Matteo Scorcia

University of Trieste, Foundations of High-Performance Computing

All the code can be found at: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1

ASSIGNMENT 1

SECTION 1

Implementation of ring.c and matrix_addition.c

Exercise 1 – Ring

Discussion about performance of ring.c

Exercise 2 – Matrix Addition

Discussion about performance about matrix_addition.c

SECTION 2

Discussion about all the Ping-Pong performed.

SECTION 3

Discussion about the Jacobi solver instances that has been run.

SECTION 1

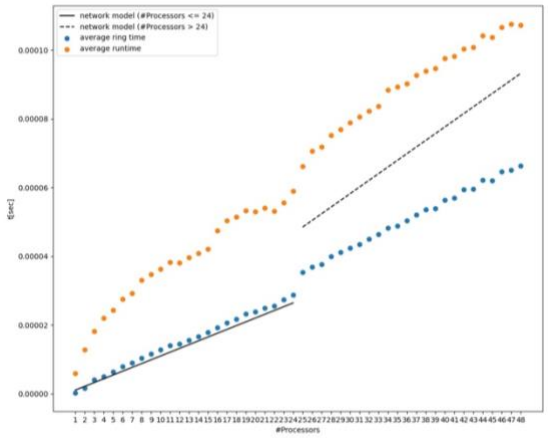
Exercise 1 – Ring

All the code: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section1/exercise1

The average results are computed by taking the average of 100000 runs of ring.c

Plot Runtime, Ring Time, Network Performance Model vs #Processors

$$T_{runtime} = T_{serial} + T_{parallel} + T_{\text{comm}}$$



- Average Runtime: time taken from start to finish of the ring.c program
- Average ring time: time take from the end of the creation of the virtual topology, until the end of the last iteration of send/receive.
  Ring time is composed by the following operations:
  - Send a message to the left neighbor
  - Send a message to the right neighbor
  - Receive a message from the right neighbor
  - Receive a message from the left neighbor

We can clearly see that there is a "bump" when we are increasing the processors to 25, this is since we are pinning the 25th process to the second THIN node, because the first THIN node is full.

Sadly, it seems that the model isn't so accurate when #Processors > 24.

There is a distance between the model and the "true fit" roughly of 0.20 us.

The λ used in the model is 0.99 us and the "true" λ that could fit perfectly the data is 0.73.

Discussion on network performance model

The general network model is reported below:

$$\mathrm{T_{comm}} = \left[\lambda_{\mathrm{network}} + \frac{message - size}{b_{network}}\right] \cdot 2 \cdot Number\ of\ Processors$$

Where T_{comm} is the time needed for the communication until all the processes have finished to send/receive messages.

Where message-size is the size of the message expressed in Bytes.

Where $b_{network}$ is the bandwidth expressed in [B/sec].

Where λ is the latency of the network.

In every iteration of the ring.c program, we are sending/receiving 2 messages of size 4 Bytes (one integer).

This means that for each iteration, the dominant latency is given by the network latency λ.

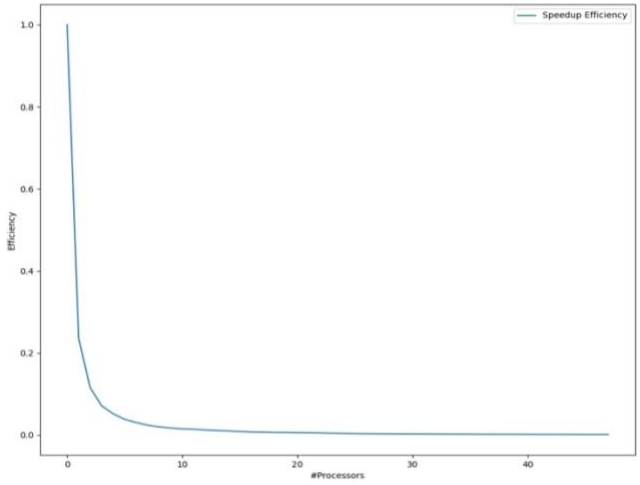
One process could send a message intrasocket, intranode, internode, we need to change our model in correspondence of the larger λ that can occur in each iteration.

This means that we have 2 models:

$$T_{\mathrm{comm}} = \left[\lambda_{socket} + \frac{message-size}{b_{vader,socket}}\right] \cdot 2 \cdot Number\ of\ Processors\ (if\ \#Processors \leq 24)$$

$$T_{\mathrm{comm}} = \left[\lambda_{node} + \frac{message - size}{b_{ucx,node}}\right] \cdot 2 \cdot Number\ of\ Processors\ (if\ \#Processors > 24)$$

Discussion on Scalability



As we can see from the plot, we have an inexistent speedup efficiency.

This is what we are expecting because our problem is growing as our number of processes increase!

There is no parallelization of the workload, we are in the worst-case scenario.

$$T_{runtime} = T_{serial} + T_{parallel} + T_{\text{comm}}$$

T_{serial} = average ring time (#Processors = 1)

$T_{parallel}$ = 0

$T_{\text{comm}} \propto number\ of\ Processors$

The dominant time is the communication time T_{comm}!

There is no scalability, as we increase the number of processes, our time of execution increase proportional to the number of processes!

Exercise 2 – Matrix Addition

All the code: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section1/exercise2

Discuss performance for the three domains in term of 1D/2D/3D distribution keeping the number of processor constant at 24

As we can see from the table of results:

- Runtime -> Time taken from the start of the program until the end of the matrix_addition.c program.
- Matrix Time -> Time taken from the end of the creation of the random matrix until the end of the gathering of the results from all the processes.
  Matrix Time is composed by the following operations:
  - Scattering of matrix A and matrix B
  - Addition of chunk of matrices
  - Gathering of results into matrix C

All the different topologies have the same behavior given the fact that we are using only collective operations, as we can see from the table, the times collected are pretty much the same except for some fluctuations in “Runtime” (caused by the matrix filling by random numbers and by the creation of the virtual topology).

Model Network Performance and Discussion on Scalability

$$T_{runtime} = T_{serial} + T_{parallel} + T_{\text{comm}}$$

$$message - size \propto \frac{1}{\#Processors}$$

$$T_{\text{comm}} = \left[\lambda_{network} + \frac{message - size}{b_{network}}\right]$$

There isn't a section that can't be parallelized in the execution of the code.

$$T_{serial} = 0$$

This is an embarrassingly parallelizable problem.

Speedup(p) = p

Efficiency Speedup(p) = 1

The only limit to parallelization here is given by the communication overhead.

$$lim_{p->\infty}\, T_{runtime} = T_{\text{comm}} = \lambda$$

SECTION 2

Introduction

In this section we discuss in detail the performance obtained in the Ping-Pong runs.

Note that by "Infiniband network" I mean the "High Speed network" and "I/O network" of ORFEO and by "Ethernet network" I mean the "In Band Management network" of ORFEO.

All the results reported have been fetched using 2 THIN nodes.

All the results reported are means of 5 runs of the same Ping-Pong command.

All the plots reported have log_2 for the "#bytes" axes.

All the plots reported have log_{10} for the "t[usec]" axes.

All the runs have been performed requiring only 2 processes, this means that there were always a lot of traffic intranode and internode.

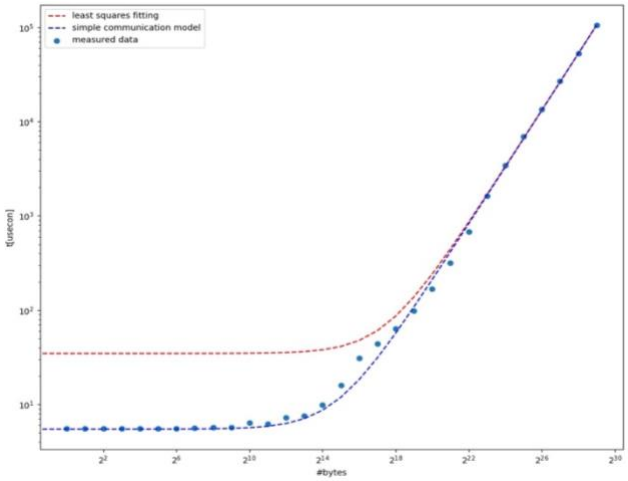
This mean that performing the same runs in another moment implies that the values may differ.

Tabular/Csv results: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section2/thin/results/

Plot results: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section2/thin/plots

Ping-Pong between 2 processes mapped by core

Openmpi implementation – ob1, tcp



ob1 is the point-to-point messaging layer module specified in the Openmpi implementation, while tcp is the byte transfer layer module.

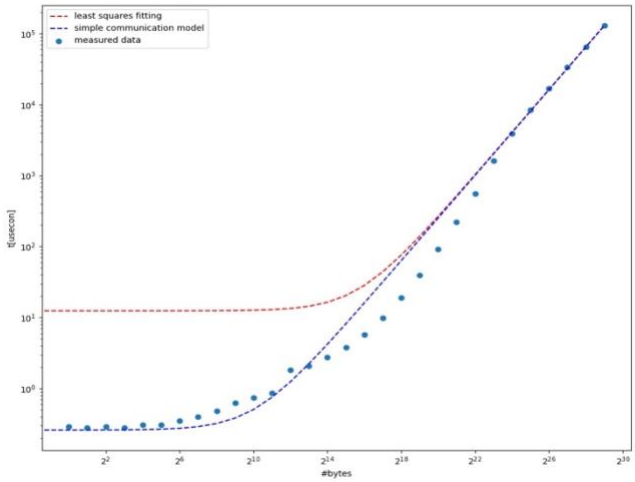
Use ob1, with tcp means that we are specifying to send our messages through the ethernet network card.

As we can see from the data table, the asymptotic bandwidth is roughly 5100 MB/s, the bandwidth computed by fitting is 5090 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 5.46 us

Openmpi implementation – ob1, vader



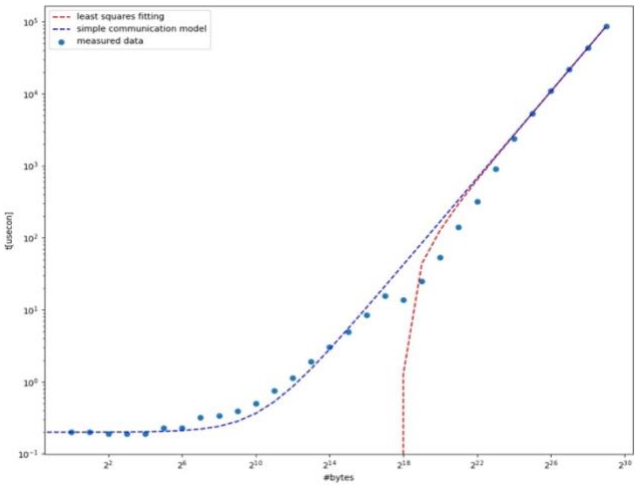
vader is another byte transfer layer module that allows to send the message in a “shared memory approach” between processes, so we are adding a virtual layer to always treat the communication between 2 processes as the communication between 2 processes in a uniform memory access region.

As we can see from the data table, the asymptotic bandwidth is roughly 4150 MB/s, the bandwidth computed by fitting is 4130 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.26 us

Openmpi implementation - ucx



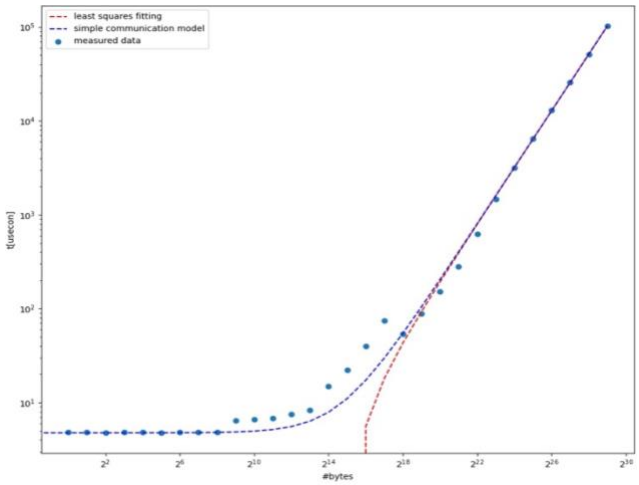
ucx is another point-to-point messaging layer module that allows to send messages through the infiniband network card.

As we can see from the data table, the asymptotic bandwidth is roughly 6220 MB/s, the bandwidth computed by fitting is 6200 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.2 us

Intel Implementation – tcp (contender: Openmpi implementation – ob1, tcp)



As we can see from the data table, the asymptotic bandwidth is roughly 5240 MB/s, the bandwidth computed by fitting is 5240 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

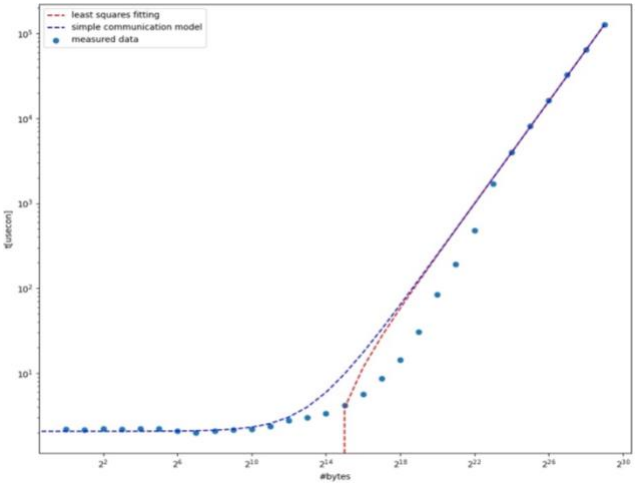
Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 4.77 us

Respect to the Openmpi implementation, we gain roughly 100MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency dropped by 13%!

Intel Implementation – shm (contender: Openmpi implementation – ob1, vader)



shm is an ofi provider that ca be specified in the Intel implementation to run the Ping-Pong through a shared memory approach between processes.

As we can see from the data table, the asymptotic bandwidth is roughly 4240 MB/s, the bandwidth computed by fitting is 4220 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

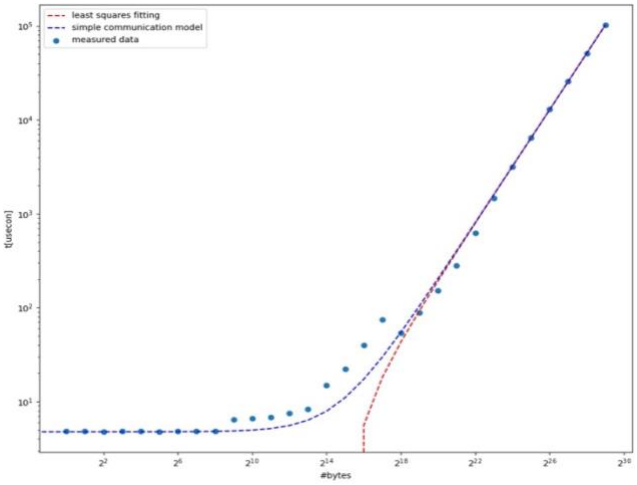
Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 2.07 us

Respect to the Openmpi implementation, we gain roughly 100MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency increased by 696%!

Intel Implementation – mlx (contender: Openmpi implementation – ucx)



mlx is an ofi provider that can be specified in the Intel Implementation to run the Ping-Pong through the underlying ucx layer

As we can see from the data table, the asymptotic bandwidth is roughly 6290 MB/s, the bandwidth computed by fitting is 6280 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.25 us

Respect to the Openmpi implementation, we gain roughly 70MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency increased by 25%!

Observations – map by core

The Intel implementation is clearly the winner in terms of bandwidth respect to openmpi.

But why do we reach these low bandwidths and high latencies between 2 cores within the same socket in all the implementations respect to the "map by node" implementations?

My guess is that when we specify to send a message, there is no control over the destination done by some unit inside the cpu.

This means that the message must exit the cpu, reach the proper network card (infiniband card or ethernet card) through the PCI express channel (route expressed by the "self" command in the "--mca btl" specification).

Only then, return to the cpu in the right core pinned by the process designed to receive the message, hence the low result numbers.

During this journey, our message can be queued and rerouted, we don't know the exact route that our message takes (the node components need to communicate with the operative system above creating traffic).

It could be a route slower than the route taken when the message needs to go outside the node.

Everything I said is just my speculation, I found this website that states that:

"For almost all recent systems, a single thread of execution can only generate a fraction of the bandwidth available within or between sockets"

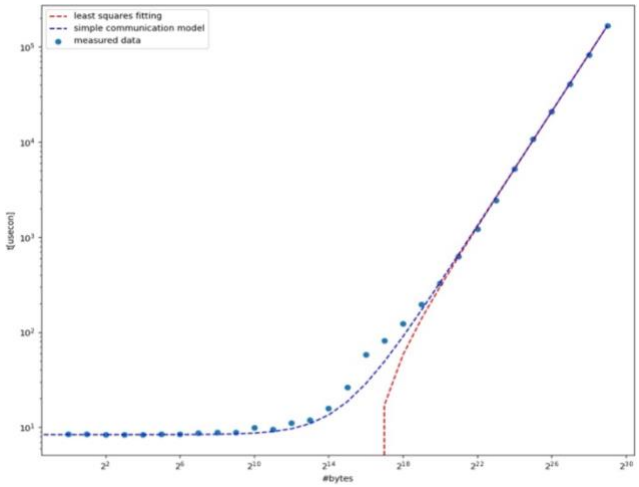
There isn't a proof of this statement, so I just copy pasted it here for reference.

This claim has credibility just because the person who wrote it is a Research Scientist in HPC Performance.

If this statement is true, then all my speculations are wrong in some extent.

Ping-Pong between 2 processes mapped by socket

Openmpi implementation – ob1, tcp

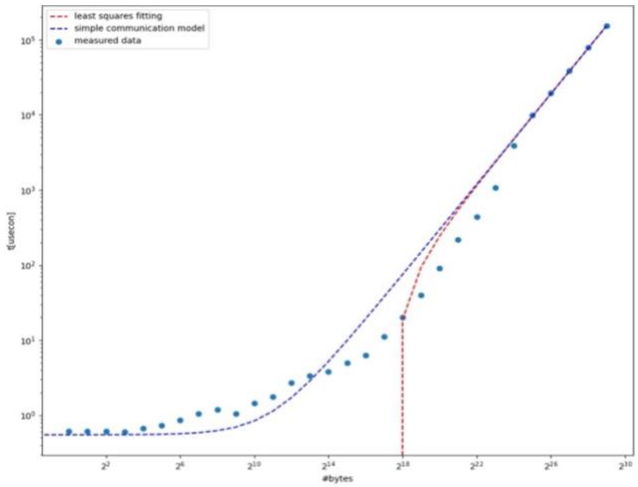


As we can see from the data table, the asymptotic bandwidth is roughly 3230 MB/s, the bandwidth computed by fitting is 3240 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 8.38 us

Openmpi implementation – ob1, vader

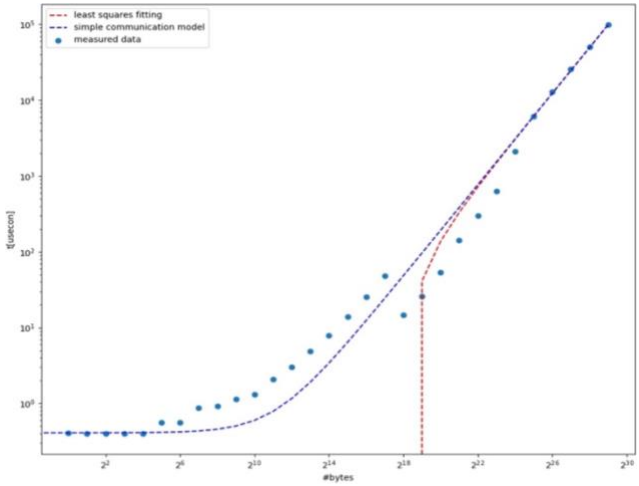


As we can see from the data table, the asymptotic bandwidth is roughly 3520 MB/s, the bandwidth computed by fitting is 3500 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.55 us

Openmpi implementation - ucx

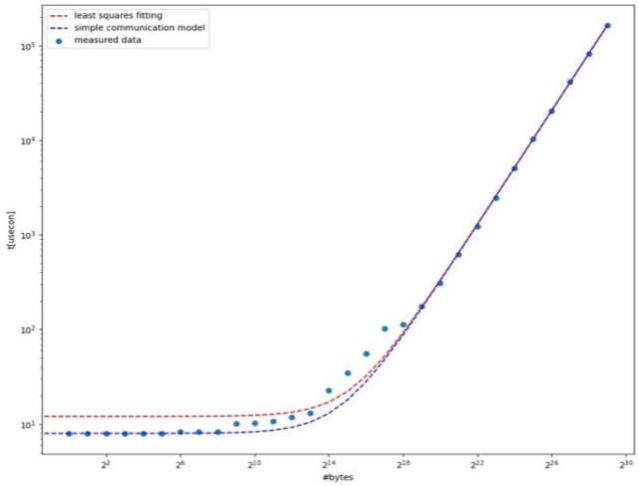


As we can see from the data table, the asymptotic bandwidth is roughly 5460 MB/s, the bandwidth computed by fitting is 5420 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.41 us

Intel Implementation – tcp (contender: Openmpi implementation – ob1, tcp)



As we can see from the data table, the asymptotic bandwidth is roughly 3280 MB/s, the bandwidth computed by fitting is 3280 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

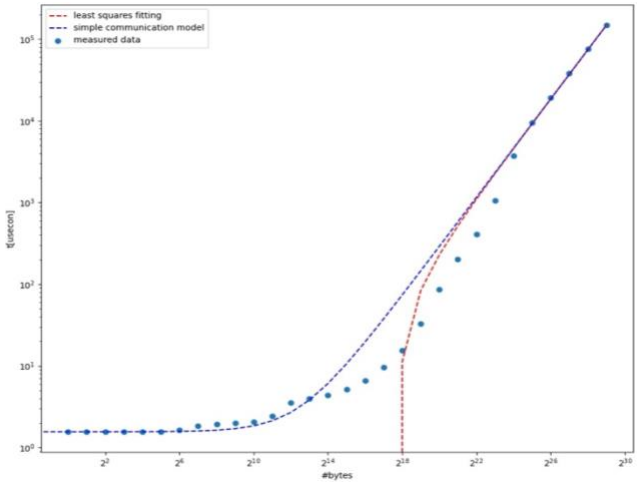
Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 7.97 us

Respect to the Openmpi implementation, we gain roughly 50MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency dropped by 5%!

Intel Implementation – shm (contender: Openmpi implementation – ob1, vader)



As we can see from the data table, the asymptotic bandwidth is roughly 3620 MB/s, the bandwidth computed by fitting is 3610 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

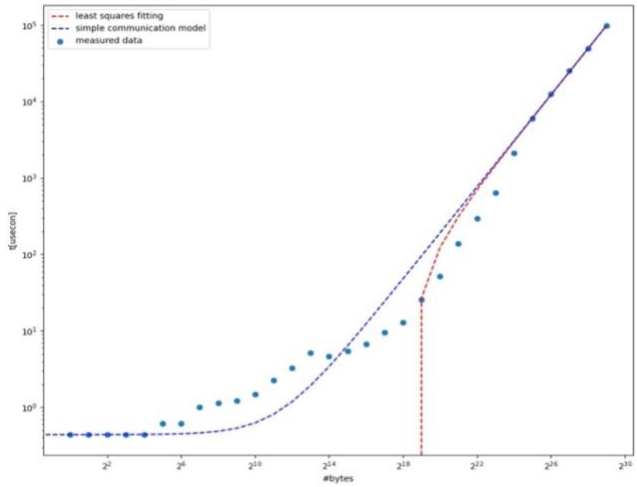
Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 1.56 us

Respect to the Openmpi implementation, we gain roughly 100MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency increased by 182%!

Intel Implementation – mlx (contender: Openmpi implementation – ucx)



As we can see from the data table, the asymptotic bandwidth is roughly 5510 MB/s, the bandwidth computed by fitting is 5490 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.44 us

Respect to the Openmpi implementation, we gain roughly 50MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency dropped by 7%!

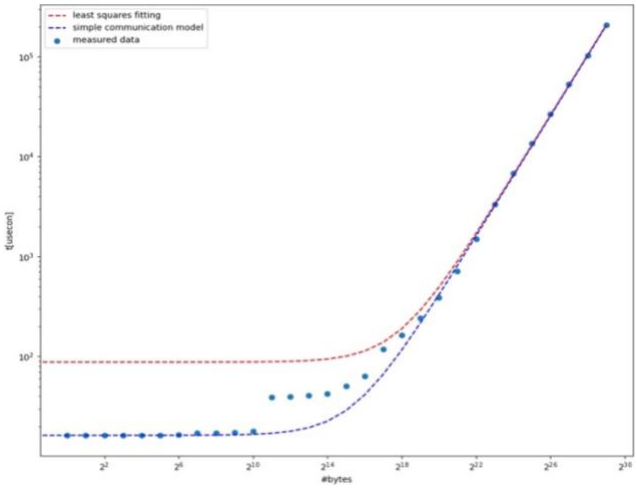
Observations – map by socket

Everything said in "Observation – map by core" holds, except for the fact that now we have a greater distance between the sender process and the receiver process.

This explains the higher latency and lower bandwidth in all the implementations.

Ping-Pong between 2 processes mapped by node

Openmpi implementation – ob1, tcp



As we can see from the data table, the asymptotic bandwidth is roughly 2630 MB/s, the bandwidth computed by fitting is 2590 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

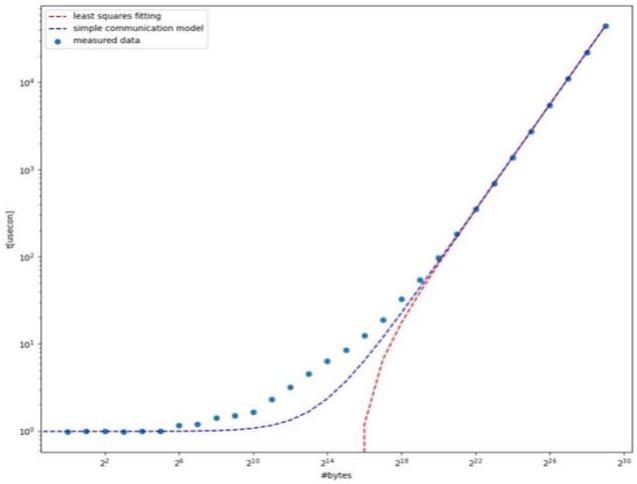
Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 16.25 us

Openmpi implementation – ob1, vader

You can't specify a memory shared approach mapping the processes by node.

Openmpi implementation - ucx

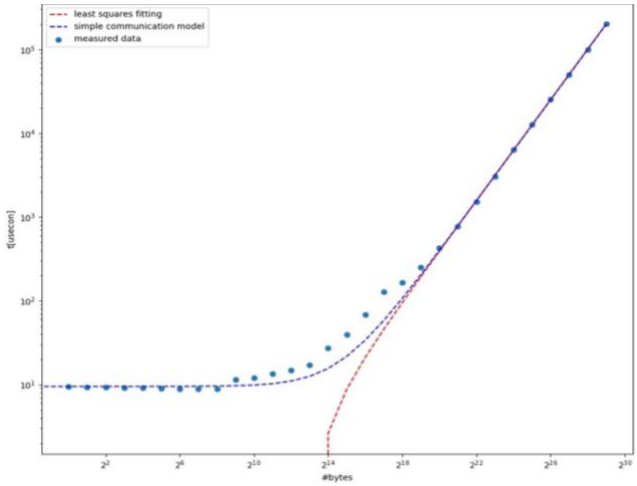


As we can see from the data table, the asymptotic bandwidth is roughly 12060 MB/s, the bandwidth computed by fitting is 12090 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 0.99 us

Intel Implementation – tcp (contender: Openmpi implementation – ob1, tcp)



As we can see from the data table, the asymptotic bandwidth is roughly 2690 MB/s, the bandwidth computed by fitting is 2660 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 9.51 us

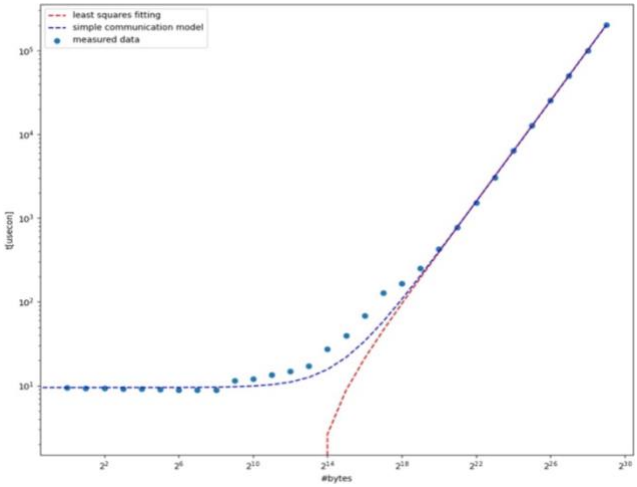
Respect to the Openmpi implementation, we gain roughly 60 MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency dropped by 41%!

Intel Implementation – shm (contender: Openmpi implementation – ob1, vader)

You can't specify a memory shared approach mapping the processes by node.

Intel Implementation – mlx (contender: Openmpi implementation – ucx)



As we can see from the data table, the asymptotic bandwidth is roughly 12180 MB/s, the bandwidth computed by fitting is 12180 MB/s, so we have a good asymptotic approximation of the model computed by fitting.

Same applies to the asymptotic latency.

The latency with a message size of 0 bit is 1.21 us

Respect to the Openmpi implementation, we gain roughly 120 MB/s of bandwidth!

Respect to the Openmpi implementation, the initial latency increased by 1%!

Observations – map by node

We know that the theoretical bandwidth for the ethernet network is 25 Gb/s on ORFEO.

Given the asymptotic bandwidth of the openmpi implementation:

2630 MB/s = 21040 Mb/s = 21.04 Gb/s

➔ We are reaching 84% of the theoretical bandwidth, not so good

Given the asymptotic bandwidth of the intel implementation:

2690 MB/s = 21520 Mb/s = 21.52 Gb/s

➔ We are reaching 86% of the theoretical bandwidth, we aren't exploiting the network resources so well even with the intel implementation.

We know that the theoretical bandwidth for the infiniband network is 100 Gb/s on ORFEO.

Given the asymptotic bandwidth of the openmpi implementation:

12060 MB/s = 96480 Mb/s = 96,48 Gb/s

➔ We are reaching 96% of the theoretical bandwidth, we are exploiting well the network resources!

Given the asymptotic bandwidth of the intel implementation:

12180 MB/s = 97440 Mb/s = 97,44 Gb/s

➔ We are reaching 97% of the theoretical bandwidth, we are exploiting well the network resources!

THIN nodes vs GPU nodes

All the measurements were also made on GPU nodes.

From the results we can see that the bandwidths are worse by a range of

[200 MB/s, 500 MB/s].

The only measure that has not worsened is the "map by node, ucx implementation" (the one that uses the infiniband network).

My guess for the worsening is that during the measurements, there were a lot of traffic on the GPU nodes, I was sharing all the resources with a lot of users.

This led to a lot of traffic that could have worsen the measures.

Maybe for the time being, the only implementation able to keep up with the internode traffic was the ucx implementation that uses the infiniband card.

It could also be that we have better bandwidths on the THIN nodes because we have better hardware except for the infiniband card, or maybe both the observations hold true, or maybe just the latter.

Final Observations

In all the cases this statement remains:

Given the shape of the sample communication model, we can see that the fitting model can't reach a good behavior in the left-hand side part of the graph.

This is because we are using a linear square method to fit a noisy dataset with outliers.

It's worth to notice that even if in all "intel implementations" the bandwidth increases, in the shared memory implementation, the latency is worse then "openmpi implementation".

The only case where the shared memory approach is faster and with better bandwidth, is when we don't have access to an ucx implementation, and we are pinning the processes by socket.

If we are interested in just the bandwidth and we do not have access to an ucx implementation and we are pinning the processes by core, the better choice is the tcp implementation.

However, this use case it isn't so interesting in the ORFEO cluster because most of the times, we are using multiple sockets or multiple nodes, and if we are just using cores of the same CPU, it is worth considering a transition of the logic to an OpenMP implementation instead.

Notice that given a mapping ("by core", "by socket", "by node"), the implementations that use the infiniband network card ("ucx" and "mlx"), are always the fastest and with more bandwidth compared to tcp ("tcp") and shared memory ("vader", "shm") implementations!

SECTION 3

Introduction

For all the runs the input to the Jacobi solver is always the same:

L=1200, boundary conditions.

The maximum iterations that the program can do to reach convergence in a single run is 10.

Csv results: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section3/csv

Plot results: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assigment1/section3/plots

For the convenience of the reader, here are reported the formulas:

$$\mathrm{P(L,N)} = \frac{L^3 \cdot N}{T_s + T_c(L,N)}$$

$$\mathrm{T_c(L,N)} = \frac{c(L,N)}{B} + k \cdot T_l$$

$$c(L,N) = L^2 \cdot k \cdot 2 \cdot 8$$

Jacobi Solver on a Single Processor

THIN node:

$T_s \approx 15.24$ s

$$Total\ time \approx 10 * T_s = 152.4\ s$$

GPU node:

$T_s \approx 22.07$ s

$$Total\ time \approx 10 * T_s = 220.7\ s$$

Jacobi Solver on 4/8/12 Processors using 1 THIN node

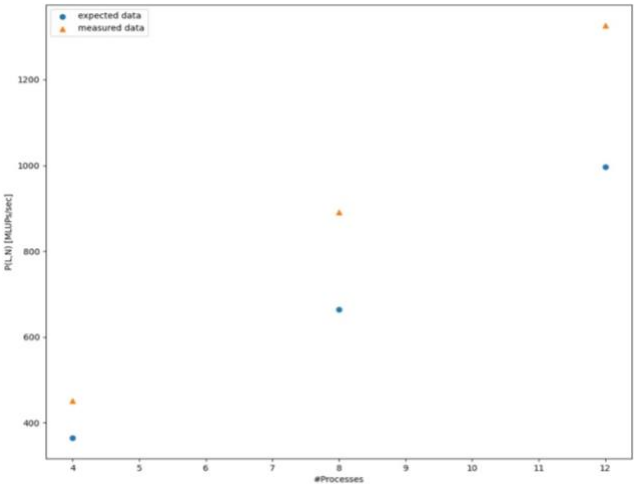
Map by core (Openmpi – map by core, ucx implementation):

All the processes are pinned in the same socket.

$$T_l = 0.2\ us$$

$$\frac{B}{2} = 6220\ MB/s$$

measured Performance vs expected Performance:

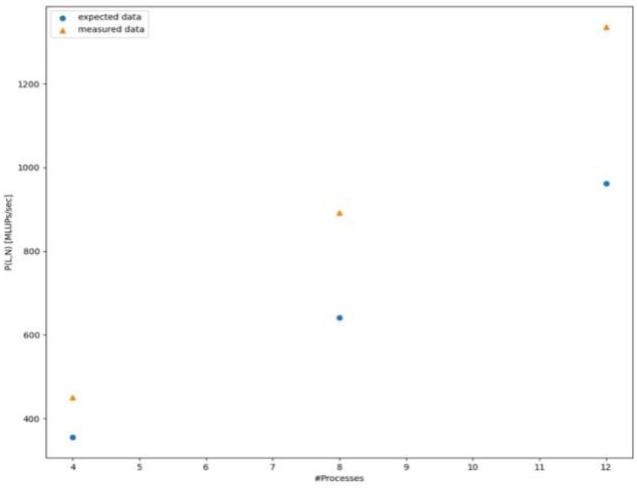


Map by socket (Openmpi – map by socket, ucx implementation):
Half of the processes are pinned in a socket; half of the processes are pinned in another socket.

$$T_l = 0.41\ us$$

$$\frac{B}{2} = 5460\ MB/s$$

measured Performance vs expected Performance:



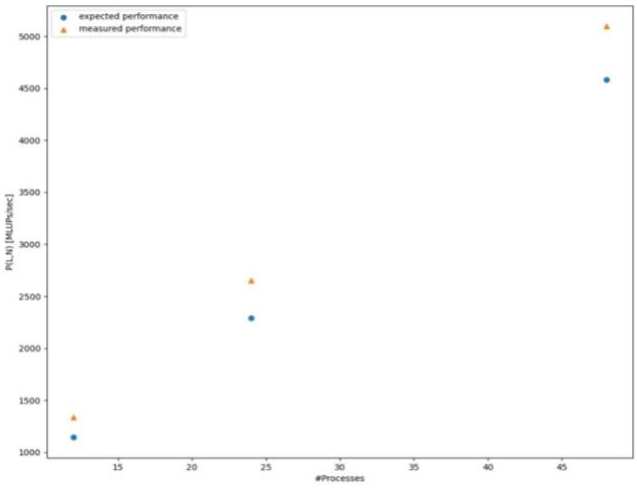
Jacobi Solver on 12/24/48 Processors using 2 THIN nodes

Jacoby Solver on 12/24/48 Processors (Openmpi – map by node, ucx implementation):

$$T_l = 0.99\ us$$

$$\frac{B}{2} = 12060\ MB/s$$

measured Performance vs expected Performance:



Jacobi Solver on 12/24/48 Processors using 1 GPU node

Remember that hyperthreading is enabled on GPU node, we can run Jacobi Solver by pinning 48 processes all on a single GPU node.

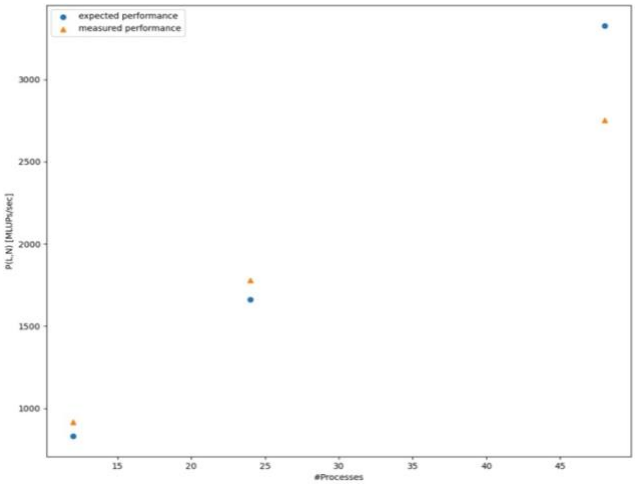
Let's assume that we have 48 processes available on our GPU node (we don't consider the hyperthreading virtualization).

Jacoby Solver on 12/24/48 Processors (Openmpi – map by node, ucx implementation):

$$T_l = 1.39\ us$$

$$\frac{B}{2} = 12020\ MB/s$$

measured Performance vs expected Performance:



Final Observations

In every estimation of the performance, we are underestimating, sometimes by 80 MLUPs/sec, sometimes by 300 MLUPs/sec.

First, we are estimating the bandwidth full duplex by simply doubling the bandwidth found in SECTION 2, and this can be a rough approximation.

Second, we are assuming that we have the same bandwidth for each gathering process of each process of each iteration of the Jacobi Solver.

This is the case when we pin 1 MPI process to 1 NODE, or 1 MPI process to 1 SOCKET, but in our experiments, most of the cases we have "hybrid" pinning of the MPI processes.

Third, as stated here: "for almost all recent systems, a single thread of execution can only generate a fraction of the bandwidth available within or between sockets".

This means that we are always underestimating the available bandwidth in SECTION 2.

Another important Observation

Jacobi Solver on 1 GPU node, 48 processors vs Jacobi Solver on 1 GPU node, 24 processors:

This is the only case where we aren't underestimating the performance, we are overestimating (by 570 MLUPs/sec!!).

This is good news, compared to the "Jacobi Solver on 1 GPU node, 24 processors", we are gaining in performance (even if the gain is less than the expected)!

We can see that this performance is comparable to the performance obtained in "Jacobi Solver on 2 THIN nodes, 24 processors".

The hyperthreading can't do miracles because there isn't too much space to exploit the scheduling of the elementary tasks queued in the cores (because all the physical cores are already in use!).

If we have 24 physical cores, and we demand 48 "threads" by hyperthreading, we can't expect to have the same performance of a demand of 48 "threads" on 48 physical cores.

Jacobi Solver on. GPU node, 24 processors vs Jacobi Solver on 2 THIN nodes, 24 processors:

In this case, we can't be sure if we are using hyperthreading or just using physical cores.

Whatever the case may be, comparing the performance with "Jacobi Solver on 2 THIN nodes, 24 processors" leads to a worsening in performance by a factor of 0.69:

1780 MLUPs/sec vs 2650 MLUPs/sec

My guess is that given the two sweep times per iterations:

$T_s \approx 15.24$ s (THIN node)

Vs.

$T_s \approx 22.07$ s (GPU node)

The ration between the 2 sweep times is 0.67!

The raw compute time T_s for all cell updates in a Jacobi sweep is the main worsening factor between the 2 runs.

The THIN node cpu behaves better than the GPU node cpu, given the same number of processes.