Report Assignment 1

Romina Doz

December 2021

1 Section 1

1.1 Ex 1

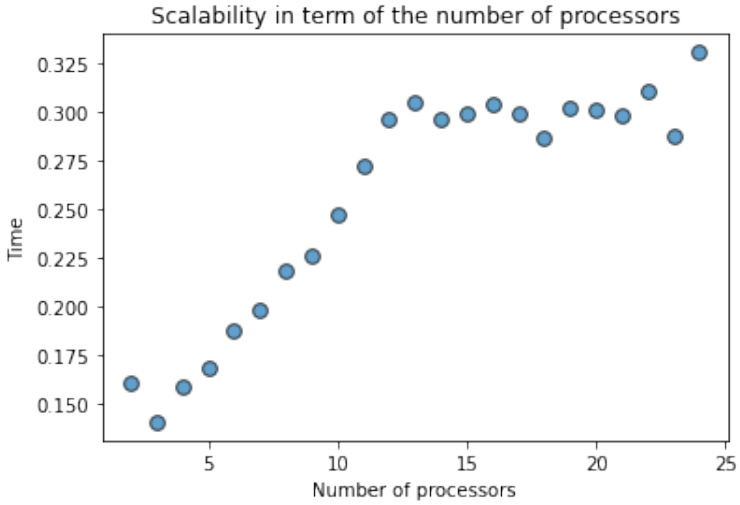
The first exercise consists on a stream of messages passed through a set of processors distributed on a ring topology.
The steps executed by the program are the following:

- At first, each processor sends its $rank$ to the previous one and $-rank$ to the following one. The tag of the message is $rank * 10$
- Each processor receives two messages, one from the previous processor and one from the next one
- Both messages are propagated keeping the same tag and forwarding them to the successor or predecessor depending on whether the original message came from the left (previous processor) or right (next processor). If the message came from left, the processor add its $rank$ to the message before forwarding it; if it came from right, it subtract its $rank$.
- The previous step is repeated until each processor receives back the initial message, that is the message with $tag = rank * 10$
- At the end, each message has traveled the entire ring. A .txt file is produced containing the following message printed by each processor:

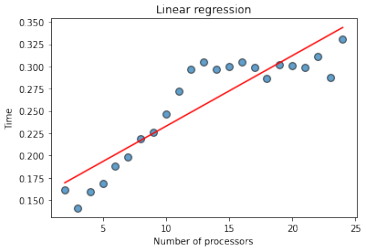
  "I am process irank and I have received np messages. My final messages have tag $itag$ and value msg-left,msg-right"

  and the time elapsed .

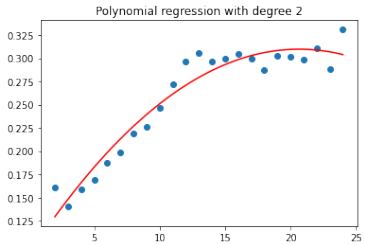
It is possible to analyze the scalability of the program in term of the number of processors. The program has been run on an increasing number of processors starting from 2 till 24. The following plot shows how the elapsed time increases when the number of processors grows.



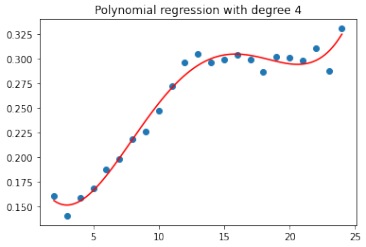
This trend can be modeled with a linear regression or a polynomial regression. The linear regression doesn't fit well the data (coefficient of determination = 0.81) while a quadratic regression has coefficient of determination of 0.93 and can be considered a good fit. Also a polynomial regression with degree 4 is used and a coefficient of determination of 0.98 is reached but graphically this regression seems overfitting the dataset.



(a)



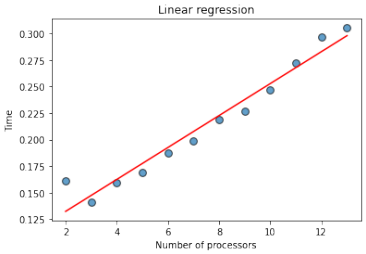
(b)



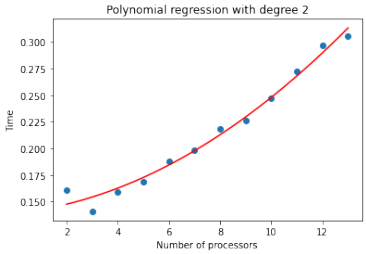
(c)

It can be noticed that there is a more precise trend when the processors are less

than 12. Modeling only the first 12 data, the linear regression has a coefficient of determination of 0.96, while the quadratic has 0.98 and fits well the data:



(a)



(b)

This reflects the architecture of the program because each time a new processor is added, there are 2 more messages that have to travel around all the ring and the ring is bigger. The total number of messages received and sent is :

$$N_{tot} = 2 * P^2$$

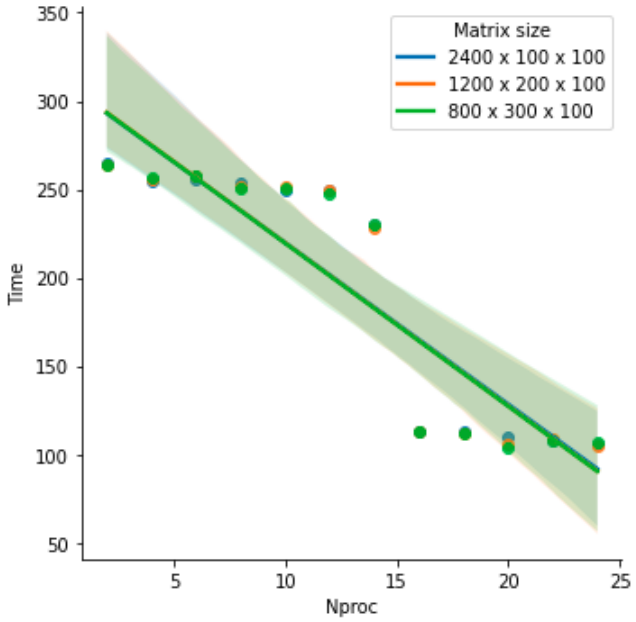
The implementation of the models and the generation of the plots can be found in the following python notebook:
`https://colab.research.google.com/drive/1bjQZVQE_5ZOBVw-K2w0TM2WU97R9NNkb?usp=sharing`

1.2 Ex 2

It is required to implement a program which exploits MPI collective operations to compute 3d matrix-matrix addition. The matrices are initialized with random numbers in double precision and are represented with a single array of $size = X * Y * Z$ where X,Y,Z are the dimensions of each matrix. These dimensions are given in input from the command line. A cartesian 3D topology is used and the ranks are reordered so that communicating processes are closer together physically in the cluster. The different domains (1D/2D/3D distribution) are not distinguished in this program, because to compute the sum of two matrices it is necessary only to compute the element by element sum without interacting with the neighbours. So, choosing any of the domains would be equivalent. The program has been run on a THIN node with an increasing number of processors. Three different sizes are tested:

- 2400 x 100 x 100
- 1200 x 200 x 100
- 800 x 300 x 100

and, because the total number of elements is the same for all the sizes, the performance is the more or less equivalent for all the three configurations.



The data points are fitted using a linear regression and the coefficient of determination is 0.81.
It can be noticed that there is a huge improvement of the performance when passing from 14 to 16 processors.
The implementation of the model and the generation of the plot can be found in the following python notebook:
`https://colab.research.google.com/drive/1_zlC6MSOR5iALLOC_SLWlmYpvWhJLPzd?usp=sharing`

2 Section 2

The aim of this exercise is to evaluate the performances with different topologies and networks. The Intel® MPI Benchmarks tool has been used, in particular the PingPong benchmark.
To have an idea of the possible commands to run, there is the help command:

```
./IMB-MPI1 -help
```

The following bash script has been run in order to test both Infiniband and gigsbit, IntelMPI and openmpi in various configurations.

```
or run in {1..10}; do
mpirun --report-bindings -np 2 -npernode 2 ./IMB-MPI1
```

```
PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ucx_n2.txt
done

for run in {1..10}; do
mpirun -np 2  --report-bindings --map-by socket ./IMB-MPI1
PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ucx_socket.txt
done

for run in {1..10}; do
mpirun -np 2 --mca pml ob1 --report-bindings --map-by socket
--mca btl vader,self ./IMB-MPI1 PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ob1_socket_vader.txt
done


for run in {1..10}; do
mpirun -np 2 --mca pml ob1 --report-bindings --map-by socket
--mca btl tcp,self ./IMB-MPI1 PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ob1_socket.txt
done


for run in {1..10}; do
mpirun -np 2 --mca pml ob1 --report-bindings --map-by node
--mca btl tcp,self ./IMB-MPI1 PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ob1_node.txt
done

for run in {1..10}; do
mpirun -np 2 --mca pml ob1 --report-bindings --map-by core
--mca btl vader,self ./IMB-MPI1 PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ob1_core_vader.txt
done

for run in {1..10}; do
mpirun -np 2 --mca pml ob1 --report-bindings --map-by core
--mca btl tcp,self ./IMB-MPI1 PingPong -mslog 28
2>/dev/null |grep -v ^# |grep -v -e '^$' >> ob1_core.txt
done

for run in {1..10}; do
mpirun -np 2 -ppn=2 -env I_MPI_DEBUG 5
-genv I_MPI_PIN_PROCESSOR_LIST 0,2 ./IMB-MPI1 PingPong
-mslog 28  2>/dev/null |grep -v ^# |grep -v -e '^$' >> intel_socket.txt
```

```
done


for run in {1..10}; do
mpirun -np 2 -ppn=2 -env I_MPI_DEBUG 5
-genv I_MPI_PIN_PROCESSOR_LIST 0,1 ./IMB-MPI1 PingPong
-mslog 28  2>/dev/null |grep -v ^# |grep -v -e '^$' >> intel_proc.txt
done
```

For each configuration, 10 trials have been performed and the values of t and Mbytes/sec have been averaged to obtain t (computed) and Mbytes/sec (computed). The data of all the experiments are available in the following github repository:
`https://github.com/u-doz/HighPerformanceComputing`

After that, for each configuration, t (computed) and Mbytes/sec (computed) have been used to estimate latency and bandwidth parameters. The estimated latency is considered equal to t (computed) when the size of the message is 0; while estimated bandwidth is the asymptotic value of the Mbytes/sec (computed) when the size of the message grows.

These estimated values are compared to the latency and bandwidth calculated by fitting the data with a linear model. In fact, a very simple model of the network performance can be expressed by:

$$T_{comm} = \lambda + \frac{size_{message}}{b_{network}}$$

So, a least-squared fitting model is simply a linear model in which the intercept represents the latency and the inverse of the slope of the regression line estimates the bandwidth.
The comparison between estimated and fitted values is performed by calculating the percentage difference between them.
The implementation of the least-squared fitting model and the estimation of latency and bandwidth parameters can be found in the following python notebook:
`https://colab.research.google.com/drive/1lq-rE9lkt3U0UMIb4QErDH35P82ZgDlI?usp=sharing`

The results are resumed in this table:

	Latency [sec] (fitted)	Bandwidth [Gb/sec] (fitted)	Accuracy of least-squared fitting model	Latency [sec] (estimated)	Bandwdith [Gb/sec] (estimated)	Latency (% diff)	Bandwith (% diff)
UCX 1 processor per node	3.50	11.86	0.998	0.99	11.88	1.12	0
UCX 2 processors per node	-0.88	13.81	0.991	0.20	21.69	3.18	0.44
UCX map by socket	2.7	14.36	0.976	0.41	21.56	1.47	0.40
OB1 BTL map by socket VADER	-0.02	10.31	0.997	0.53	13.73	2.16	0.28
OB1 BTL map by socket	15.70	3.42	0.997	8.33	3.39	0.61	0.01
OB1 BTL map by node	29.95	2.75	0.997	16.25	2.75	0.59	0
OB1 BTL map by core VADER	-1.64	9.63	0.996	0.25	14.34	2.72	0.39
OB1 BTL map by core	9.10	7.12	0.997	5.33	7.05	0.52	0.01
INTEL some socket	-1.59	11.25	0.990	0.23	17.62	2.68	0.44
INTEL contiguous processors	0.02	6.25	0.999	0.43	7.60	1.82	0.19

It can be noticed that the latency obtained by fitting the data is often under-estimated (there are some negative values) or over-estimated (in particular for OB1 without VADER), probably because of the values corresponding to higher data sizes that don't fit well with a linear model (see Figure 1). So the percentage difference between estimated and fitted latency is quite high.

For bandwidth, the percentage difference between estimated and fitted values is smaller in general and for some configurations is 0 or 0.01 that implies a good approximation of the bandwidth with the slope of the linear regression line. When the error is higher, the estimated bandwidth is greater that the one computed by fitting the data. That can be due to the fact that to approximate the asymptotic value of the Mbytes/sec (computed) it has been considered the max value (see Figure 2).

INTEL and UCX mapped by socket have different performances: INTEL shows a smaller latency but also a smaller bandwidth:

	Latency [sec] (fitted)	Bandwidth [Gb/sec] (fitted)	Accuracy of least-squared fitting model	Latency [sec] (estimated)	Bandwdith [Gb/sec] (estimated)	Latency (% diff)	Bandwith (% diff)
UCX map by socket	2.7	14.36	0.976	0.41	21.56	1.47	0.40
INTEL same socket	-1.59	11.25	0.990	0.23	17.62	2.68	0.44

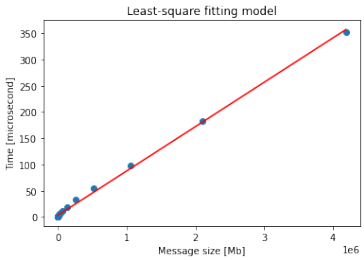
INTEL working on contiguous processors has an higher latency and smaller bandwidth with respect to UXC mapped by node, that is more performant:

	Latency [sec] (fitted)	Bandwidth [Gb/sec] (fitted)	Accuracy of least-squared fitting model	Latency [sec] (estimated)	Bandwdith [Gb/sec] (estimated)	Latency (% diff)	Bandwith (% diff)
UCX 2 processors per node	-0.88	13.81	0.991	0.20	21.69	3.18	0.44
INTEL contiguous processors	0.02	6.25	0.999	0.43	7.60	1.82	0.19

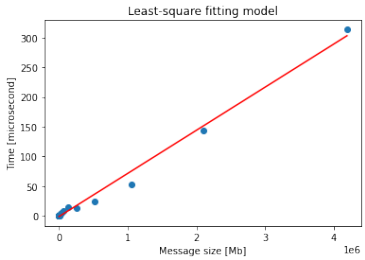
It is also possible to notice an high increase of performances when using OB1 PML with BTL VADER (shared memory) instead of TC:

	Latency [sec] (fitted)	Bandwidth [Gb/sec] (fitted)	Accuracy of least-squared fitting model	Latency [sec] (estimated)	Bandwdith [Gb/sec] (estimated)	Latency (% diff)	Bandwith (% diff)
OB1 BTL map by socket	15.70	3.42	0.997	8.33	3.39	0.61	0.01
OB1 BTL map by socket VADER	-0.02	10.31	0.997	0.53	13.73	2.16	0.28
OB1 BTL map by core	9.10	7.12	0.997	5.33	7.05	0.52	0.01
OB1 BTL map by core VADER	-1.64	9.63	0.996	0.25	14.34	2.72	0.39

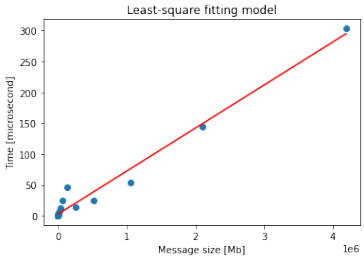
The linear regressions and the bandwidth estimations can be visualized by the following plots.



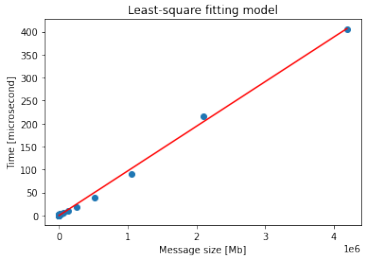
(a) UCX 1 processor per node



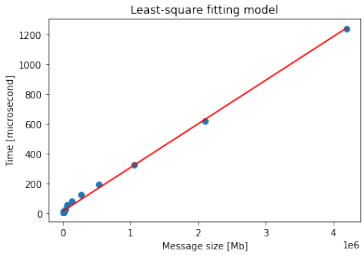
(b) UCX 2 processors per node



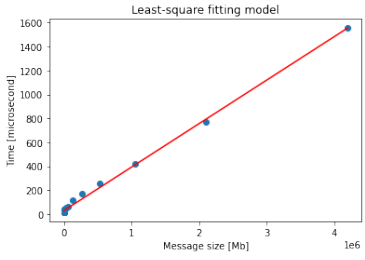
(c) UCX map by socket



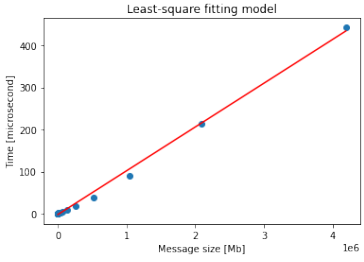
(d) OB1 BTL map by socket VADER



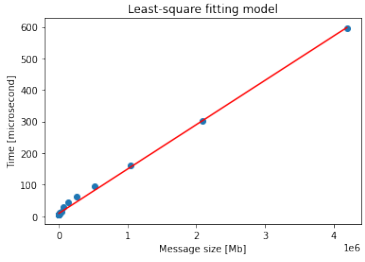
(e) OB1 BTL map by socket



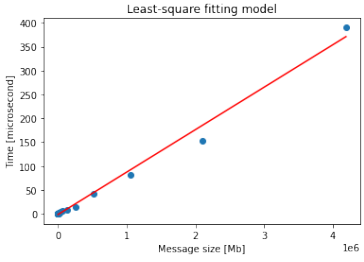
(f) OB1 BTL map by node



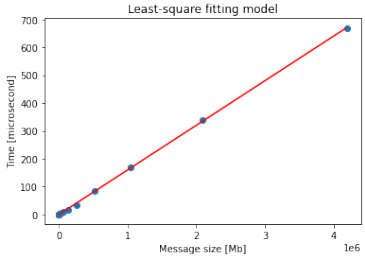
(g) OB1 BTL map by core VADER



(h) OB1 BTL map by core

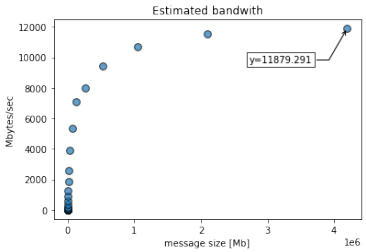


(i) INTEL same socket

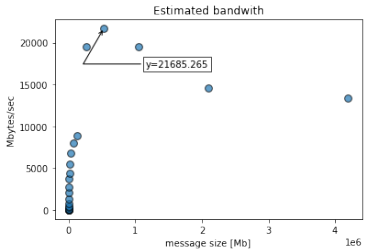


(j) INTEL contiguous processors

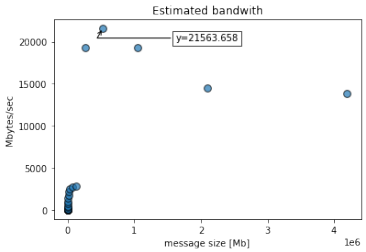
Figure 3: least-squared fitting model for each configuration



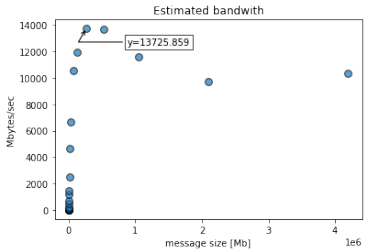
(a) UCX 1 processor per node



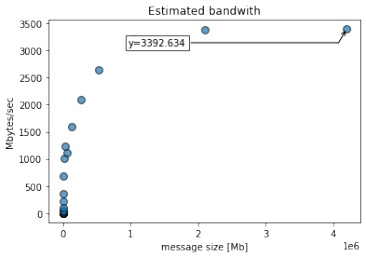
(b) UCX 2 processors per node



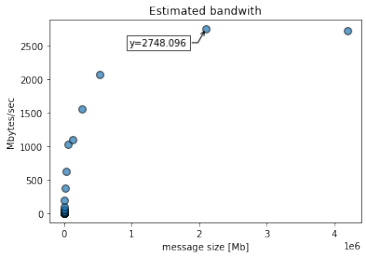
(c) UCX map by socket



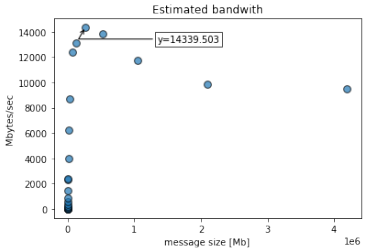
(d) OB1 BTL map by socket VADER



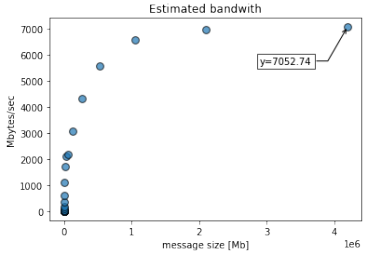
(e) OB1 BTL map by socket



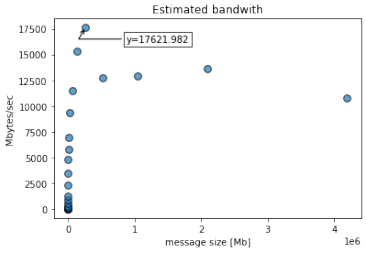
(f) OB1 BTL map by node



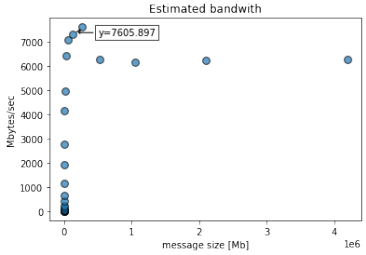
(g) OB1 BTL map by core VADER



(h) OB1 BTL map by core



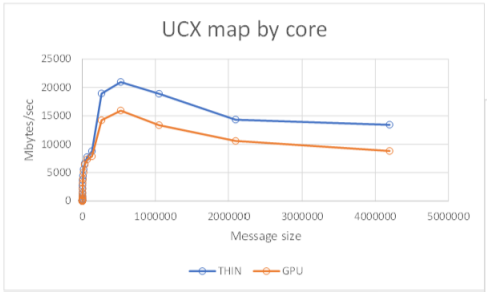
(i) INTEL same socket

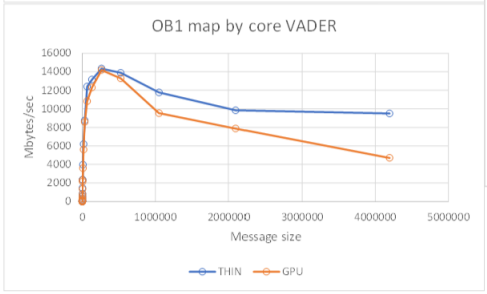


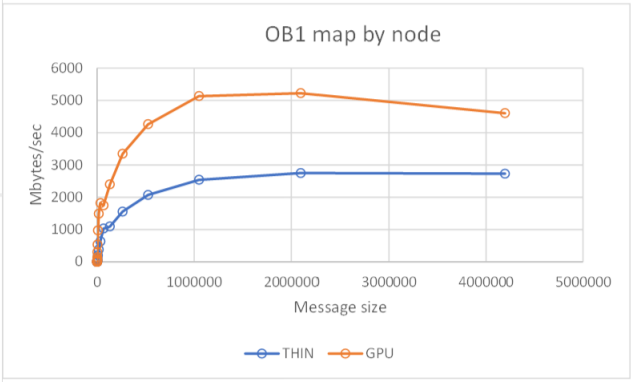
(j) INTEL contiguous processors

Figure 4: bandwidth estimation for each configuration

Some configurations have been tested also on the GPU in order to compare the performances:







3 Section 3

The last section is dedicated to Jacoby 3D solver, that is run with different configurations:

- On THIN and GPU node on one single core (serial execution)
- On THIN node with 4/8/12 processors within the same socket
- On THIN node with 4/8/12 processors across two sockets
- On two THIN nodes with 12/24/48 processors
- On GPU node with 12/24/48 processors

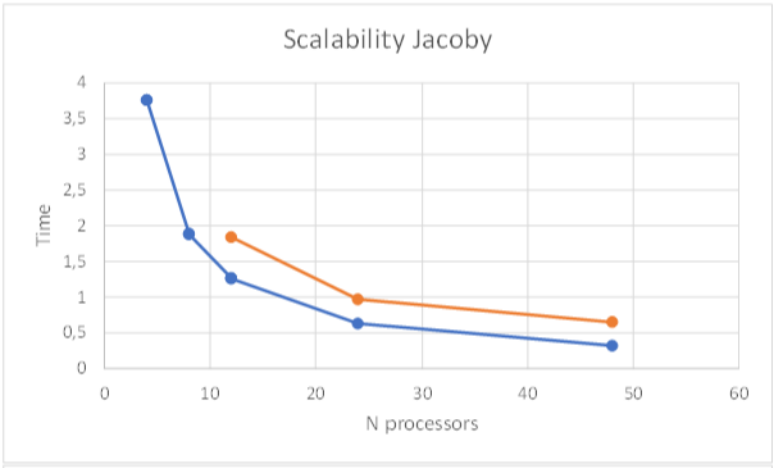
For each configuration, the right latency and bandwidth are taken from the table of the previous section.
The following table reports the results:

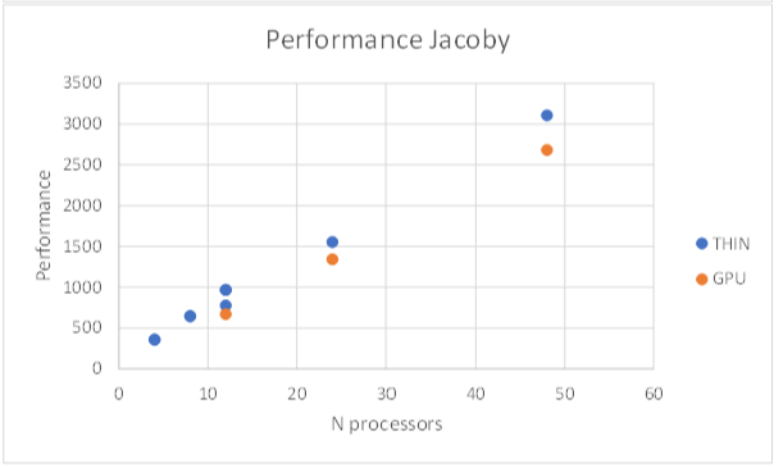
N	Nx	Ny	Nz	k	C(L,N)	Tc(L,N)	Latency	Bandwidth	Tmeasured(s)	P(L, N)	THIN/ GPU	Details
1	1	1	1	0					15,05		T	serial execution
1	1	1	1	0					21,73		G	serial execution
4	2	2	1	4	92160	4,25	0,20	21,69	3,76	358,14	T	map by core
4	2	2	1	4	92160	4,28	0,41	21,56	3,76	357,65	T	map by socket
8	2	2	2	6	138240	6,37	0,20	21,69	1,89	645,24	T	map by core
8	2	2	2	6	138240	6,41	0,41	21,56	1,88	644,05	T	map by socket
12	3	2	2	6	138240	6,37	0,20	21,69	1,27	967,86	T	map by core
12	3	2	2	6	138240	6,41	0,41	21,56	1,26	966,07	T	map by socket
12	3	2	2	6	138240	11,66	0,99	11,86	1,26	776,28	T	map by node
24	4	3	2	6	138240	11,66	0,99	11,86	0,63	1552,56	T	map by node
48	4	4	3	6	138240	11,66	0,99	11,86	0,32	3105,13	T	map by node
12	3	2	2	6	138240	9,22	0,20	15,00	1,84	670,04	G	hyperthreading
24	4	3	2	6	138240	9,22	0,20	15,00	0,97	1340,09	G	hyperthreading
48	4	4	3	6	138240	9,22	0,20	15,00	0,65	2680,18	G	hyperthreading

where:

- $C(L,N) = L^2 * k * 2 * 8$
- $T_C(L,N) = \frac{C(L,N)}{Bandwidth} + k * Latency$
- $T_{measured}$ is obtained by averaging Jacoby mintime and maxtime from the output of the program
- $P(L,N) = \frac{L^3 * N}{T_{serial}(L) + T_C(L,N)}$

It is possible to see the trend of the performance calculated above ($P(L,N)$) and the measured time with respect to the number of processors. It can be noticed that there is a linear dependence between number of processors and calculated performance, but the measured time doesn't behave in the same way.
Also, it is evident that GPU with hyperthreading enabled doesn't improve performances and takes more time for the execution of the program.


Scalability Jacoby
Time
4
3,5
3
2,5
2
1,5
1
0,5
0
0
10
20
30
40
50
60
N processors


Performance Jacoby
Performance
3500
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
N processors
THIN
GPU