Foundations of HPC - Assignment 1

Alessandro Masini

December 30, 2021

Source files for all sections of this report can be found at the following link. Results for each section are contained in its dedicated folder. In each of these directories you can find an `upload` folder, where the data uploaded on Orfeo are stored, a `results` folder containing additional plots and tables, a `scripts` folder with all the scripts used to process data, and a few data folders containing raw data obtained from cluster jobs (folders starting with `data_`).

1 MPI programming

1.1 Processor Ring

In Figure 1 we can see how the execution time of the message ring changes depending on the number of processors. The first thing we may notice is that the behavior is slightly different for thin (blue) and gpu (orange) nodes. In the former case, the communication time increases in linear fashion, but there are points where the line slope changes abruptly. This points correspond to `#procs` = 12, 24, i.e. the size of the socket and the node respectively. We can explain this behavior by noticing that the communication time for a single processor is always the same, due to the size of the messages exchanged being fixed at one `int`, i.e 4 bytes. From the results of the Ping Pong benchmark in Section 2, we can see that the time needed to exchange 4 bytes is only determined by the latency of the network λ, thus the overall communication time for a ring with n_p processors can be (roughly) estimated as

$$t_c(n_p) = 2\lambda n_p \,, \tag{1}$$

where the factor 2 is due to each processor sending 2 messages (clockwise and counterclockwise on the ring). The slope changes are readily explained by a change in network latency when we move from processors being inside the same socket (`#nprocs` ≤ 12) to processors being in the same node ($13 \leq$`#nprocs` ≤ 24) and finally to processors being on two different nodes (`#nprocs` > 24).

For the gpu nodes the situation is similar, but there is an important difference: hyperthreading is enabled. This allows us to schedule up to 48 tasks on the same processor (with 24 physical cores). Due to this, we do not need two different nodes to run the ring with `#procs` > 24, thus in the graph we only see the slope change corresponding to the shift from having all processes on the same-socket to having all processes on the same-node, but on different sockets (i.e. the same as the first slope change for the thin node case).

Fitting the data obtained from the cluster jobs with the model in Eq. (1) we can obtain a rough estimate of the latencies for the different node-binding configurations. The results are shown in Table 1. As we can see, these results are a bit rough compared to the ones obtained from the PingPong benchmark, but we

`node`	`binding`	`latency[usec]`
thin	same socket	0.35
gpu	same socket	0.46
thin	same node	0.54
gpu	same node	0.70
thin	different nodes	1.20

Table 1: Estimated latencies for different node types and process-binding configurations.

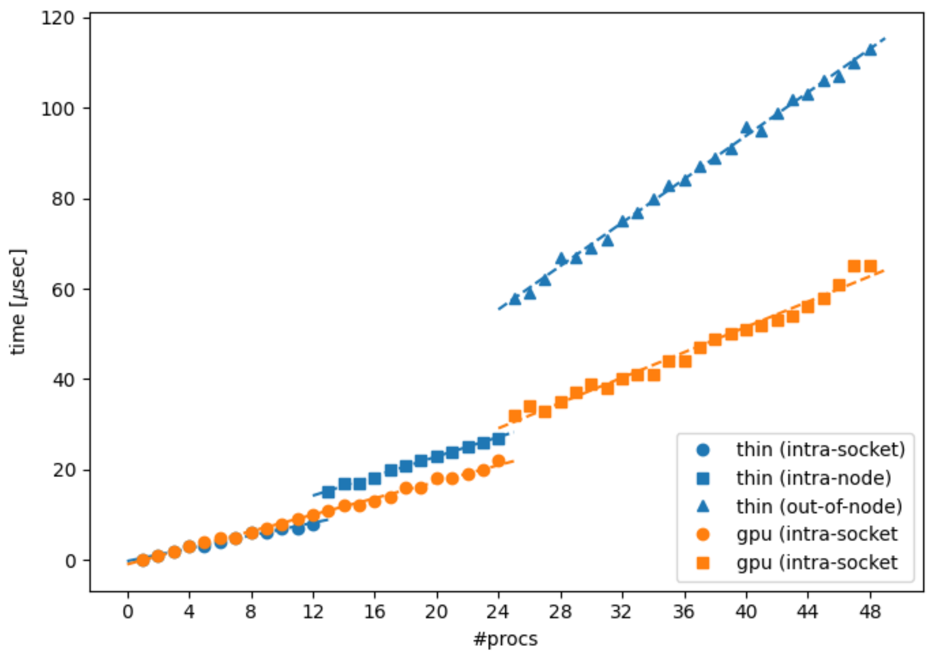

Figure 1: Communication time vs. process number for the ring

still recognize two interesting features. First, thin nodes have smaller latencies with respect to gpu nodes, second, as we may expect, the latency increases moving from same-core communications to same-socket communications and finally to communications between entirely different nodes.

1.2 3D Matrix Sum

Nx	Ny	Nz	800x300x100	1200x200x100	2400x100x100
24	1	1	0.1003	0.1004	0.0998
12	2	1	0.1002	0.1034	0.1029
8	3	1	0.1006	0.1008	0.1006
6	2	2	0.1003	0.1032	0.1005
6	4	1	0.1009	0.1000	0.1009
4	3	2	0.1006	0.1020	0.1007

Table 2: Communication time for 3D matrix sum with different virtual topologies.

In Table 2 we can see the communication times obtained from the `sum3Dmatrix.c` program (computed as an average over 10 runs) using different test virtual topologies. Since the sum can be performed without the need to communicate halo layers between different subdomains virtual topology is not exploited, thus resulting in all communication times being roughly the same.

It is possible to obtain an estimate of the theoretical bandwidth of the network, to be compared with the one obtained from the PingPong benchmark in Section 2. To do so we notice that the message exchange between processors is composed of three phases:

- root process scattering first matrix;
- root process scattering second matrix;
- each process sending back summed elements to the root.

In each of these three phases 23 messages are sent (excluding root process sending to itself), each containing $24000000/24 = 1000000$ `doubles` (i.e. 8 MB of data), for a total of $3 \times 23 \times 8\,\text{MB} = 552\,\text{MB}$ of data. Since

	`thin node`		`gpu node`	
`components`	`lat[usec]`	`bw[GB/s]`	`lat[usec]`	`bw[GB/s]`
`------------------ map-by core -------------------`				
ucx	0.197	6.4	0.217	6.4
intel	0.232	5.5	0.289	5.2
vader	0.269	4.5	0.279	4.4
tcp	5.48	5.3	5.0	5.1
`----------------- map-by socket -----------------`				
ucx	0.397	5.6	0.412	5.6
intel	0.43	4.2	0.475	4.3
vader	0.598	3.9	0.564	3.7
tcp	7.95	3.2	8.62	3.1
`------------------ map-by node -------------------`				
ucx	0.983	12.0	1.33	12.0
intel	1.05	12.0	1.33	12.0
tcp	16.4	2.6	15.4	2.2

Table 3: Results of PingPong benchmark from the Intel MPI benchmarks suite. The table shows estimated latencies and (asymptotic) bandwidths for different network configurations, obtained by fitting the data from the benchmarks (averaged over 10 runs) to the linear network model discussed during the lectures.

all the communication times we obtained are close to 0.1 s, the resulting bandwidths will be of the order of 5520 MB/s $\approx$ 5.5 GB/s, which is a good estimate for intra-node asymptotic bandwidth obtained using `ucx` as `pml` for `openmpi` (see Table 3, `map-by socket` section).

2 Point to point performance measurement

Table 3 shows network latencies and asymptotic bandwidths obtained by fitting the results of the PingPong benchmark (averaged over 10 runs) with the linear model discussed in class. As we can see, the latencies for thin nodes are generally lower than the corresponding values of the gpu nodes, while the bandwidths are higher. This may be due to the different CPU frequencies of the two kinds of nodes, with the thin nodes having higher frequency than the gpu nodes. For a detailed representation of the bandwidth profile we refer to Figure 2. This picture shows bandwidth behavior when using different or different components for the same MPI library (`openmpi` with `ucx` or `ob1` and `vader` or `tcp`), or an entirely different library (`intel`).The results presented are for thin nodes, but for gpu nodes the situation is the same. The solid lines represent the bandwidths computed as N_{bytes}/T_c, where T_c is the communication time computed using the linear model

$$T_c = \lambda + \frac{N_{\text{bytes}}}{B} \tag{2}$$

with λ and B obtained from a fit of the real data outputted by the benchmark.

We start our discussion with intranode communications, i.e. core and socket cases. As we can see from Figure 2, the bandwidth profile initially increases, independently from the specific type of process binding (core, socket, node), or MPI components used. This is simply due to the communication overhead becoming less and less important with respect to the total communication time. Focusing on the `ucx` case, we immediately notice that for message sizes up to approximately 128 kB the bandwidth increases slowly, with the core case having higher bandwidth value than the socket case. This is probably due to the processes sharing the L3 cache in the former case, in contrast to the latter where they only share off-chip memory, which has higher access times. After 128 kB the situation changes, and we see almost the same bandwidth in both cases, with a jump to valuse as high as 20 GB/s. This may be due to a change of the message transfer

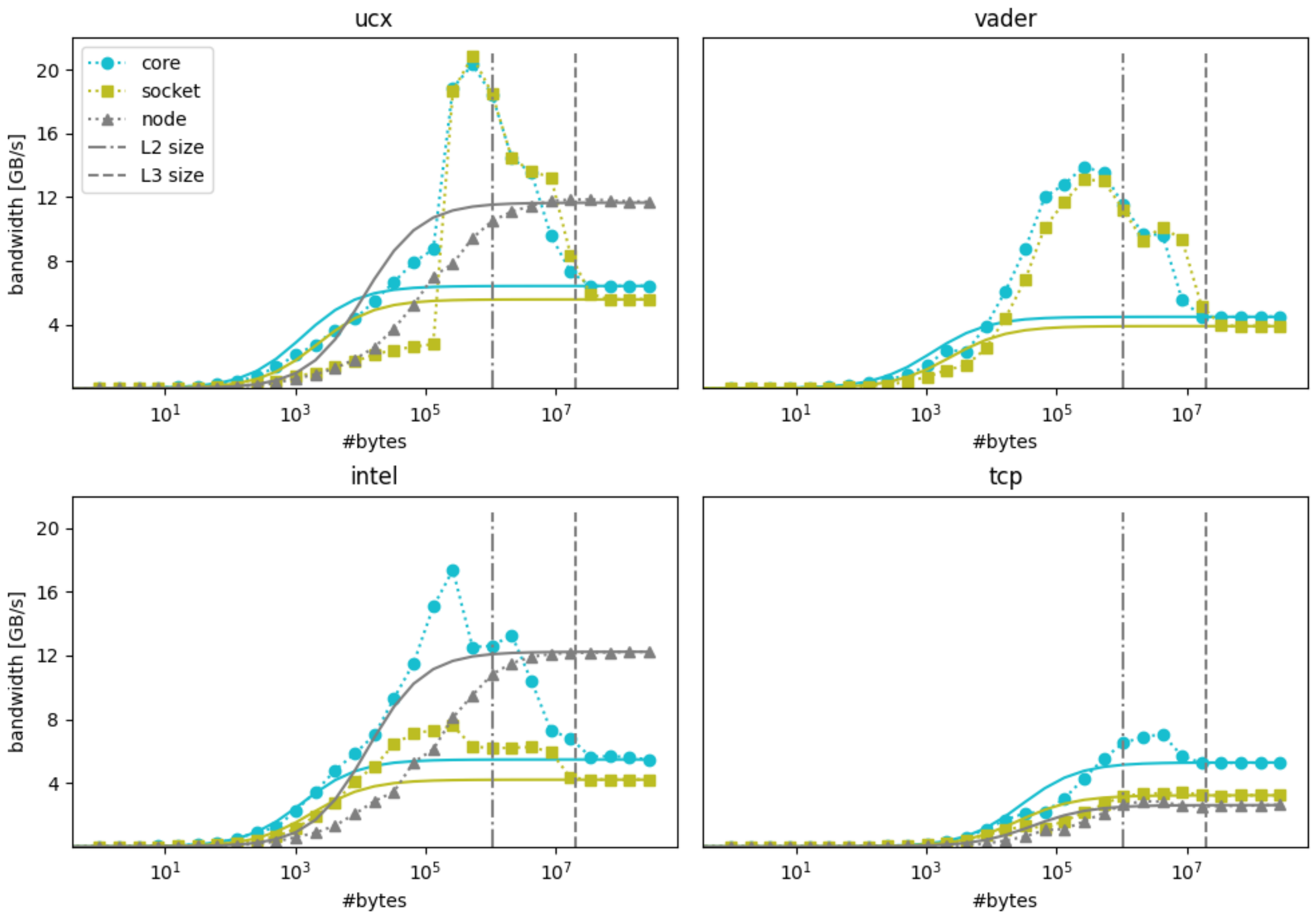

Figure 2: Bandwidth profiles obtained from the Ping Pong benchmark using different MPI components.

protocol used by MPI, possibly shifting from a double-copy strategy (*eager protocol* or *copy-in/copy-out* protocol) to a single direct memory copy with message pipelining. The advantage of this method is absolute up to message sizes comparable with the size of the L2 cache (1 MB). This is due to the buffers always being contained in the L2 cache, providing fast access to data. For message sizes larger than 1 MB we have a slight decrease in performance, due to the message being too large to fit entirely in L2, thus leading to cache misses. After this the performance slowly decreases until we approach the L3 cache size of 19MB, after which the bandwidth collapses to its asymptotic value determined by the memory access efficiency ($5 - 6$ GB/s).

The situation is similar for the `vader` `btl` (which uses `ob1` as `pml`), with two main differences: the first is that there is no bandwidth jump as for `ucx` at 128 kB, instead we see a faster increase starting from a few kilobytes. The second difference is that the socket and core bandwidths are always close to each other, even for small message sizes. This is probably due to `vader` using different messaging protocols from `ucx`, which do not rely on the shared L3 cache for making the communication faster.

The opposite consideration applies to the `intel` case: here map-by socket and map-by core bandwidths are always well distinct. Furthermore, the descent from the maximum value of bandwidth to the asymptotic value is less smooth, and there is even a point where it grows again. This last feature may be due to the `intel` messaging protocol using buffer reuse to reduce cache misses.

Moving on to the `tcp` case, we immediately see that the bandwidth doesn't grow as much as in the other cases. For small message sizes this is due to the `tcp` protocol having a much higher latency with respect to the other cases considered, thus resulting in a slower initial growth. Nonetheless, at least for the core case, the bandwidth grows to values compatible with the shared memory case (5.5 GB/s). For the socket case, on the other hand, the asymptotic value is a bit lower, and it is closer to the bandwidth reached in the map-by-node case, which is determined by the speed of the Ethernet connection (25 Gb/s $\approx$ 3 GB/s).

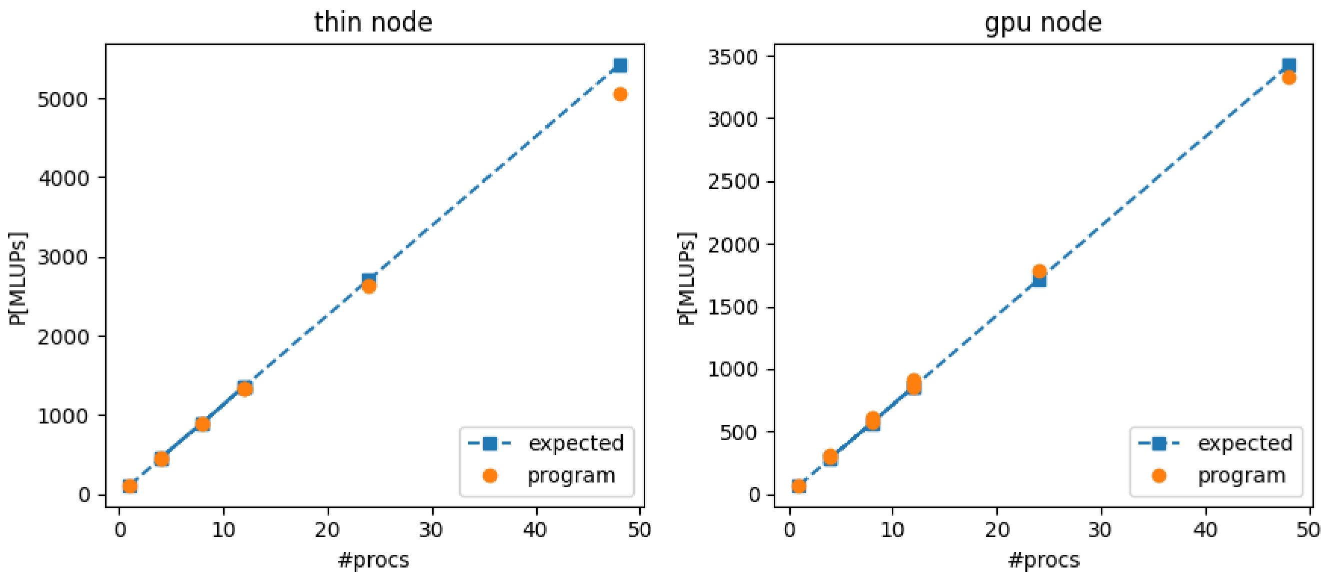

Figure 3: Caption

The last remaining cases to be considered are map-by-node `ucx` and `intel`. The situation here is the same for the two cases: the message passing between two nodes does not involve shared memory, instead it fully exploits the InfiniBand architecture. We can see this as the bandwidth continuously grows until it reaches the an approximate asymptotic bandwidth of 12 GB/s.

To conclude, the data we have gathered seem to suggest that the linear model discussed in class is a good approximation of the behavior of the network for small and very large message sizes. For intermediate message sizes, comparable with the L2 and L3 cache sizes, we have additional effects that disturb the expected linear behavior.

3 Jacobi solver performance analysis

Table 4 shows the data obtained from the Jacobi solver. The last two columns of thin and gpu sections compare the expected performance computed from the model discussed in class (`P_exp`), with the performance outputted by the program itself (`P_prg`). As we can see, the results are close to each other, but not perfectly matching (see Figure 3). This may be due to the program using a slightly different formula for the computation of `P_prg`. In the `P_exp` case we made use of the formula

$$P = \frac{N_p L^3}{T_s + T_c}, \tag{3}$$

where the value of T_s is estimated by running the program on a single processor. Since the program does not posses such estimate of the serial time, it has to resort to a different method for the computation of `P_prg`. Nonetheless, the resulting values are not so different, and we can conclude that the model discussed in class is a good approximation of the behavior of the performance of the Jacobi solver.

As a final consideration, it is also interesting to note that the expected performance value mainly depends on the serial time. Since the communication time is three orders of magnitude smaller than the the serial times ($T_s \approx 15$ sec for thin nodes and $T_s \approx 24$ sec for gpu nodes), the sum $T_s + T_c$ is almost identical to T_s, and the expected performance is mainly determined by this value.

						thin node			gpu node		
N	Nx	Ny	Nz	k	c(k)[MB]	Tc[s]	P_exp[MLUPs]	P_prg[MLUPs]	Tc[s]	P_exp[MLUPs]	P_prg[MLUPs]
-------------------------------- communication level: core --------------------------------											
1	1	1	1	0	0.0000	0.0000	112.9678	112.9680	0.0000	71.3881	72.3324
4	2	2	1	4	92.1600	0.0144	451.4460	452.5030	0.0144	285.3828	303.4910
8	2	2	2	6	138.2400	0.0216	902.4676	892.0190	0.0216	570.5959	582.5570
12	3	2	2	6	138.2400	0.0216	1353.7014	1325.0400	0.0216	855.8939	848.6420
------------------------------- communication level: socket -------------------------------											
4	2	2	1	4	92.1600	0.0165	451.3853	451.4320	0.0165	285.3585	309.9950
8	2	2	2	6	138.2400	0.0247	902.2858	885.7230	0.0247	570.5233	608.6970
12	3	2	2	6	138.2400	0.0247	1353.4287	1332.2400	0.0247	855.7849	895.3720
-------------------------------- communication level: node --------------------------------											
12	3	2	2	6	138.2400	0.0115	1354.5924	1337.1100	0.0115	856.2499	917.6100
24	4	3	2	6	138.2400	0.0115	2709.1848	2641.4800	0.0115	1712.4999	1787.3600
48	4	4	3	6	138.2400	0.0115	5418.3696	5066.4600	0.0115	3424.9998	3330.1000

Table 4: Results from the Jacobi solver for thin and gpu nodes.