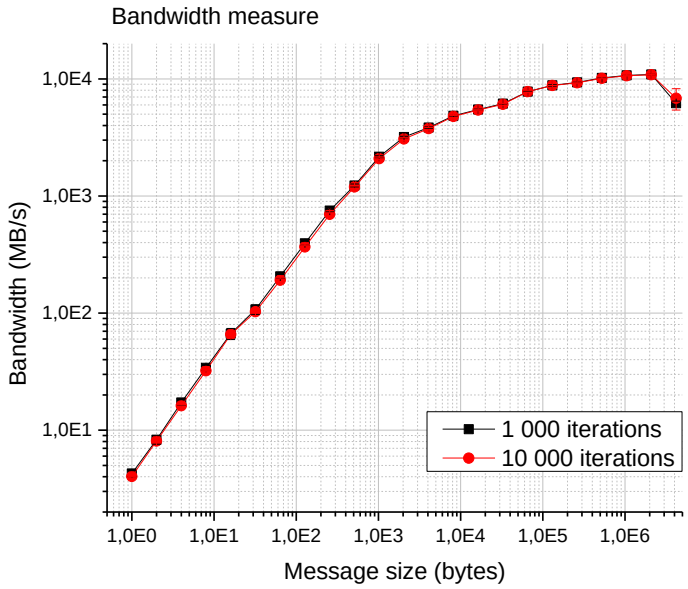
#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		MB/s	MB/s	MB/s	MB/s	MB/s
0	1000	0,00	0,00	0,00	0,00	0,00
1	1000	4,49	4,28	4,00	4,26	0,25
2	1000	9,07	7,79	7,98	8,28	0,69
4	1000	17,71	17,79	16,13	17,21	0,94
8	1000	35,98	32,75	33,39	34,04	1,71
16	1000	74,81	64,26	60,92	66,66	7,25
32	1000	98,77	105,22	118,24	107,41	9,92
64	1000	207,54	212,71	197,23	205,83	7,88
128	1000	401,57	402,83	376,75	393,72	14,71
256	1000	758,24	765,32	729,86	751,14	18,77
512	1000	1266,93	1230,03	1190,70	1229,22	38,12
1024	1000	2185,12	2196,84	2113,52	2165,16	45,10
2048	1000	3201,88	3228,37	3132,70	3187,65	49,40
4096	1000	3796,11	3898,17	3824,02	3839,43	52,75
8192	1000	4826,99	4879,82	4822,72	4843,18	31,81
16384	1000	5477,77	5532,80	5426,29	5478,95	53,26
32768	1000	6145,54	6113,01	6163,16	6140,57	25,44
65536	640	7747,32	7794,29	7844,77	7795,46	48,74
131072	320	8662,56	8842,68	8908,65	8804,63	127,38
262144	160	9306,46	9308,78	9373,27	9329,50	37,92
524288	80	10185,91	10185,91	10192,41	10188,08	3,75
1048576	40	10702,14	10709,32	10709,66	10707,04	4,25
2097152	20	10948,68	10873,82	10876,99	10899,83	42,34
4194304	10	6113,92	6501,91	5971,07	6195,63	274,69

Table 4: Bandwidth Measure - PingPong output - 10 000 iterations

#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		MB/s	MB/s	MB/s	MB/s	MB/s
0	10000	0,00	0,00	0,00	0,00	0,00
1	10000	4,05	4,01	4,03	4,03	0,02
2	10000	8,17	8,02	8,06	8,08	0,08
4	10000	16,39	16,01	16,10	16,17	0,20
8	10000	32,37	32,00	32,19	32,19	0,19
16	10000	66,43	64,17	66,47	65,69	1,32
32	10000	105,09	100,40	103,17	102,89	2,36
64	10000	197,33	187,89	187,05	190,76	5,71
128	10000	377,11	364,72	359,18	367,00	9,18
256	10000	717,64	695,68	678,08	697,13	19,82
512	10000	1218,87	1206,98	1162,28	1196,04	29,84
1024	10000	2116,74	2079,98	2051,13	2082,62	32,88
2048	10000	3076,52	3145,87	2989,18	3070,52	78,52
4096	10000	3813,60	3792,64	3676,30	3760,85	73,97
8192	5120	4842,33	4812,60	4675,08	4776,67	89,23
16384	2560	5474,16	5462,22	5336,52	5424,30	76,25
32768	1280	6123,63	6134,49	5937,79	6065,30	110,56
65536	640	7835,61	7876,44	7636,59	7782,88	128,33
131072	320	8800,01	8803,70	8776,76	8793,49	14,61
262144	160	9342,73	9268,92	9164,87	9258,84	89,36
524288	80	10099,76	10047,74	10185,91	10111,14	69,78
1048576	40	10712,39	10625,55	10593,35	10643,76	61,57
2097152	20	10841,50	10822,61	10872,41	10845,51	25,14
4194304	10	6002,80	6034,10	8475,48	6837,46	1418,65

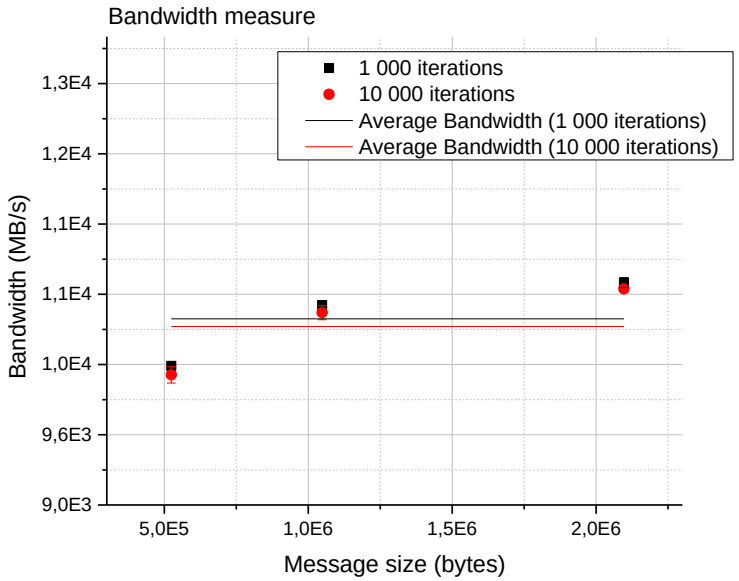
One can notice that the bandwidth grows in a non linear fashion with the message size in bytes. This behaviour can be verified in graph 2. It could be noticed that, as expected, there is an approximately flat region at the larger message region.

Exercise 05



Graph 2: Bandwidth Measure (MB/s) - 1 000 and 10 000 iterations

The last point present a similar behaviour in both curves of graph 2: just after the flat region is noticed a sharp decline in bandwidth. Considering the points in the flat region of the graph 2, it could be computed the average bandwidth, as shown in graph 3.



Graph 3: Average Bandwidth (MB/s) - 1 000 and 10 000iterations

Considering the graph 3, where the average bandwidth is computed for 1 000 and 10 000 iterations one can notice that the behaviour is approximately constant, being around 10 Gb/s, regardless of the number of iterations.

Exercise 05

Result table 2: Average Bandwidth for Ping Pong 1 000 and 10 000 iterations, with the standard deviation of the mean.

iterations	Average Bandwidth (MB/s)	Std (MB/s)
1 000	10598	368
10 000	10533	379

Until this point of the analysis of the latency and bandwidth it was not specified where the benchmark should run. By default, the Intel MPI library will try to select the fastest possible communication path for any particular runtime configuration. Here, the most probable is that the shared memory channel is being used[2].
With the command "hwloc-bind" it is possible to select where to run the program and compare the metrics more carefully. Firstly rerunning the Ping Pong program for the cores 0 and 7 (intrasocket) and then for the cores 0 and 13 (intersocket), tables 5 and 6 were obtained.

```
[@cn01-08 ~]$ mpirun -np 2 hwloc-bind core:0 core:7
/u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong
[@cn01-08 ~]$ mpirun -np 2 hwloc-bind core:0 core:13
/u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong
```

Table 5: Latency Measure - PingPong output - 1 000 iterations
Cores: 0 and 7 - Intrasocket

#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		(μs)	(μs)	(μs)	(μs)	(μs)
0	1000	0,23	0,20	0,19	0,21	0,02
1	1000	0,20	0,21	0,19	0,20	0,01
2	1000	0,20	0,18	0,20	0,19	0,01
4	1000	0,22	0,20	0,24	0,22	0,02
8	1000	0,20	0,21	0,20	0,20	0,01
16	1000	0,21	0,20	0,19	0,20	0,01
32	1000	0,25	0,22	0,24	0,24	0,02
64	1000	0,32	0,28	0,27	0,29	0,03
128	1000	0,31	0,30	0,27	0,29	0,02
256	1000	0,31	0,30	0,29	0,30	0,01
512	1000	0,36	0,37	0,38	0,37	0,01
1024	1000	0,42	0,41	0,40	0,41	0,01
2048	1000	0,55	0,55	0,54	0,55	0,01
4096	1000	0,92	0,90	0,91	0,91	0,01
8192	1000	1,45	1,44	1,45	1,45	0,01
16384	1000	2,56	2,57	2,57	2,57	0,01
32768	1000	4,45	4,48	4,48	4,47	0,02
65536	640	7,83	7,61	7,74	7,73	0,11
131072	320	14,37	14,39	14,15	14,30	0,13
262144	160	27,33	27,16	27,15	27,21	0,10
524288	80	49,89	50,03	49,91	49,94	0,08
1048576	40	95,24	95,91	95,40	95,52	0,35
2097152	20	181,07	182,87	181,45	181,80	0,95
4194304	10	353,60	353,50	350,45	352,52	1,79

Table 6: Latency Measure - PingPong output - 1 000 iterations
Cores: 0 and 13 - Intersocket

#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		(μs)	(μs)	(μs)	(μs)	(μs)
0	1000	0,55	0,53	0,50	0,53	0,03
1	1000	0,62	0,62	0,52	0,59	0,06
2	1000	0,61	0,60	0,52	0,58	0,05
4	1000	0,59	0,59	0,53	0,57	0,03
8	1000	0,59	0,62	0,52	0,58	0,05
16	1000	0,59	0,62	0,56	0,59	0,03
32	1000	0,57	0,56	0,54	0,56	0,02
64	1000	0,60	0,61	0,57	0,59	0,02
128	1000	0,61	0,66	0,59	0,62	0,04
256	1000	0,63	0,64	0,62	0,63	0,01
512	1000	0,83	0,83	0,81	0,82	0,01
1024	1000	0,98	0,98	0,95	0,97	0,02
2048	1000	1,26	1,32	1,27	1,28	0,03
4096	1000	1,65	1,69	1,63	1,66	0,03
8192	1000	3,15	3,12	3,20	3,16	0,04
16384	1000	5,48	5,49	5,45	5,47	0,02
32768	1000	11,12	11,15	10,97	11,08	0,10
65536	640	12,75	12,92	12,36	12,68	0,29
131072	320	22,17	22,75	23,04	22,65	0,44
262144	160	40,00	42,10	41,34	41,15	1,06
524288	80	80,09	86,71	85,88	84,23	3,61
1048576	40	162,15	177,41	175,46	171,67	8,30
2097152	20	326,93	357,98	356,72	347,21	17,57
4194304	10	651,19	723,45	717,45	697,36	40,10

[2] Optimizing HPC Applications with Intel Cluster Tools, Paperback – October 15, 2014 by Alexander Supalov, Andrey Semin, Michael Klemm, ISBN-13: 978-1430264965 ISBN-10: 1430264969 Edition: 1st.

Exercise 05

Analyzing the latency of both cases, as it was performed before, now it is possible to notice a considerable difference: the latency for the cores 0 and 13 (intersocket) is approximately three times higher than for the cores 0 and 7 (intrasocket).

As regards to the bandwidth, tables 7 and 8 were obtained:

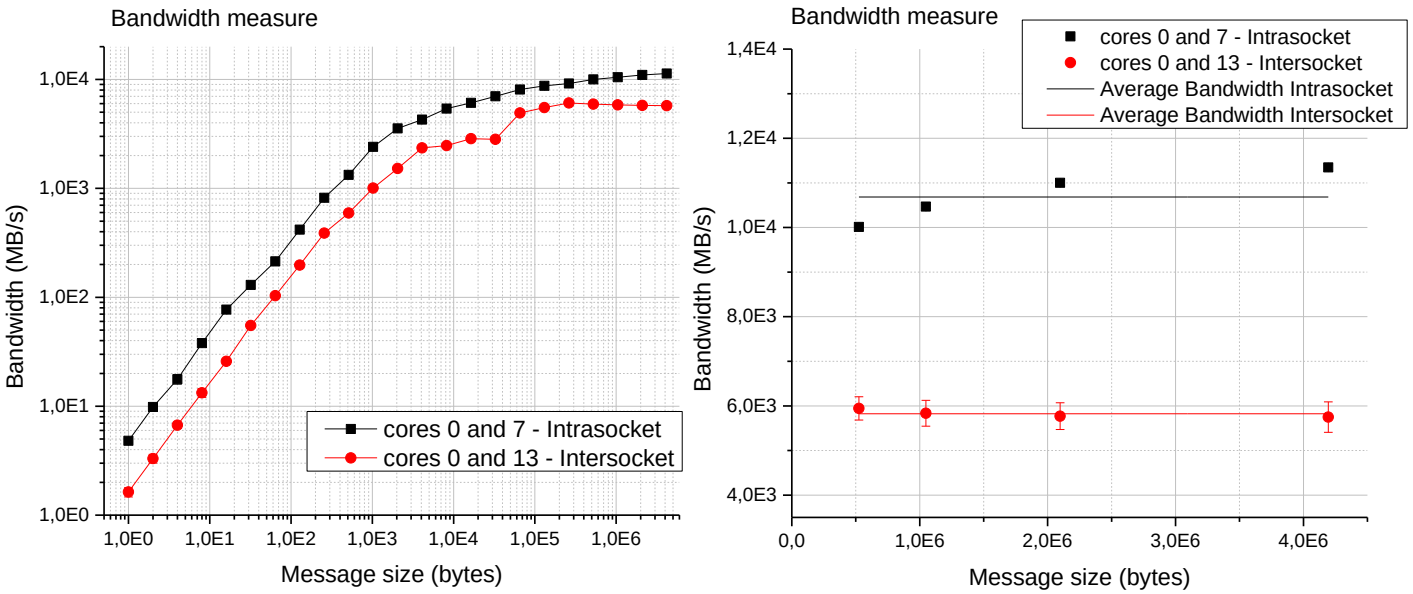
Table 7: Bandwidth Measure - PingPong output - 1 000 iterations
Cores: 0 and 7 - Intrasocket

#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		MB/s	MB/s	MB/s	MB/s	MB/s
0	1000	0,00	0,00	0,00	0,00	0,00
1	1000	4,82	4,60	5,02	4,81	0,21
2	1000	9,59	10,34	9,58	9,84	0,44
4	1000	17,70	19,46	15,96	17,71	1,75
8	1000	38,55	36,93	38,25	37,91	0,86
16	1000	74,25	78,05	79,06	77,12	2,54
32	1000	122,08	138,75	129,29	130,04	8,36
64	1000	193,50	218,43	229,80	213,91	18,57
128	1000	392,49	412,40	447,94	417,61	28,09
256	1000	784,98	820,84	852,27	819,36	33,67
512	1000	1368,07	1316,20	1301,97	1328,75	34,79
1024	1000	2352,67	2402,35	2465,98	2407,00	56,80
2048	1000	3528,75	3521,93	3610,40	3553,69	49,23
4096	1000	4241,26	4323,53	4278,37	4281,05	41,20
8192	1000	5374,89	5434,61	5388,14	5399,21	31,36
16384	1000	6112,86	6077,43	6088,16	6092,82	18,17
32768	1000	7029,50	6982,50	6970,80	6994,27	31,07
65536	640	7977,75	8215,27	8076,65	8089,89	119,31
131072	320	8696,46	8685,21	8835,69	8739,12	83,82
262144	160	9147,13	9205,86	9208,13	9187,04	34,58
524288	80	10021,33	9993,87	10018,64	10011,28	15,14
1048576	40	10499,87	10426,14	10482,16	10469,39	38,49
2097152	20	11045,27	10936,55	11022,41	11001,41	57,32
4194304	10	11312,26	11315,31	11413,85	11347,14	57,79

Table 8: Bandwidth Measure - PingPong output - 1 000 iterations
Cores: 0 and 13 - Intersocket

#bytes	#rep	Run 1	Run 2	Run 3	Mean	Std
		MB/s	MB/s	MB/s	MB/s	MB/s
0	1000	0,00	0,00	0,00	0,00	0,00
1	1000	1,53	1,55	1,82	1,63	0,16
2	1000	3,11	3,16	3,65	3,31	0,30
4	1000	6,44	6,43	7,23	6,70	0,46
8	1000	12,86	12,29	14,67	13,27	1,24
16	1000	25,75	24,55	27,20	25,83	1,33
32	1000	53,68	54,54	56,93	55,05	1,68
64	1000	102,15	100,63	107,56	103,45	3,64
128	1000	200,59	185,54	207,08	197,74	11,05
256	1000	389,72	382,66	393,77	388,72	5,62
512	1000	588,93	589,01	605,02	594,32	9,27
1024	1000	993,93	996,96	1025,80	1005,56	17,59
2048	1000	1553,72	1482,45	1532,50	1522,89	36,59
4096	1000	2365,92	2312,16	2392,00	2356,69	40,71
8192	1000	2479,79	2501,18	2438,00	2472,99	32,13
16384	1000	2853,61	2848,65	2866,21	2856,16	9,05
32768	1000	2810,26	2802,21	2848,96	2820,48	24,99
65536	640	4900,75	4837,02	5056,88	4931,55	113,12
131072	320	5638,55	5493,79	5424,52	5518,95	109,21
262144	160	6249,43	5937,79	6047,81	6078,34	158,05
524288	80	6242,69	5766,16	5822,39	5943,75	260,41
1048576	40	6167,07	5636,56	5699,27	5834,30	289,89
2097152	20	6117,60	5586,91	5606,61	5770,37	300,87
4194304	10	6142,58	5529,10	5575,31	5749,00	341,64

Exercise 05



Graphs 4 and 5: Bandwidth Measure (MB/s) - 1 000 iterations - comparison Intrasocket/Intersocket

At this point, we begin our investigation about the latency and bandwidth when considering the internode configuration. With this aim, the following command lines were used:

```
[@cn01-08 ~]$ mpirun -np 2 -ppn 1 -hosts cn01-08,cn01-12
/u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong
```

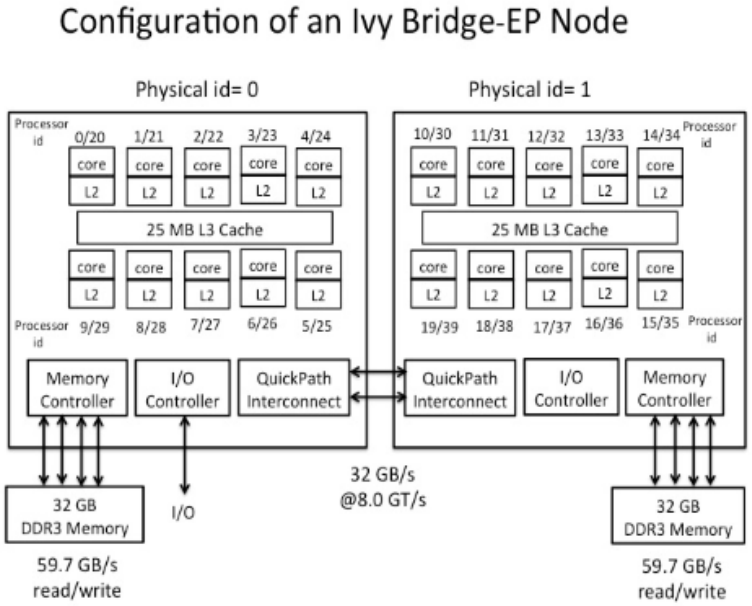


Figure 1: Configuration of an Ivy Bridge-EP Node[3]

[3] www.nas.nasa.gov/hecc/support/kb/ivy-bridge-processors_445.html

Exercise 05

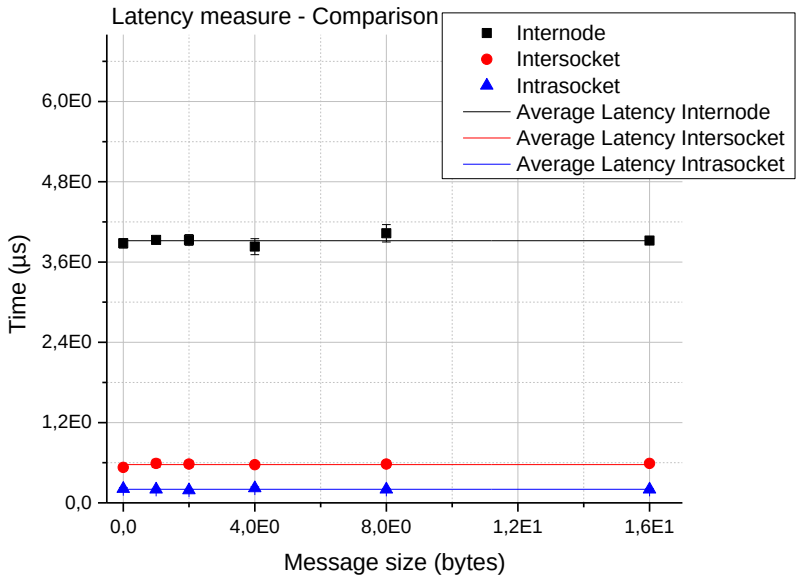
As can be seen in figure 1, the cores 0 and 7 share the L3 cache, what explains the results of lower latency and higher bandwidth for this case. On the other hand, the cores 0 and 13 can only communicate through a Quick Path Interconnect (QPI), what explains a higher latency and lower bandwidth.
The location of cores can be confirmed through the command line:

```
[@cn01-08 ~]$ numactl --hardware
```

That, in this case, produces the output:

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 20451 MB
node 0 free: 19515 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 20480 MB
node 1 free: 19626 MB
node distances:
node   0   1
  0:  10  11
  1:  11  10
```

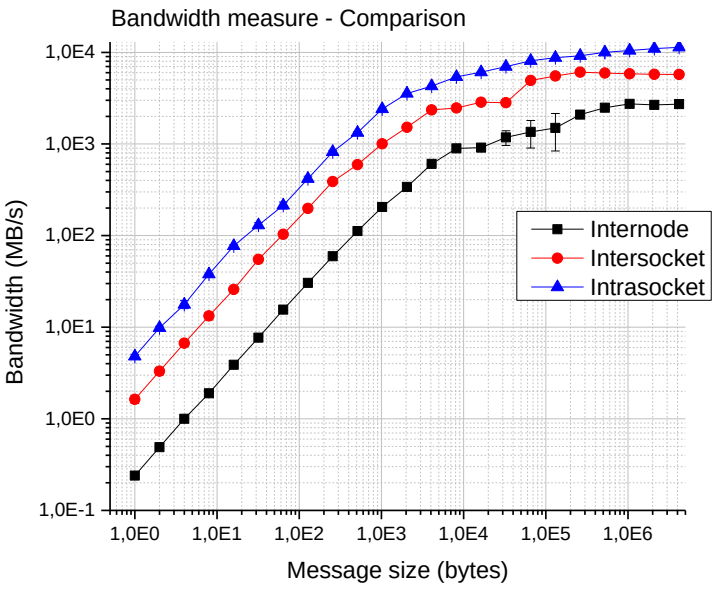
The latency of the configurations analyzed (Intrasocket, Intersocket, and Internode) can be compared by plotting a graph with the results, as can be seen in graph 6.

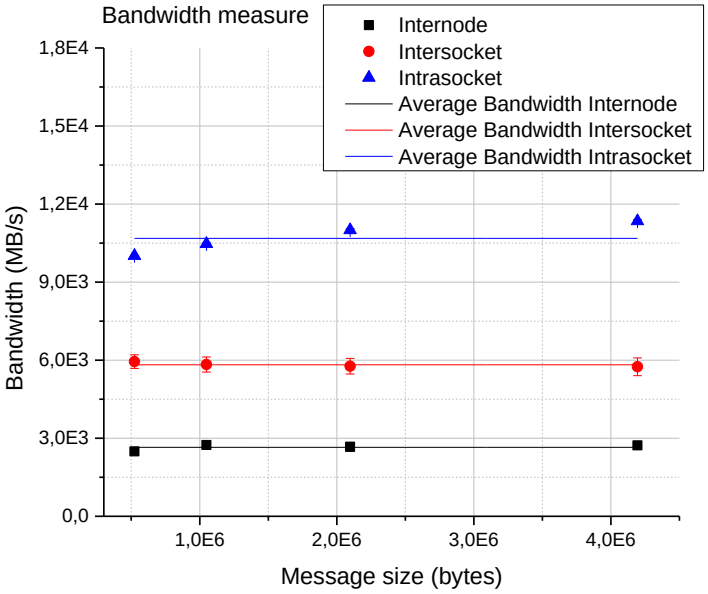


Graph 6: Average Latency (µs) - 1 000 iterations - comparison Intrasocket/Intersocket/Internode

The same is done for the bandwidth in graphs 7 and 8.

Exercise 05





Graphs 7 and 8: Bandwidth measure (MB/s) - 1 000 iterations - comparison Intrasocket/Intersocket/Internode

Comparing the results obtained for the intrasocket, intersocket and internode configurations, one can conclude that experimental results are in agreement with the theory. It is possible to notice that, as expected, the intrasocket configuration presents lower latency, followed by intersocket and internode configurations. As regards to the bandwidth measure, the intrasocket configuration presents higher bandwidth, followed by intersocket and internode configurations. All those results are strongly related to the physical distance among hardware components and memory sharing in the intranode configurations.

Result table 3: Average Latency for Ping Pong - 1 000 iterations - comparison Intrasocket/Intersocket/Internode, with the standard deviation of the mean.

	Average Latency (µs)	Std (µs)
Intrasocket	0,20	0,01
Intersocket	0,57	0,02
Internode	3,92	0,07

Result table 4: Average Bandwidth for Ping Pong - 1 000 iterations - comparison Intrasocket/Intersocket/Internode, with the standard deviation of the mean.

	Average Bandwidth (MB/s)	Std (MB/s)
Intrasocket	10707	588
Intersocket	5824	87
Internode	2711	36

Exercise 05

Exercise 5.2 : Stream

The memory bandwidth data may be obtained by use of the STREAM benchmark code. STREAM is a synthetic benchmark, written in standard Fortran 77, which measures the performance of four long vector operations as follows:

```
-------------------------------------------
per iteration:
name kernel bytes FLOPS
-------------------------------------------
COPY: a(i) = b(i) 16 0
SCALE: a(i) = q*b(i) 16 1
SUM: a(i) = b(i) + c(i) 24 1
TRIAD: a(i) = b(i) + q*c(i) 24 2
-------------------------------------------
```

These operations are intended to represent the elemental operations on which long-vector codes are based, and are specifically intended to eliminate the possibility of data re-use (either in registers or in cache)[4]. The TRIAD operation was considered in the following analysis.
Remembering the meaning of the commands:

numactl
Runs processes with a specific NUMA scheduling or memory placement policy.
--membind=nodes
Only allocate memory from nodes.
--cpunodebind=nodes
Only execute command on the CPUs of nodes.

In order to compile the Stream program, the following command line may be used:

```
[@cn01-30 ex_05_02]$ export OMP_NUM_THREADS=1
[@cn01-30 ex_05_02]$ numactl --cpunodebind=0 --membind=0 ./stream_omp.x
```

One output is presented in appendix B. With “number of threads” varying from 1 to 10. In this context, the configuration “numactl --cpunodebind=0 --membind=0” is related to a closer physical distance from the place where the memory is allocated and where the commands are executed, on the other hand, the configuration “numactl --cpunodebind=0 --membind=1” is related to a more distant physical distance (see figure 1 for details). The results from this experiment is shown in tables 9 and 10.

Table 9: TRIAD (--cpunodebind=0--membind=0)

#threads	Run 1	Run 2	Run 3	Mean	Std
	MB/s	MB/s	MB/s	MB/s	MB/s
1	13778,4	13779,0	13743,5	13767,0	20,3
2	20170,2	19914,2	19709,7	19931,4	230,7
3	21467,2	21623,8	21894,8	21661,9	216,3
4	21731,0	22317,2	22568,1	22205,4	429,6
5	22310,8	21845,7	21998,2	22051,6	237,1
6	22088,3	22266,0	22443,0	22265,8	177,4
7	21970,7	21986,9	21922,6	21960,1	33,4
8	22140,5	21794,4	21954,6	21963,2	173,2
9	22031,0	21609,2	22062,4	21900,9	253,1
10	21572,6	22356,8	22116,0	22015,1	328,0

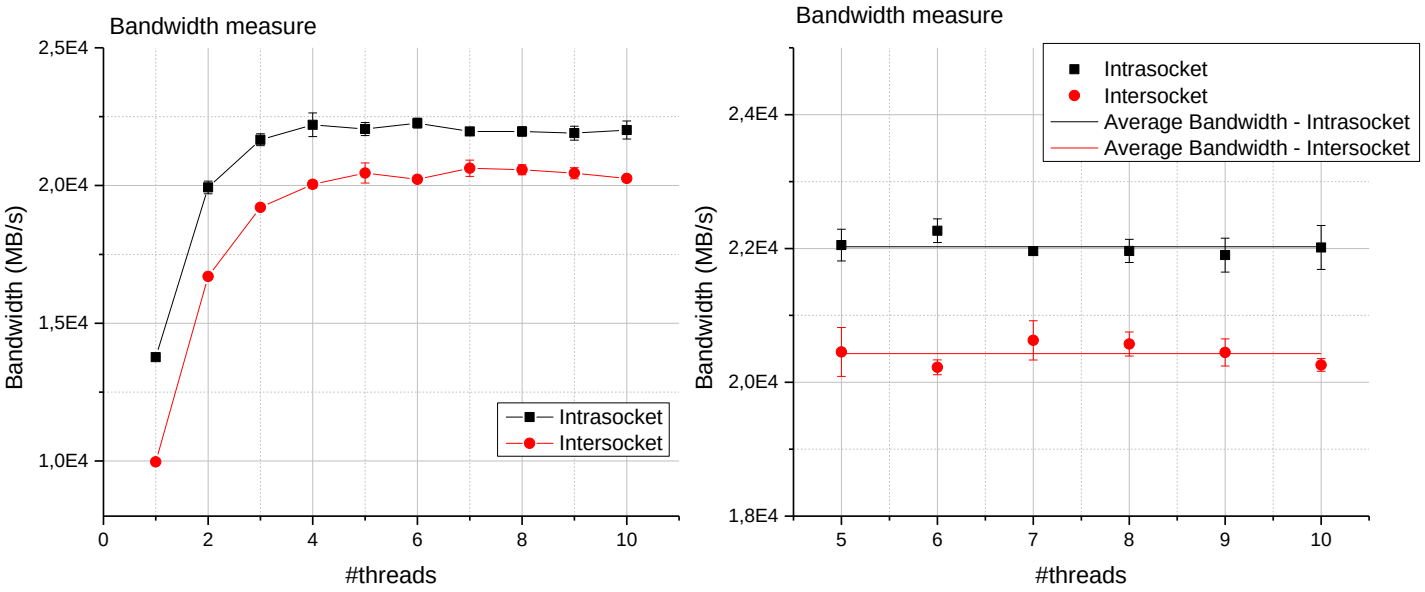
Table 10: TRIAD (--cpunodebind=0--membind=1)

#threads	Run 1	Run 2	Run 3	Mean	Std
	MB/s	MB/s	MB/s	MB/s	MB/s
1	9966,2	9975,8	9959,6	9967,2	8,1
2	16714,1	16764,6	16624,4	16701,0	71,0
3	19307,4	19216,6	19104,5	19209,5	101,6
4	20141,8	19892,9	20099,0	20044,6	133,1
5	20038,0	20723,4	20598,4	20453,3	365,0
6	20340,7	20120,5	20209,3	20223,5	110,8
7	20861,2	20297,9	20721,8	20627,0	293,4
8	20689,6	20663,2	20365,5	20572,8	180,0
9	20387,3	20280,4	20671,8	20446,5	202,3
10	20257,0	20356,2	20165,5	20259,6	95,4

[4] McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

Exercise 05

From tables 9 and 10 it was possible to analyze the dependency between the memory bandwidth and the number of threads for the Stream benchmark (TRIAD) and estimate the average memory bandwidth for each configuration considered (see graphs 9 and 10).



Graphs 9 and 10: Memory Bandwidth Measure (MB/s) - Stream (TRIAD) - Intrasocket/Intersocket

The average memory bandwidth was estimated in approximately 22 GB/s for the intrasocket configuration and 20 GB/s for the intersocket configuration:

Result table 5: Average Memory Bandwidth for Stream (TRIAD) - comparison Intrasocket/Intersocket, with the standard deviation of the mean.

	Average Memory Bandwidth (MB/s)	Std (MB/s)
Intrasocket	21971	25
Intersocket	20324	60

Analyzing the results it is possible to confirm the previous results that a closer physical distance among hardware components is linked to a higher bandwidth.

Exercise 05

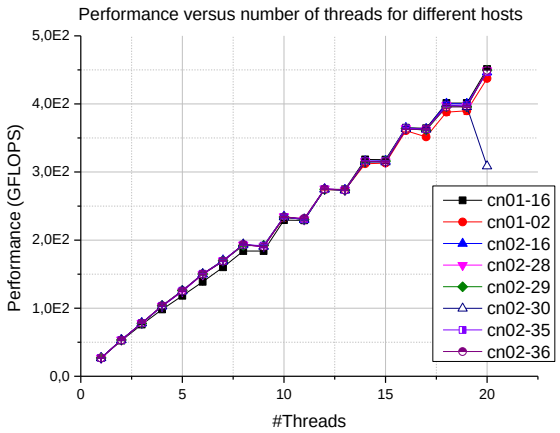
Exercise 5.3: Nodeperf

The code nodeperf.c is a sample utility that tests the performance for matrix-matrix multiplication[5] on all the nodes across the cluster. This simple MPI program runs a highly optimized version of a Double Precision General Matrix Multiply (DGEMM) library routine from the MKL. The "nodeperf" program can go through all available nodes, one at a time, and report the performance (in MB/s) of DGEMM followed by some host identifier. Therefore, the higher the penultimate number then, the faster that node was performing.[6]

In order to compile and execute the code, the following command lines were used:

```
[@cn01-10 ex_05_03]$ mpiicc -O2 -xHost -fopenmp -mkl nodeperf.c -o nodeperf
[@cn01-10 ex_05_03]$ export OMP_NUM_THREADS=20
[@cn01-10 ex_05_03]$ export OMP_PLACES=cores
```

One output is presented on appendix C.



Graph 11: Performance (GFLOPS) versus number of threads for different hosts – Compiler mpiicc –O2

Compiling the program "nodeperf" for different number of threads and hosts was possible to plot graph 11. One can notice that the performance grows linearly with the number of threads. The last point of the graph, related to 20 threads, however, presents a sharp decline in performance for many hosts. This behaviour is detailed in table 11, where it is possible to notice the high standard deviation related to the measure. This case is particularly true for the host cn02-30.

[5] https://software.intel.com/en-us/mkl/features/benchmarks

[6] Intel Math Kernel Library for the Linux Operating System User's Guide, October 2007, Document Number: 314774-005US.

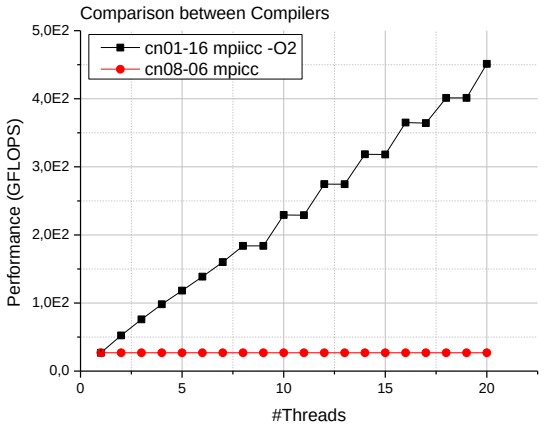
Exercise 05

Table 11 : 20 Threads Behaviour

	Run 1	Run 2	Run 3	Mean	Std
	GFLOPS	GFLOPS	GFLOPS	GFLOPS	GFLOPS
cn01-16	452,33	450,73	450,63	451,23	0,96
cn01-02	225,49	287,67	437,33	316,83	108,89
cn02-16	241,41	446,63	309,18	332,41	104,56
cn02-28	448,97	309,18	297,09	351,75	84,41
cn02-29	339,82	449,60	447,61	412,34	62,81
cn02-30	308,61	301,93	271,16	293,90	19,97
cn02-35	302,95	446,87	303,22	351,01	83,01
cn02-36	449,47	323,60	330,30	367,79	70,81

Compiling the program with mpicc, on the other hand, it is possible to notice a completely different behaviour in performance (see graph 12). The command line used is the following:

```
$ mpicc -fopenmp 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
```



Graph 12: Performance (GFLOPS) versus number of threads for different Compilers– Compiler mpiicc –O2 and mpicc

The result table 6 presents the maximum performance achieved for the mpiic and mpic compilers in GFLOPS. One can notice from graph 12 that when using the mpicc compiler the output is constant, not depending on the number of threads considered. It is possible to conclude that it is not possible to access the performance of the machine compiling the code in this manner. This can be explained by the fact that the program nodeperf was designed to run optimally with Intel compilers.

Result table 6: Maximum performance achieved (GFLOPS) – comparison between compilers, with the standard deviation of the mean.

Compiler	Mean	Std
	GFLOPS	GFLOPS
mpiicc -O2	451,23	0,96
mpicc	27,02	0,04

Exercise 05

APPENDIX A - Output Ping Pong:

```
 benchmarks to run PingPong
#---------------------------------------------------------------
#    Intel (R) MPI Benchmarks 4.0, MPI-1 part
#---------------------------------------------------------------
# Date                  : Tue Jan 15 02:16:15 2019
# Machine               : x86_64
# System                : Linux
# Release               : 2.6.32-573.12.1.el6.x86_64
# Version               : #1 SMP Wed Dec 16 11:39:16 CET 2015
# MPI Version           : 3.0
# MPI Thread Environment:

# New default behavior from Version 3.2 on:

# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:

# /u/shared/programs/x86_64/intel/impi_5.0.1/bin64/IMB-MPI1 PingPong

# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong

#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         0.21         0.00
            1         1000         0.21         4.49
            2         1000         0.21         9.07
            4         1000         0.22        17.71
            8         1000         0.21        35.98
           16         1000         0.20        74.81
           32         1000         0.31        98.77
           64         1000         0.29       207.54
          128         1000         0.30       401.57
          256         1000         0.32       758.24
          512         1000         0.39      1266.93
         1024         1000         0.45      2185.12
         2048         1000         0.61      3201.88
         4096         1000         1.03      3796.11
         8192         1000         1.62      4826.99
        16384         1000         2.85      5477.77
        32768         1000         5.08      6145.54
        65536          640         8.07      7747.32
       131072          320        14.43      8662.56
       262144          160        26.86      9306.46
       524288           80        49.09     10185.91
      1048576           40        93.44     10702.14
      2097152           20       182.67     10948.68
      4194304           10       654.24      6113.92


# All processes entering MPI_Finalize
```

Exercise 05

APPENDIX B - Output Stream:

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 80000000 (elements), Offset = 0 (elements)
Memory per array = 610.4 MiB (= 0.6 GiB).
Total memory required = 1831.1 MiB (= 1.8 GiB).
Each kernel will be executed 50 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 1
Number of Threads counted = 1
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 64309 microseconds.
   (= 64309 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:          13197.1     0.097150     0.096991     0.098976
Scale:         13051.4     0.098191     0.098074     0.098714
Add:           13730.0     0.140080     0.139840     0.143345
Triad:         13778.4     0.139551     0.139349     0.140918
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Exercise 05

APPENDIX C - Output Nodeperf:

```
Multi-threaded MPI detected

The time/date of the run...  at Fri Jan 18 04:31:17 2019

This driver was compiled with:
     -DITER=4 -DLINUX -DNOACCUR -DPREC=double
Malloc done.  Used 1846080096 bytes
(0 of 1): NN lda=15000 ldb= 192 ldc=15000 0 0 0 450016.843 cn01-10
```