Foundations of High Performance Computing

Assignment 1

Academic year: 2021-2022
Author: Erika Lena
Username: elena
Id: SM3500498

1 Introduction

The following paper is meant to illustrate the work done for assignment 1 of Foundation of High Performance Computing course at the University of Trieste.
The code for Section 1 has been written in C, while the other tests make use of simple bash scripts. For data analysis and charts R was used.
All the code and the results are available on Github: `https://github.com/erikalena/hpc_assignment1`

2 Section 1

2.1 Ring

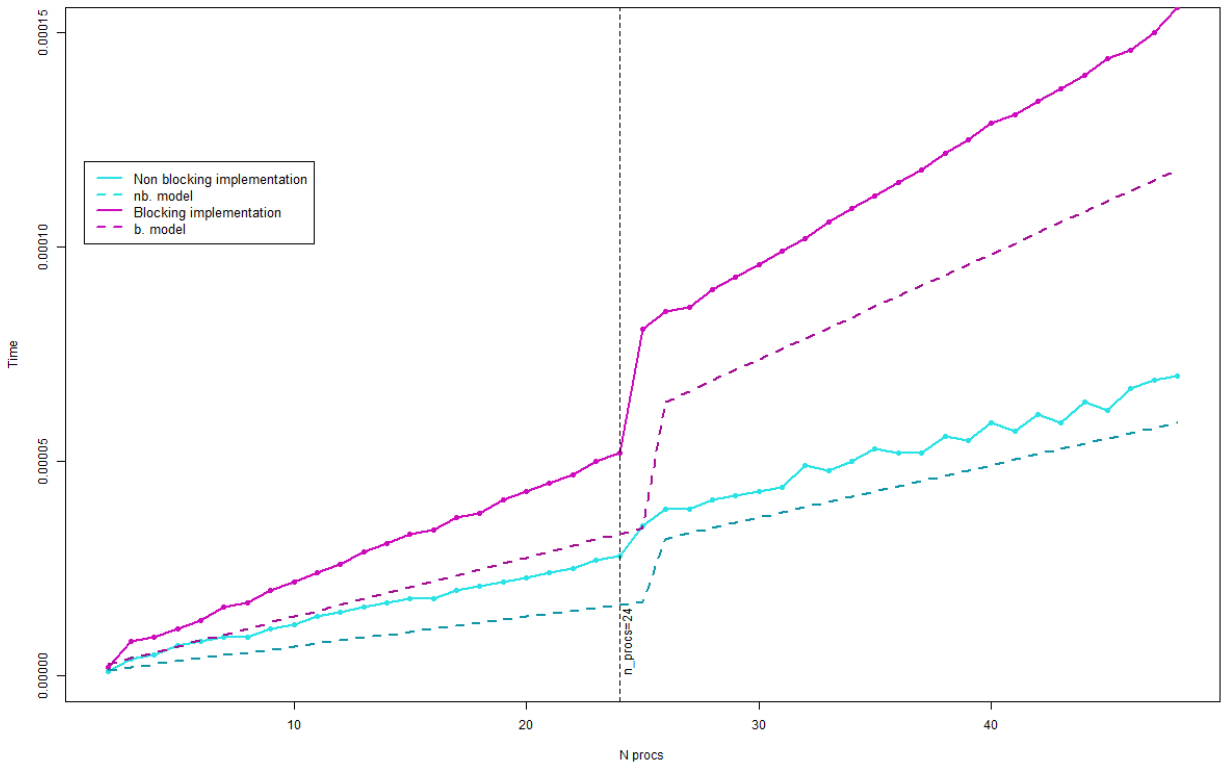
The first part of the assignment was focused on MPI programming and asked to develop a simple code which does the following:

- as first step P sends a message (msgleft = rank) to its left neighbour (P-1) and receives from its right neighbour (P+1) and send another message (msgright = -rank) to P+1 and receive from P-1.
- it then does enough iterations till all processors have received back the initial messages. The content of the message change at each message passing between different processes, messages are then identified by their tag.

The whole process has been repeated a suitable number of times (10000 repetitions) in order to collect enough statistics. Two different implementations for this simple message exchange have been tried; they are both provided in file *section1/ring.c.*

- The first one uses non-blocking operations (MPI_Isend, MPI_Irecv), the code executed by each process is the same and they all start sending and receiving at the same time. Still, this is possible because messages have small sizes and we do not really have to check for each connection to be completely closed.
- The second implementation, instead, is blocking and provides a more general solution which can be applied even for larger messages. To avoid deadlocks since the communication is blocking, just half of the processes start sending messages and the others receive, then roles are exchanged.

The code measures the time taken by each implementation to exchange messages between each other. It has been run on rings of different sizes using 2 to 48 processors.
Results are summarized in the following plots and the corresponding *.csv* file can be found in *section1 folder.*



The amount of time taken to exchange messages and complete the ring is almost linear in the number of processors used.
Indeed, each process sends and receives two messages (4 communications) for *ntimes*; nonetheless since a lot of small messages are exchanged, the whole communication id latency-bound.

First thing we can observe is that non blocking implementation is faster than the blocking one, as we expected.
Second thing, in both cases the times increase notably when the code is run on more than 24 cores, which is where the processes are split on more than one thin core and for this reason communication time increases significantly.

To check for the obtained results, expected network performances for the ring in both cases has been plotted as well. In non blocking implementation all processes start sending messages all together, they both send two messages at a time, then the whole time taken can be roughly approximated as:

$$T = n \cdot \left(\frac{2 \cdot size}{B} + \lambda \right) \tag{1}$$

where *size* is given by the message size, which is two bytes, while B is the bandwidth and λ is he latency (empirical results obtained on sockets and nodes respectively for section 2 has been used to estimate the time taken). Lastly, n factor is equal to the number of processes and it is needed because each message exchange is repeated n *times* to complete the ring.

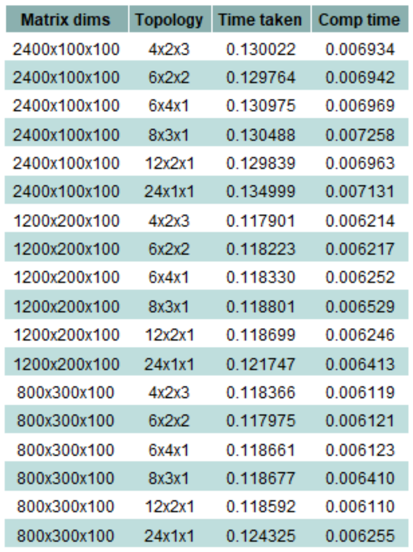
For blocking implementation we also have to consider that the processes work alternately, before starting sending messages half of processes wait until they have received first two messages and so on, then the whole time taken is almost two times the previous one.
Replacing in formula empirical results obtained in section 2 (for bandwidth and latency), we obtain the following models:

$$T = \begin{cases} n \cdot (\dfrac{2 \cdot size}{12000} + 0.68 \cdot 10^{-6}) & \text{if } n \leq 24 \\ n \cdot (\dfrac{2 \cdot size}{12000} + 1.23 \cdot 10^{-6}) & \text{if } n > 24 \end{cases} \tag{2}$$

this formula has been used to plot theoretical models in the plot above.

2.2 3D matrix

In this second part, collective operations have been used to distribute 3D matrices data between processes, each of them computed a partial sum and root recollected the data providing final matrix given by the sum of initial ones. Processes are arranged in different 1D, 2D, 3D virtual topologies, but from the obtained results it can be seen how different configurations do not affect the time taken to compute results, this is because processes do not have to exchange information and they just communicate with root.



Matrix dims	Topology	Time taken	Comp time
2400x100x100	4x2x3	0.130022	0.006934
2400x100x100	6x2x2	0.129764	0.006942
2400x100x100	6x4x1	0.130975	0.006969
2400x100x100	8x3x1	0.130488	0.007258
2400x100x100	12x2x1	0.129839	0.006963
2400x100x100	24x1x1	0.134999	0.007131
1200x200x100	4x2x3	0.117901	0.006214
1200x200x100	6x2x2	0.118223	0.006217
1200x200x100	6x4x1	0.118330	0.006252
1200x200x100	8x3x1	0.118801	0.006529
1200x200x100	12x2x1	0.118699	0.006246
1200x200x100	24x1x1	0.121747	0.006413
800x300x100	4x2x3	0.118366	0.006119
800x300x100	6x2x2	0.117975	0.006121
800x300x100	6x4x1	0.118661	0.006123
800x300x100	8x3x1	0.118677	0.006410
800x300x100	12x2x1	0.118592	0.006110
800x300x100	24x1x1	0.124325	0.006255

Topologies are pretty useless in this exercise and the same results can be obtained without the use of any topology.
In the last column are reported the times needed by each process to compute the sum of their submatrices.
Then the majority of the time needed is taken by communications and it can be roughly estimated using the usual theoretical communication model:

$$T_c = n \cdot \left(\frac{size}{B} + \lambda \right)$$

where n is the number of collective operations performed.

3 Section 2

The code provided by the Intel MPI benchmark, and the Ping Pong test in particular, has been used to estimate network performances on ORFEO platform. Both ORFEO High speed network (100 Gb) and in band management network (25 Gb) have been used and compared to check for their differences.
The results of all the tests are in *section2/csv* subfolder. These empirical results have been compared with the very simple communication model provided by the following formula:

$$T_{comm} = \lambda + \frac{size_{msg}}{B} \tag{3}$$

Even if the trend of the model is similar to the data collected, it doesn't fit the data perfectly, then a least-square fitting model is provided in order to obtain a more accurate approximation of latency (λ) and bandwidth (B), whose results have been added to each *.csv* file.
If times have more or less the same behaviour for all the configurations, bandwidth behaves slightly differently; in **figure 1** results obtained using Infiniband with OpenMPI are plotted with respect to map by core, by socket and by node.

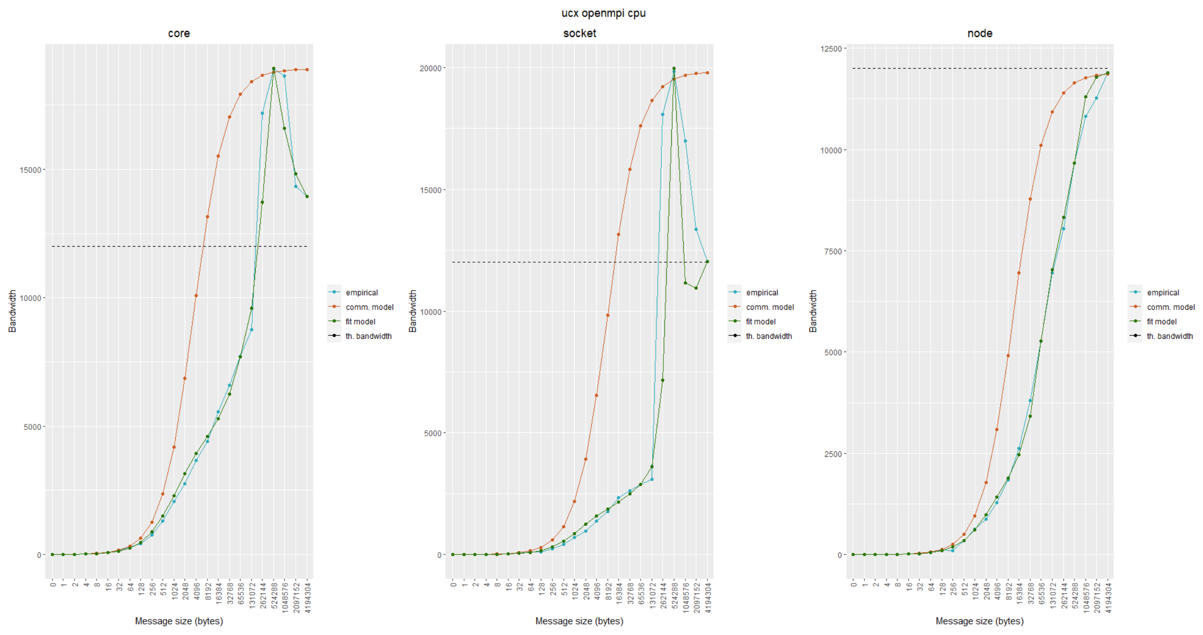


Figure 1: Results obtained for the bandwidth with respect to map by core, by socket and by node using Infiniband and OpenMPI.

When we map by core and by socket we see that the experimental bandwidth is much higher than the maximum speed of the network (which is about 100 Gb = 12.5 GB for Infiniband), even higher than the one we obtain by exploiting shared memory. This is mainly due to an appropriate use of cache hierarchy and of in-cache copy operations. When the same tests are run using Intel, instead of OpenMPI, a smaller bandwidth is obtained on sockets in particular, which indicates non-use or a far less efficient use of the cache.

To further investigate this result, the same test has been repeated adding a flag which explicitly says to not use the cache. The same was done using both Intel and OpenMPI and in **figure 2** it can be seen how the results are now much lower.

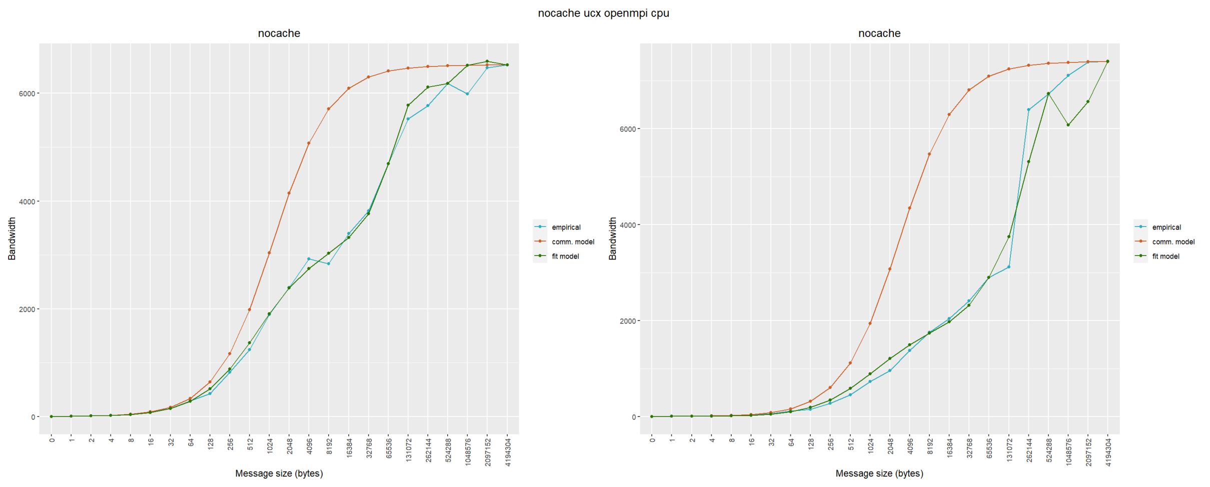


Figure 2: Results obtained for the bandwidth with respect to map by core and map by socket when the use of the cache is explicitly excluded.

When we change the network the bandwidth gets much smaller, but we can still see how the maximum bandwidth reached when mapping by core and by socket overcomes the maximum network speed. This is stll due to some intranode communication and and of in-cache copy operations.
Tests have bee run on both cpu and gpu nodes, but no substantial differences were found.

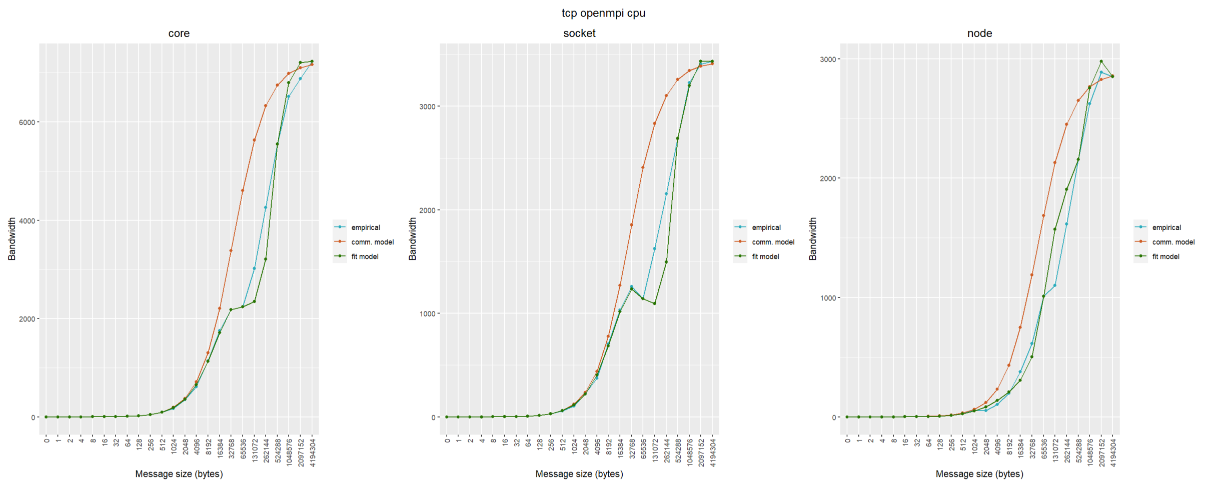


Figure 3: Results obtained for the bandwidth with respect to map by core and map by socket when using tcp and OpenMPI.

4 Section 3

Third section was focused on the execution of already provided implementation of Jacobi algorithm. A 3D grid point of dimensions 1200x1200x1200 was taken and the code was executed on:

- 4/8/12 cores within the same socket
- 4/8/12 cores across two sockets
- 12/24/36/48 processes across two nodes

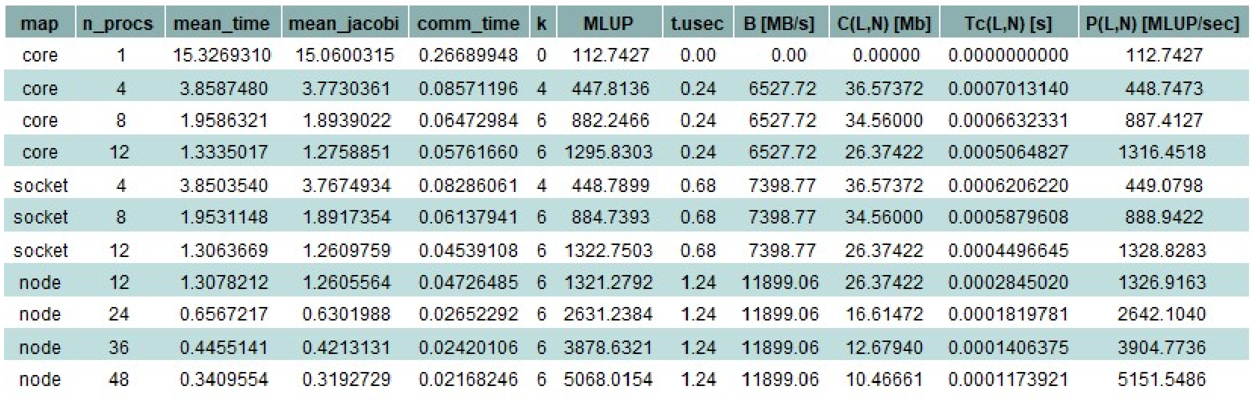
The code was run both on cpu and gpu nodes; the results can be found in *section3/*.csv*.
The following tables summarize results obtained, reporting the number of processes and how they are mapped and for each configuration the global mean time taken, the time needed for jacobi computation and the communication time, which is given by the difference of the previous ones.
The code provides also the number of MLUP (Million Lattice Update Per Second), which were compared with theoretical expected results obtained through the following formula:

$$P(L,N) = \frac{L^3 \cdot N}{Ts(L) + Tc(L,N)} \tag{4}$$

where L is the *subdomain size* for each process, Ts is the time *sequential time* taken by one single process, while Tc is the *communication time.*

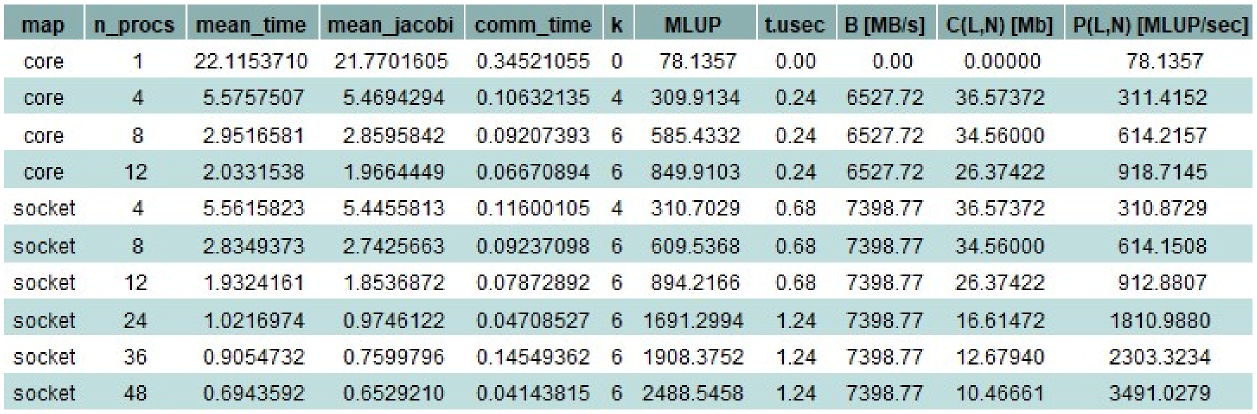
$$Tc(L,N) = \frac{L^2 \cdot k \cdot 2 \cdot 8}{B} + k \cdot lambda \tag{5}$$

For bandwidth and latency empirical results from section 2 have been used.
The code has been run on a 3D matrix with $L = 1200$, keeping the total problem size constant and testing the scalability as the number of processors increases (strong scalability). Resulting performances (MLUP/sec) are reported in last column.
Finally, the same code has been run again keeping a fixed problem size ($L = 1200$) per processor in order to check even for weak scalability. It can be seen how sequential time is almost constant as well as communication time and the global performances increase linearly in the number of processors.



map	n_procs	mean_time	mean_jacobi	comm_time	k	MLUP	t.usec	B [MB/s]	C(L,N) [Mb]	Tc(L,N) [s]	P(L,N) [MLUP/sec]
core	1	15.3269310	15.0600315	0.26689948	0	112.7427	0.00	0.00	0.00000	0.0000000000	112.7427
core	4	3.8587480	3.7730361	0.08571196	4	447.8136	0.24	6527.72	36.57372	0.0007013140	448.7473
core	8	1.9586321	1.8939022	0.06472984	6	882.2466	0.24	6527.72	34.56000	0.0006632331	887.4127
core	12	1.3335017	1.2758851	0.05761660	6	1295.8303	0.24	6527.72	26.37422	0.0005064827	1316.4518
socket	4	3.8503540	3.7674934	0.08286061	4	448.7899	0.68	7398.77	36.57372	0.0006206220	449.0798
socket	8	1.9531148	1.8917354	0.06137941	6	884.7393	0.68	7398.77	34.56000	0.0005879608	888.9422
socket	12	1.3063669	1.2609759	0.04539108	6	1322.7503	0.68	7398.77	26.37422	0.0004496645	1328.8283
node	12	1.3078212	1.2605564	0.04726485	6	1321.2792	1.24	11899.06	26.37422	0.0002845020	1326.9163
node	24	0.6567217	0.6301988	0.02652292	6	2631.2384	1.24	11899.06	16.61472	0.0001819781	2642.1040
node	36	0.4455141	0.4213131	0.02420106	6	3878.6321	1.24	11899.06	12.67940	0.0001406375	3904.7736
node	48	0.3409554	0.3192729	0.02168246	6	5068.0154	1.24	11899.06	10.46661	0.0001173921	5151.5486

Figure 4: Results obtained for jacobi code run on 4,8,12,(24) cores across one and two sockets and 12,24,36,48 cores across two nodes.



map	n_procs	mean_time	mean_jacobi	comm_time	k	MLUP	t.usec	B [MB/s]	C(L,N) [Mb]	P(L,N) [MLUP/sec]
core	1	22.1153710	21.7701605	0.34521055	0	78.1357	0.00	0.00	0.00000	78.1357
core	4	5.5757507	5.4694294	0.10632135	4	309.9134	0.24	6527.72	36.57372	311.4152
core	8	2.9516581	2.8595842	0.09207393	6	585.4332	0.24	6527.72	34.56000	614.2157
core	12	2.0331538	1.9664449	0.06670894	6	849.9103	0.24	6527.72	26.37422	918.7145
socket	4	5.5615823	5.4455813	0.11600105	4	310.7029	0.68	7398.77	36.57372	310.8729
socket	8	2.8349373	2.7425663	0.09237098	6	609.5368	0.68	7398.77	34.56000	614.1508
socket	12	1.9324161	1.8536872	0.07872892	6	894.2166	0.68	7398.77	26.37422	912.8807
socket	24	1.0216974	0.9746122	0.04708527	6	1691.2994	1.24	7398.77	16.61472	1810.9880
socket	36	0.9054732	0.7599796	0.14549362	6	1908.3752	1.24	7398.77	12.67940	2303.3234
socket	48	0.6943592	0.6529210	0.04143815	6	2488.5458	1.24	7398.77	10.46661	3491.0279

Figure 5: Results obtained for jacobi code run on 4,8,12,(24),(36),(48) cores across one and two sockets on a gpu node with hyperthreading enabled.

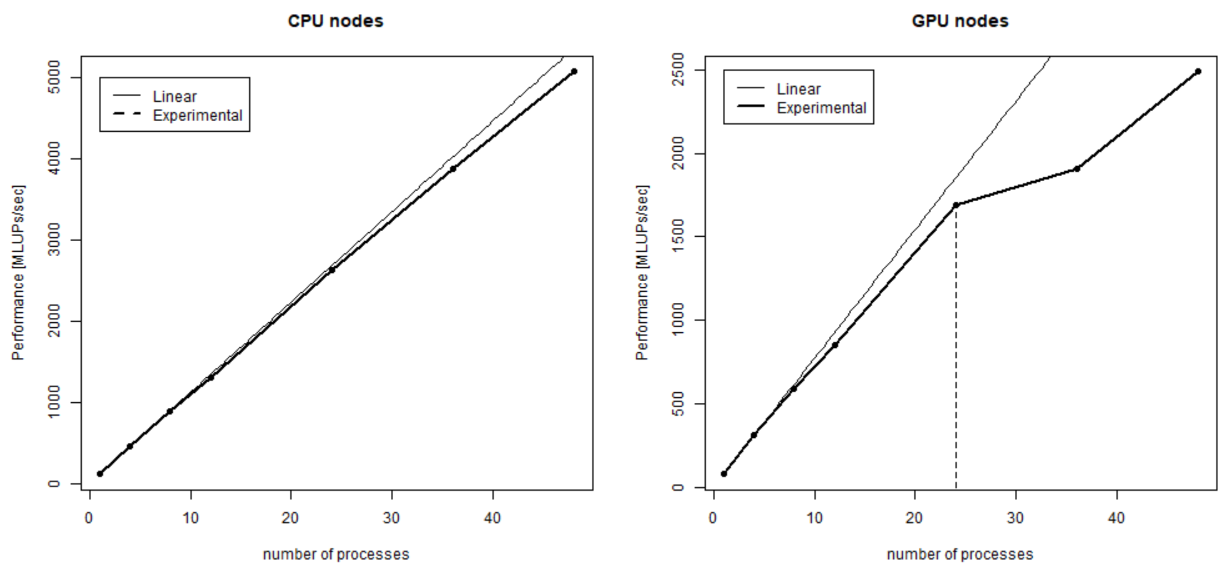
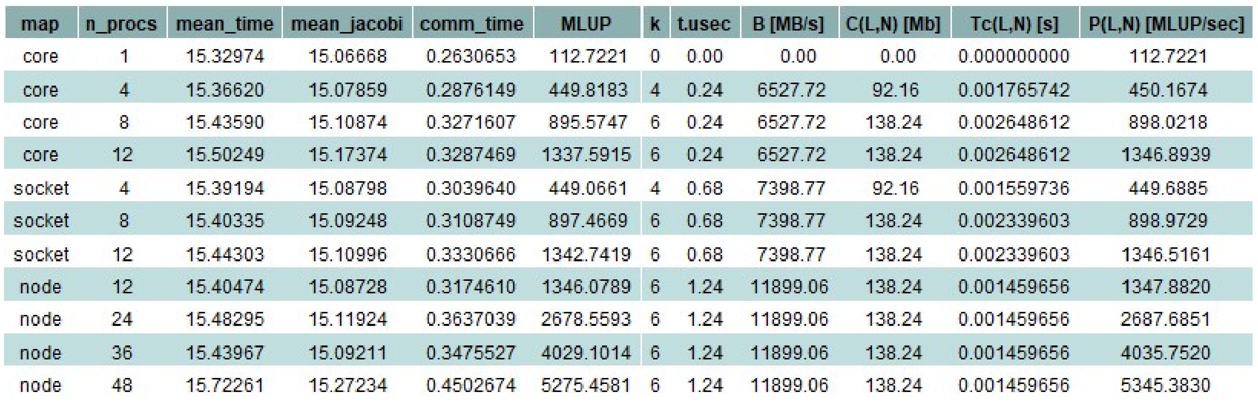


Figure 6: Comparison of the results obtained with respect to strong scalability on CPU an GPU nodes. It is possible to see the drop of performances on GPU nodes when hyperthreading is exploited.



map	n_procs	mean_time	mean_jacobi	comm_time	MLUP	k	t.usec	B [MB/s]	C(L,N) [Mb]	Tc(L,N) [s]	P(L,N) [MLUP/sec]
core	1	15.32974	15.06668	0.2630653	112.7221	0	0.00	0.00	0.00	0.000000000	112.7221
core	4	15.36620	15.07859	0.2876149	449.8183	4	0.24	6527.72	92.16	0.001765742	450.1674
core	8	15.43590	15.10874	0.3271607	895.5747	6	0.24	6527.72	138.24	0.002648612	898.0218
core	12	15.50249	15.17374	0.3287469	1337.5915	6	0.24	6527.72	138.24	0.002648612	1346.8939
socket	4	15.39194	15.08798	0.3039640	449.0661	4	0.68	7398.77	92.16	0.001559736	449.6885
socket	8	15.40335	15.09248	0.3108749	897.4669	6	0.68	7398.77	138.24	0.002339603	898.9729
socket	12	15.44303	15.10996	0.3330666	1342.7419	6	0.68	7398.77	138.24	0.002339603	1346.5161
node	12	15.40474	15.08728	0.3174610	1346.0789	6	1.24	11899.06	138.24	0.001459656	1347.8820
node	24	15.48295	15.11924	0.3637039	2678.5593	6	1.24	11899.06	138.24	0.001459656	2687.6851
node	36	15.43967	15.09211	0.3475527	4029.1014	6	1.24	11899.06	138.24	0.001459656	4035.7520
node	48	15.72261	15.27234	0.4502674	5275.4581	6	1.24	11899.06	138.24	0.001459656	5345.3830

Figure 7: Results obtained for jacobi code run on 4,8,12,(24) cores across one and two sockets and 12,24,36,48 cores across two nodes by keeping problem size per processor fixed and equal to 1200^3.