FOUNDATIONS OF HPC
First assignment

First report
Master in High Performance Computing
Universitá degli studi di Trieste
Trieste, Italy

presented by:
Alexander Cristhoper Trujillo Ochoa

December 17, 2021

Chapter 1

MPI PROGRAMMING

1.1 Ring communication

1.1.1 Ring implementation

The implementation for this problem starts from defining a MPI Communicator that allows you to traverse all the processors in order and periodically. In this case using MPI Cart Create we define a 1D array with periodic boundary, MPI Shift gives to every processor the next and the previous processor, in this way you know who to send the message to.

```
MPI_Init( &argc, &argv );
MPI_Comm uroboros;
MPI_Comm_size( MPI_COMM_WORLD, &size );

MPI_Cart_create(MPI_COMM_WORLD, 1, proc_dims, periods, reorder,
 ↪  &uroboros);

MPI_Comm_rank(uroboros ,&current_rank);

MPI_Cart_shift(uroboros, 0, 1, &previous_ra
nk,&next_rank);
```

Now we can make a scheme of the communication between processor as shown in Fig. 1.1, each processor will send its rank as a MPI INT, but it will be positive for its right neighbor and negative for the left. The goal of the code is to obtain the message with its original tag but the content is the sum of every rank in the parallel program.

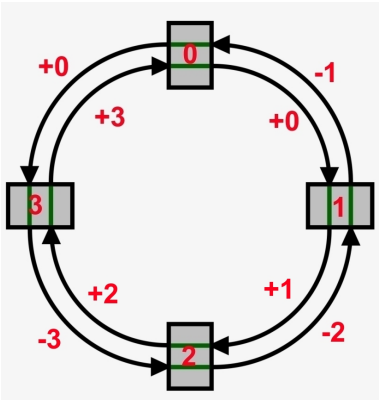

Figure 1.1: Scheme of the ring communication for 4 processors.

To start each processor sends two messages with the content mentioned before with a unique TAG proportional to its rank.

```
value_right=current_rank;
value_left=-current_rank;
MPI_Send( &value_right, 1, MPI_INT, next_rank, 10*current_rank,
 ↪  MPI_COMM_WORLD);
MPI_Send( &value_left, 1, MPI_INT, previous_rank,
 ↪  10*current_rank, MPI_COMM_WORLD );
```

Once they send their messages, each of them will start to collect the messages from their neighbors at the same time, and adding its rank to the content. Here to avoid deadlocks we will use MPI ANY TAG , this means that every processor will accept any message, just adding its rank to the content no matter what and send it to its neighbor.

```
for (int rank_iter=1; rank_iter<size;++rank_iter){
    MPI_Recv( &value_right, 1, MPI_INT, previous_rank,
     ↪  MPI_ANY_TAG, MPI_COMM_WORLD, &statusR);
    value_right+=current_rank;
    MPI_Send( &value_right, 1, MPI_INT, next_rank,
     ↪  statusR.MPI_TAG, MPI_COMM_WORLD);
    MPI_Recv( &value_left, 1, MPI_INT, next_rank, MPI_ANY_TAG,
     ↪  MPI_COMM_WORLD, &statusL);
    value_left-=current_rank;
    MPI_Send( &value_left, 1, MPI_INT, previous_rank,
     ↪  statusL.MPI_TAG, MPI_COMM_WORLD );}
```

As we can see, this code will be repeated as many times as number processor minus one are, in this way we assure that the message will travel to its original creator.

```
MPI_Recv( &value_right, 1, MPI_INT, previous_rank, MPI_ANY_TAG,
 ↪  MPI_COMM_WORLD, &statusR);
MPI_Recv( &value_left, 1, MPI_INT, next_rank, MPI_ANY_TAG,
 ↪  MPI_COMM_WORLD, &statusL);
```

At the end of the loop each processor will receive the message with its original tag. The left message will contain the negative of the sum of every rank, and the right message this positive value.

1.1.2 Scalability

To study the scalability of ring communication, lets graph the maximum walltime of each processor vs the number of processor involved as shown in Fig. 1.2.

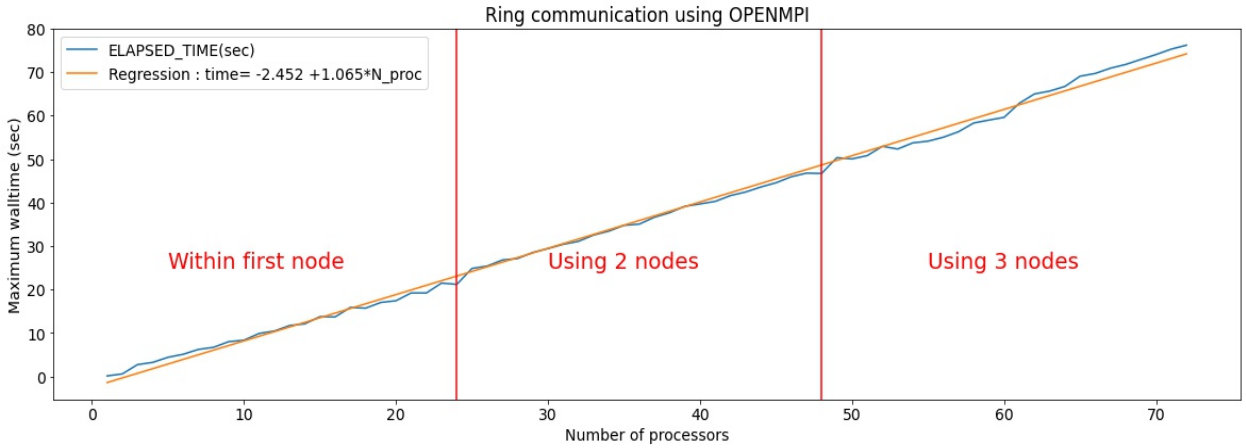

Figure 1.2: The walltime required to complete the ring communication scales linearly with the number of processors

We clearly see a linear trend, each time we add a processor it is predicted that the time required to perform 1M iterations will increase in 1 second, but will this continue ad infinitum? . Lets check the network performance!.

1.1.3 Network Performance

We performed **1 million times** the whole process, why?, because It is useful to compare network bandwidth in the order of Megabytes.
For example, if we have 3 processors, each processor will send 8 bytes (2 ints), and will do it 3 times, it means in total there was 72 bytes in movement. By iterating this for 1M times we obtain 72 MBytes in total, divided by the walltime it is in the order of MB/s and as we know that the peak performance of the network bandwidth is around 12000 MB/s. Considering that the overhead

caused by the calculations in each processor is less than 5% of the total walltime we can make a linear fit of the network bandwidth as shown in Fig. 1.3.

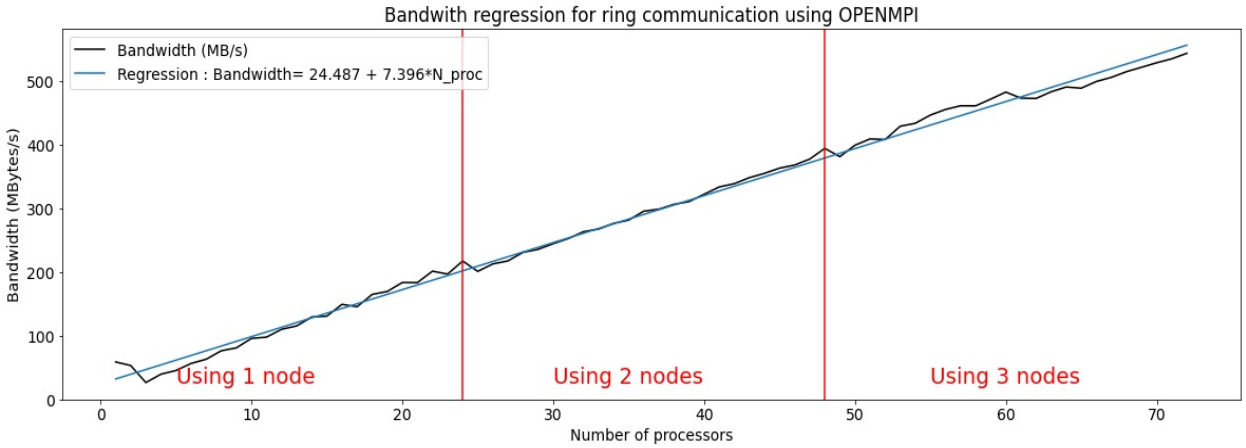

Figure 1.3: The network bandwidth increases linearly with the number of processors

The Infiniband network of ORFEO has a peak performance of around 12000 MB/s, so this means that the bandwidth will increase linearly with the number of processors but remain constant when reaches the peak performance, according to the linear regression this will happen when the number of processors is 1620. What will happen next? This means that the maximum walltime will not scale linearly anymore due to the saturation of the bandwidth, then this will remain constant, but the number of sent messages will grow quadratically and therefore the maximum walltime too.
To sum up , according to this implementation the maximum walltime will scale linearly up to 1600 processors and then quadratically.

1.1.4 An additional version

In the other-versions directory we can find a code called ring-spiral.c, the main modification of the first version is the iteration part, instead of each processor send their messages at the same time now will be ordered by rank. This means that the processor 0 will send their messages first, and then every processor will only take care of its content, once processor 0 receives back its message, processor 1 will start to send its messages and so on.

A caveat: This is extremely inefficient for the required goal, as we are sending only messages of a INT size and we are losing resources in this way. So what is it useful for? Could be useful in the case where you are sending huge size messages and the number of processors is considerable, then if every processor try to send their messages at the same time the network bandwidth will be over-saturated causing a bottleneck. So it will be a much better approach using this code in that case.

1.2 Sum of 3D matrices implementation

1.2.1 Sum 3D matrices implementation

We start calling the initialization of the parallel zone MPI Init and then asking the user to define the desired dimensions for the matrices and grid processors, after this we treat these dimension as the following:
- MPI DIMS CREATE will set up automatically the dimension of the grid processors if there is zeros in the array of dimensions. If it is not multiple of the available processors or it is a illegal input, the program will abort.
- If the dimensions of the grid processor are multiple of the dimension of the matrix, then the program will leave the matrix as it is.
- Otherwise, will enlarge the matrix to meet this requirement, the dimension of this new matrix will be stored in the augmented dimensions array.
- Once sure the dimensions of the grid are multiple of the dim of the matrix we can divide these and obtain the local dimensions for every processor.

After the main processor finishes this initialization, will broadcast this data to the remaining processors.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if(rank==0){
    scanf(<dimensions>);
    if(p_x==0p_y==0p_z==0){
        MPI_Dims_create(size, 3, proc_dims);}
    for(int i=0;i<3;++i){
        if(original_dim[i]\%proc_dims[i]==0){
            local_dim[i]=original_dim[i]/proc_dims[i];}
        else{
        aumented_dim[i]=original_dim[i]+<residual>;
        local_dim[i]=(aumented_dim[i]/proc_dims[i]);}
}}
MPI_Bcast(proc_dims, 3, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(aumented_dim, 3, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(local_dim, 3, MPI_INT, 0, MPI_COMM_WORLD);
```

Now we start with the initialization of the matrices by means of several processors. A 3D matrix is in fact a 1D array which size is the multiplication of the dimensions (matrix-size) and where you can access the memory allocation through 3 indexes (row, column, height). To start this initialization we consider the matrix-size, we check if it is divisible by the number of processors available (grid-size), if not, we find the maximum number of processors available multiple of this size (initialization-cores). We divide the matrix size and the number of processors (initialization-size), allocate 2 arrays (initialization arrays) of this size on each initialization core and fill it with

float random numbers between 0 and 1, in this way we balance the workload.
To obtain the original matrices we gather all these arrays in the main processor (We defined a MPI Communicator which only calls the initialization cores). We can find a scheme of this process in Fig. 1.4.

```
int init_cores=(matrix_size)\%(grid_size);
if(init_cores==0){
    init_cores=grid_size;}
else{
    for(int i=grid_size-1;i>0;--i){
        if((matrix_size)\%(i)==0){
            init_cores=i;
            break;}}}
if(my_rank<n_init_cores){
     for (size_t i=0;i<init_size;++i){
        init_array_1[i]= (rand() \% 101)/100.0;
        init_array_2[i]= (rand() \% 101)/100.0;
}}
MPI_Gather(init_array_1, init_size, MPI_DOUBLE, matrix_1,
 ↪  init_size, MPI_DOUBLE, 0, init_communicator);
MPI_Gather(init_array_2, init_size, MPI_DOUBLE, matrix_2,
 ↪  init_size, MPI_DOUBLE, 0, init_communicator);
```

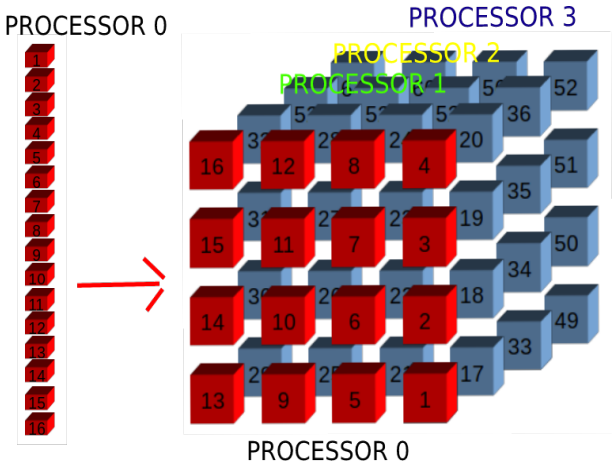

Figure 1.4: Gather initialization example: Consider 4 processors and a 4x4x4 matrix, each processor initialize one layer of the matrix in this case

The next step is to scatter these matrices in blocks to every single processor available, to do this we used MPI ScatterV, which allow us to send an array with a certain pattern distribution on a block of memory. To achieve this we set the parameters:
- Counts: this set the number of determined data types you want to collect starting from the current

memory pointer.
- Displaces: once collected the data, this will move the memory pointer the times you want the size of determined data types.

With this information we can create a new data type, in this case we will call it Block. This will check the local dimensions (partial size is the multiplication of these) for each processor and scatter the right pattern of memory to obtain the specific partial block of the original matrix. A scheme is shown in Fig. 1.5.

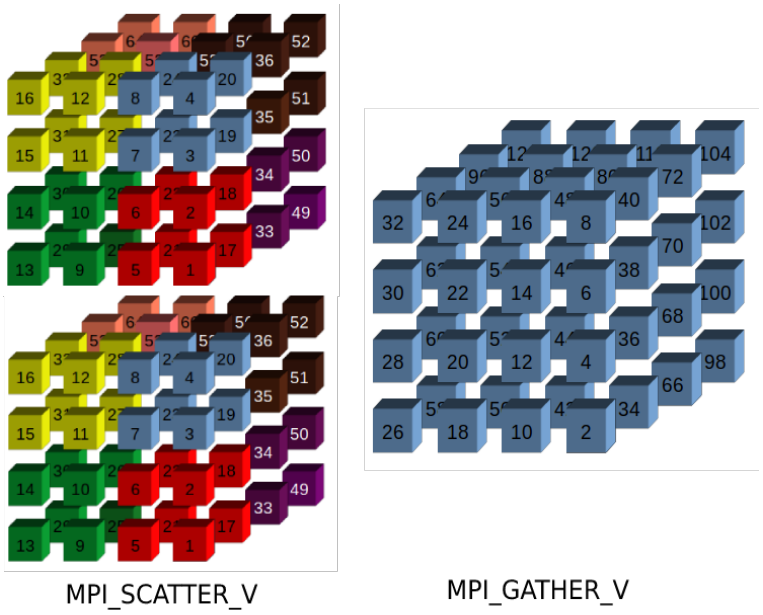

Figure 1.5: Example of distribution of the matrix in blocks, consider 8 processors and 4x4x4 matrix, each processor will have a 2x2x2 matrix (different color, different processor), then will compute the sum and finally all the result blocks are gathered in the original matrix

A caveat: If it is the case that original dimensions of the matrix are not multiple of grid processor dimensions, the we need to reallocate the original matrix as the size of the multiplication of the augmented dimensions.

```
if(my_rank==0){
    matrix_1=realloc(matrix_1,(augm_size)*sizeof(double));
    matrix_2=realloc(matrix_2,(augm_size)*sizeof(double));
}

MPI_Scatterv(matrix_1, counts, displaces, Block, partial_1,
 ↪  partial_size, MPI_DOUBLE, 0, WORLD COMM);
MPI_Scatterv(matrix_2, counts, displaces, Block, partial_2,
 ↪  partial_size, MPI_DOUBLE, 0, WORLD COMM);
```

The last steps involved are the sum computing on each processor and the final gathering of each piece of the matrix on the main processor. We set two elapsed times for each processor, one for the total walltime from initialization to calling free allocation functions and another walltime for calculating the communication overhead of MPI Gather and Scatter functions. Finally using MPI Reduce we collect the maximum of these walltimes.

```
for (int i=0;i<m_x;++i){
    for(int j=0;j<m_y;++j){
        for(int k=0;k<m_z;++k){
            partial_1[i][j][k]+=partial_2[i][j][k];}}}

MPI_Gatherv(partial_1, partial_size, MPI_DOUBLE, original_1,
 ↪  counts, displaces, resized_block, 0, WORLD COMM);
MPI_Reduce(&elapsed_time, &max_time,
 ↪  1,MPI_DOUBLE,MPI_MAX,0,WORLD COMM);
MPI_Reduce(&communication_time, &max_time_comm,
 ↪  1,MPI_DOUBLE,MPI_MAX,0,WORLD COMM);
```

1.2.2 Results

We kept the number of processors constant, for this case 24, then decomposed in 3D/2D/1D blocks each matrix in which we repeated 5 times every run to obtain the mean of the total and communication walltime. **Additionally to the previous parameters we compare the communication overhead with the total walltime, obtaining a relative percentage.**

Matrix dimensions	Grid processor	Max walltime	Comm walltime	relative%
2400x100x100	24x1x1	0.416384	0.342143	82.2
	6x4x1	0.418056	0.345782	82.7
	8x3x1	0.434415	0.356598	82.1
	3x8x1	0.446300	0.363094	81.4
	3x4x2	0.455446	0.378607	83.2
	4x3x2	0.478241	0.389986	81.6
	2x4x3	0.500079	0.409967	82
	1x8x3	0.514222	0.422041	82.7
1200x200x100	24x1x1	0.412987	0.346081	83.9
	3x8x1	0.417137	0.346645	83
	6x4x1	0.417228	0.3468	83
	8x3x1	0.429996	0.351212	81.6
	3x4x2	0.454580	0.382828	84.2
	4x3x2	0.470505	0.386433	82
	2x4x3	0.497770	0.409221	82.1
	1x8x3	0.507958	0.41172	81.1

Matrix dimensions	Grid processor	Max walltime	Comm walltime	relative%
800x300x100	8x3x1	0.415018	0.345112	83.1
	3x8x1	0.433947	0.355467	81.8
	6x4x1	0.434049	0.353042	81.3
	4x3x2	0.456969	0.377843	82.5
	3x4x2	0.473086	0.383209	81
	2x4x3	0.498368	0.408079	81.9
	1x8x3	0.504474	0.416242	82.5
	24x1x1	0.515322	0.395937	76.9

Table 1.2: Ordered results for sum of 3D matrices with 24 processors and size 24M doubles

For this new case, we used 48 processors for a bigger matrix size.

Matrix dimensions	Grid processor	Max walltime	Comm walltime	relative%
2400x1200x800	48x1x1	35.245861	30.3229	86
	16x3x1	35.476627	29.82564	84.1
	8x6x1	35.756171	31.02215	86.7
	8x3x2	37.701900	32.7236	86.8
	6x4x2	37.830866	29.23534	77.3
	2x3x8	39.911261	33.7938	84.7
	4x4x3	56.484326	36.1311	64
	2x8x3	56.683891	34.350395	60.6

Table 1.3: Ordered results for sum of 3D matrices with 48 processors and size 2.3G doubles

1.2.3 Performance discussion

According to the results presented before on the tables 1 and 2. We can observe the following:
- For 2400x100x100 and 1200x200x100 matrices the 1D distribution grid processor is slightly faster to complete the job in comparison with 2D distribution and 3D distribution is not so efficient, but for 800x300x100 matrix the 2D distribution is the fastest and outperforms 1D followed by 3D.
- **The communication overhead is predominant on this code and 80% percentage remains approximately constant when the dimension of the matrix are 24M doubles independent of the dimensions of the matrix or grid processor.**
- When the matrix increase in size (2.3G doubles) we observe that is not like the previous case, the communication overhead percentage decreases when the grid processor dimensions are unbalanced in comparison with the matrix dimensions, meaning that the memory access overhead starts to overcome because the memory allocation for each block is more spread and not cache friendly.

1.2.4 Other versions

There are two extra version of this code contained in the other version directory:
- The first one is Sum3DmatrixINT.c, a self explanatory name, instead of the float operations, the code only handles int operations as well as sending through MPI Scatter and MPI Gather only MPI INT. This could be up to 100% more efficient than the original with floats.
- The second one is Sum3Dinitcore.c, where one single processor takes care of the initialization of the matrix, useful to compare the initial communication overhead to send partial arrays to every core and gather vs the computation time of only one core.

Matrix dimensions	Grid processor	Max walltime	Comm walltime	relative%
2400x1200x800	48x1x1	111.134055	24.702932	22.2
	16x3x1	111.947573	25.027725	22.4
	8x6x1	111.863468	25.001880	22.3
	8x3x2	114.277251	27.334864	23.9
	6x4x2	113.801247	27.137679	23.8
	2x3x8	116.258030	29.550365	25.4
	4x4x3	116.588134	29.291302	25.1
	2x8x3	116.435344	29.300056	25.1

Table 1.4: Ordered results for sum of 3D matrices with 48 processors and size 2.3G doubles, only one core takes care of the initialization

The communication overhead here decreased drastically to 25% of the total walltime because we got rid of 2 Gather and 2 Scatter operations for the initialization of the matrices. Even though the original approach with multicore initialization outperforms this due to a great workload that falls in one single core. This means that this approach can not scale with the number of processor but it give us an idea if we obtain beforehand the matrices to sum, we would have gotten a better performance for the original approach.

Chapter 2

MEASURE MPI P2P PERFORMANCE

For every Framework we will run Intel MPI Benchmark PingPong sending in total 28 messages with different sizes(power of 2), dividing the cases between inter-node and intra-node (within same socket or different sockets). We used ORFEO THIN (ct1pt-tnode008,ct1pt-tnode009) and GPU nodes (ct1pg-gnode001,ct1pg-gnode003) for this study. Also we will compare the results with the communication time model as shown in Eq 2.1.

$$T_{comm} = \lambda + \frac{size}{b}. \tag{2.1}$$

Where λ is the latency and b the network bandwidth.

2.1 Using UCX point to point messaging layer

Unified communication X (UCX) is a independent library characterized by its low software-overhead. This library will use by default INFINIBAND network which peak performance is 100 Gbit/s (12500 Mbytes/s), few of the reasons why this library is highly optimized for HPC facilities are the efficient use of the memory registration, taking advantage of the available hardware as well as the asynchronous way to send messages where there is no need to of confirmations of the receiver.
Command to pin two processors of different nodes.

```
mpirun -np 2 --map-by node -report-bindings ./IMB-MPI1 PingPong
 ↪ -msglog 28
```

2.1.1 COMPARISON THIN-GPU

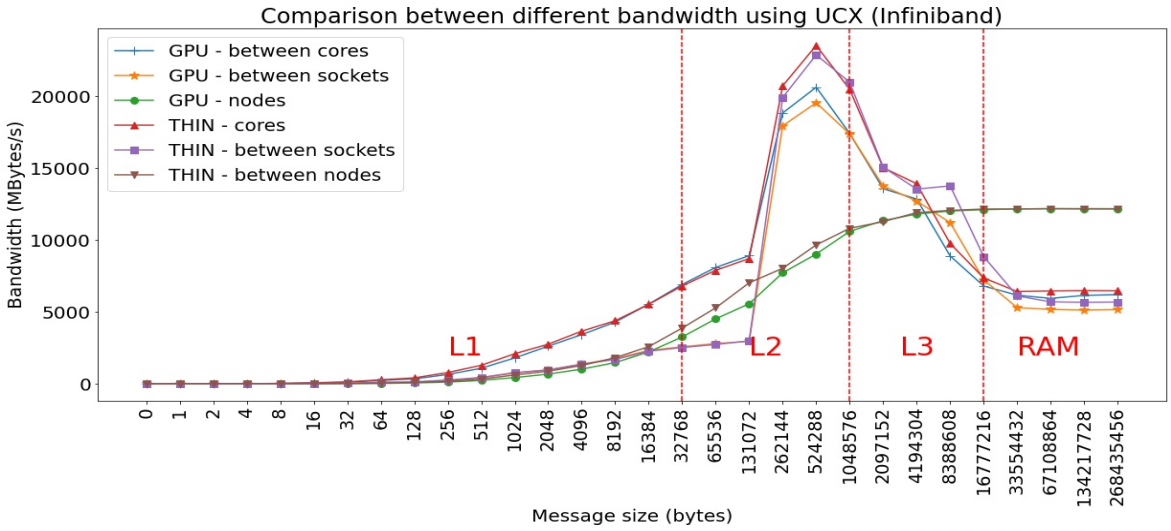

Figure 2.1: Comparison of bandwidth performance intra and inter node for GPU and THIN nodes using UCX

BANDWIDTH BEETWEEN CORES: As shown in Fig. 2.1, along L1 and L2 cache, network bandwidth between cores outperforms between sockets and nodes, this is because of cache-to-cache direct transaction [1] which is relativily cheap in terms of CPU cycles, once in the L3 zone the performance starts to drop due to L3 cache saturation as this is the shared cache between cores that belong to a socket and additionally other processors starts to make use of this resource as well. This behavior last until reach size messages more than L3 size , now it is compulsory for the processor to access the RAM, in this paying a lot in terms of latency and then we achieve a steady bandwidth.

BANDWIDTH BEETWEEN SOCKETS: Unlike the previous case, as processor does not share any cache, this communication will rely on a buffer located in their caches. This is possible as we are in a NUMA topology so memory access is allowed within the same node, we can see that it shows the same behavior as between cores network bandwidth but with less performance, this because now there is an additional overhead, the local bus interconnect between NUMA domains.

BANDWIDTH BEETWEEN NODES: This case is totally different, now the message must go through the network interface card, the performance now will depend on the transmission performance of the Infiniband network architecture (12500 Mbytes/s) . The behavior of this throughput is well known asymptotic and reaches its peak when sends messages with size bigger than 20 MBytes.

EXAMPLE: INTER-NODE NETWORK BANDWIDTH

We will compare the inter-node configuration with the network performance model shown in Eq. 2.1. Besides, using UCX the peak performance achieved is 12150 Mbytes/s as shown in Fig. 2.2, meaning a 97,2% efficiency.

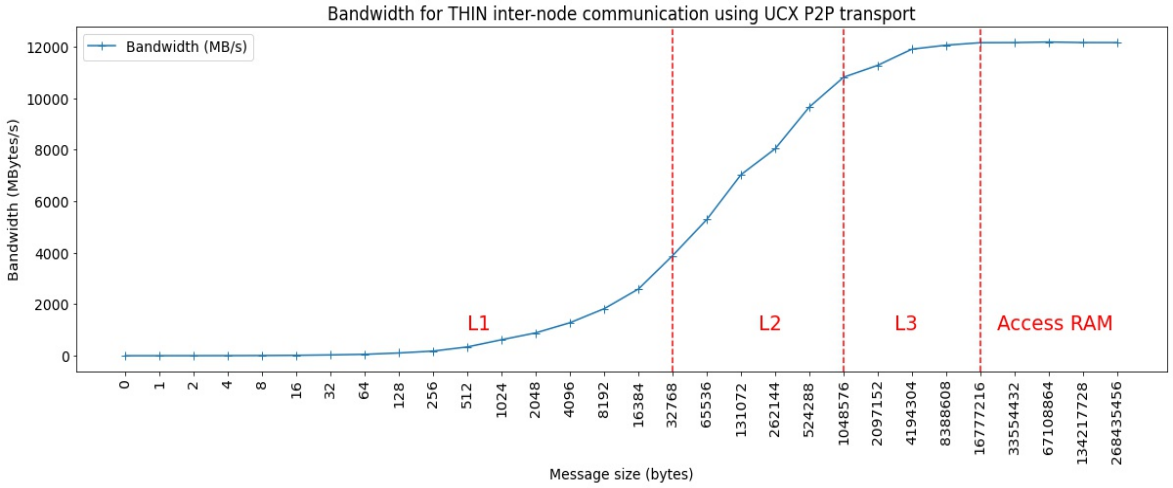

Figure 2.2: Bandwidth performance vs the message size for communication between different nodes

Using the previous model we obtain a latency of 1.01 microsecond and bandwidth of 12157 Mbytes/s. As shown in Fig. 2.3, exists three zones, one where the latency predominates as the message size is to small, then when the bandwidth starts to increase there is a overestimation and therefore a mismatch in the model, at the end the result starts to match the model as the throughput reaches its maximum performance.

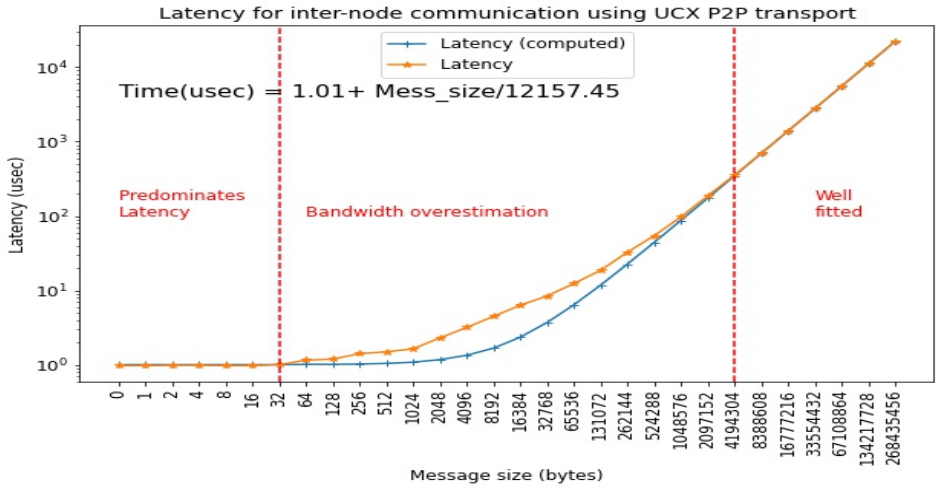

Figure 2.3: Regression for time communication P2P using the latency and bandwidth parameters

If interested in the left cases using UCX,OB1 and INTELMPI: csv files, fits and plots are in the FIGURES directory.

2.2 Using OB1 point to point messaging layer

OB1 was the OPENMPI's original point to point transport engine, by default using TCP/IP protocol and ETHERNET network which peak performance is 25 Gbit/s (3125 Mbytes/s), using this protocol the latency increases because it will split the messages into several parts and makes sure your message has been received by waiting for an acknowledgment from the receiver, taking lots of buffering and checking so latency become higher even for small messages.
Command to pin two processors of different nodes.

```
mpirun --mca pml ob1 --mca btl self,tcp  -npernode 1 -np 2
 ↪ -report-bindings ./IMB-MPI1 PingPong -msglog 28
```

2.2.1 COMPARISON THIN-GPU

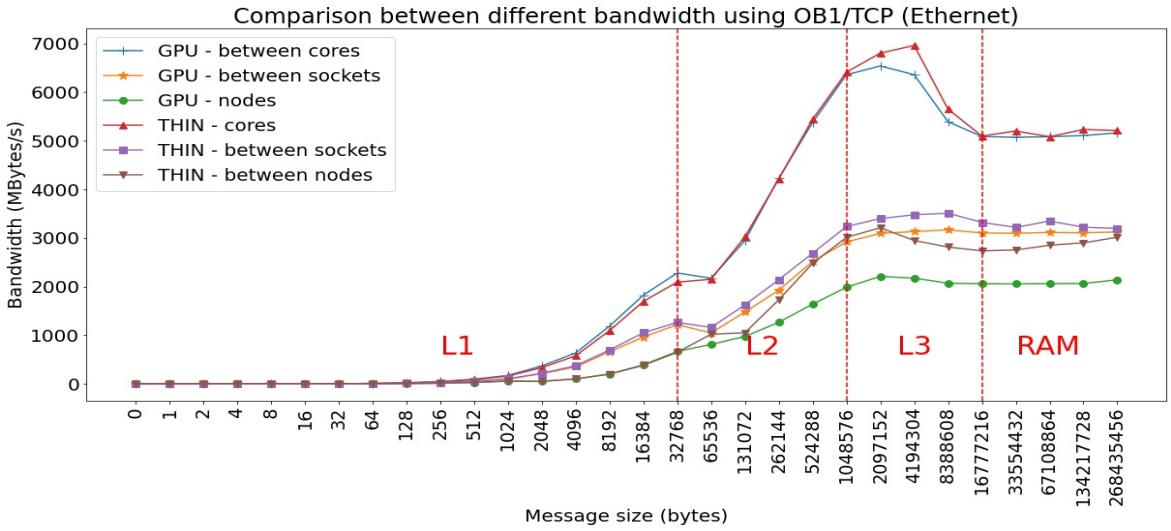

Figure 2.4: Comparison of bandwidth performance intra and inter node for GPU and THIN nodes using OB1

BANDWIDTH BEETWEEN CORES: As shown in Fig. 2.4, network bandwidth between cores totally outperforms between sockets and nodes, now this is because of two reasons, cache-to-cache direct transaction which is relatively cheap in terms of CPU cycles and now thanks to OB1 Point to point messaging layer, the message is split across byte transfer layers (in this case VADER as replicates shared memory approach) therefore the communication overhead increases but now the throughput behavior is almost asymptotic because the subdivision allows cache friendly messaging.

BANDWIDTH BEETWEEN SOCKETS: Similar process as the previous case, but this time communication overhead will increase because of the local bus interconnect between NUMA domains.
BANDWIDTH BEETWEEN NODES: The performance now will depend on the transmission performance of the Ethernet network architecture (3.125 MBytes/s) . The behavior of this throughput is well known asymptotic and reaches its peak when sends messages with size bigger than 20 MBytes.

2.3 Using INTELMPI (MLX point to point messaging layer)

INTELMPI uses MLX providers and uses by default INFINIBAND network, this is highly optimized for INTEL processors, taking care of the latency, which is the best for the available hardware.
Command to pin two processors of different nodes.

```
mpirun -n 2 -ppn 1 -hosts ct1pt-tnode008,ct1pt-tnode009
 ↪  ./IMB-MPI1 PingPong -msglog 28
```

2.3.1 COMPARISON THIN-GPU

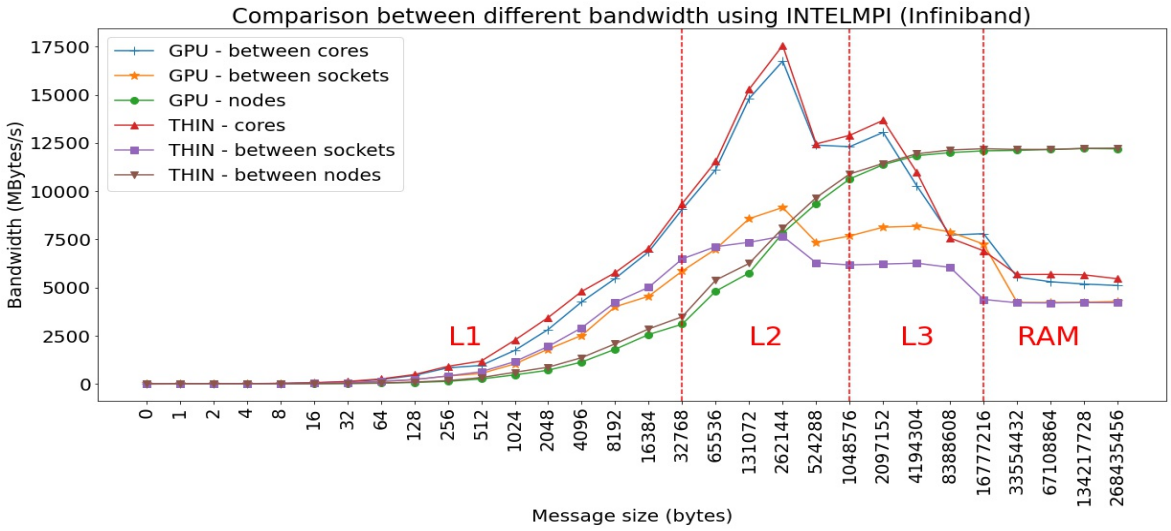

Figure 2.5: Comparison of bandwidth performance intra and inter node for GPU and THIN nodes using INTELMPI

As shown in Fig. 2.5, the behavior of the throughput is comparable and explained in the same way as UCX protocol.

Chapter 3

PERFORMANCE MODEL FOR JACOBI SOLVER

3.1 Data extraction

To compile this code OPENMPI-4.1.1 was loaded (UCX Library and Infiniband network as default), we set up the following dimension for the grid: 1200x1200x1200, we let MPI decide the grid processor and we set periodic boundaries on 3 dimension. The output will give the size of each allocated array, each processor will take care of 2 arrays.
For every run of JacobiMPI we obtain 10 iterations of the solver, and for each one we will focus on the maximum walltime and Million Latice updates /s.

According to the network bandwidth model as shown in Fig. 2.1, we obtained the following parameters from the Section 2 of the report.

Node	pinning	$\lambda(\mu sec)$	Bandwidth (Mbytes/s)
THIN	inter-node	1.01	12157.45
THIN	different socket	0.44	5703.02
THIN	same socket	0.19	6476.264
GPU	inter-node	1.41	12173.45
GPU	different socket	0.46	5171.58
GPU	same socket	0.23	6186.6

Table 3.1: Parameters for network bandwidth model (USING OPENMPI AND INFINIBAND)

3.2 Performance Model and results

According to [2] the performance model is proportional to the number of processors as they have a balanced amount of work, expressed in the formulas shown in Fig. 3.1.

The data volume transferred between the network link can be modeled as $C = L^2 * k * 2 * 8$ Mb, where k is twice the largest number (over all sub-domains) of coordinate directions in which the number of processes is greater than one.
We can compute the difference between the theoretical and actual performance using the "slow-down" factor which is the ratio between N times the single processor performance over the actual multiprocessor performance.

$$P(L,\vec{N}) = \frac{L^3 N}{T_s(L) + T_c(L,\vec{N})} \qquad T_c(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_\ell$$

(a) Performance formula for N processors (b) Communication time

Figure 3.1: The performance model for the Jacobi solver

#Node, #proc and k	pinning	Max walltime(s)	Array size(MB)	Tc (ms)
2 thin/ 48/ 6	inter-node	0.313780	586.62	0.0114
2 thin/ 24/ 6	inter-node	0.596978	1169.36	0.0114
1 gpu/ 48/ 6	hyperthread	0.630304	586.62	0.0267
1 gpu/ 24/ 6	hyperthread	0.930464	1169.36	0.0223
1 thin/ 12/ 6	same socket	1.185807	2330.98	0.0213
1 thin/ 12/ 6	different socket	1.187631	2330.98	0.0242
2 thin/ 12/ 6	inter-node	1.189551	2330.98	0.0114
2 gpu/ 48/ 6	inter-node	1.214391	586.62	0.0114
2 gpu/ 24/ 6	inter-node	1.560386	1169.36	0.0114
1 thin/ 8/ 6	different socket	1.760208	3490.68	0.0242
1 gpu/ 12/ 6	different socket	1.763089	2330.98	0.0267
1 thin/ 8/ 6	same socket	1.763395	3490.68	0.0213
1 gpu/ 12/ 6	same socket	1.866811	2330.98	0.0223
2 gpu/ 12/ 6	inter-node	2.430502	2330.98	0.0114
1 gpu/ 8/ 6	different socket	2.570280	3490.68	0.0267
1 gpu/ 8/ 6	same socket	2.687856	3490.68	0.0223
1 thin/ 4/ 4	different socket	3.820511	6969.75	0.0161
1 thin/ 4/ 4	same socket	3.831171	6969.75	0.0142
1 gpu/ 4/ 4	different socket	5.581861	6969.75	0.0178
1 gpu/ 4/ 4	same socket	5.593812	6969.75	0.0149
1 thin/ 1/ 2	single core	15.256329	27786.50	0
1 gpu/ 1/ 2	single core	22.077433	27786.50	0

Table 3.2: Ordered results for the performance of different nodes and number of cores

As we presented in the table 3.2, the maximum walltime for completion and communication overhead (Tc) for every scenario, we can see the best cases are using 2 THIN nodes or 1 single GPU node using hyper-threading.

SERIAL TIME

According to the table 3.2. we obtained the serial time and performance for THIN node : (15.256 s ,113.264 MLUPS/s) ; and for GPU node: (22.077 s. ,78.270 MLUPS/s).
In the following tables we will compare the results obtained by running the JACOBI SOLVER in ORFEO with the performance model.

INTRA-NODE SAME SOCKET

Node # proc	Actual P(L,N)	Actual $\frac{NP_1}{P(L,N)}$	Model P(L,N)
THIN 4	451.037	1.00	453.068
THIN 8	809.857	1.12	906.135
THIN 12	1204.327	1.13	1359.203
GPU 4	308.913	1.01	313.043
GPU 8	531.315	1.18	626.087
GPU 12	764.994	1.23	939.13

Table 3.3: Performance parameters for Jacobi solver considering communication between cores

INTRA-NODE DIFFERENT SOCKETS

Node # proc	Actual P(L,N)	Actual $\frac{NP_1}{P(L,N)}$	Model P(L,N)
THIN 4	452.295	1.0	453.068
THIN 8	811.324	1.12	906.135
THIN 12	1202.478	1.13	1359.203
GPU 4	309.574	1.01	313.043
GPU 8	555.620	1.13	626.087
GPU 12	809.998	1.16	939.13

Table 3.4: Performance parameters for Jacobi solver considering communication between sockets

INTER NODE COMMUNICATION

Node # proc	Actual P(L,N)	Actual $\frac{NP_1}{P(L,N)}$	Model P(L,N)
THIN 12	1200.536	1.13	1359.203
THIN 24	2392.217	1.14	2718.406
THIN 48	4562.674	1.19	5436.811
GPU 12	587.574	1.6	939.130
GPU 24	923.321	2.034	1878.261
GPU 48	1185.469	3.17	3756.521

Table 3.5: Performance parameters for Jacobi solver considering communication between nodes

HYPERTHREADING ENABLED

In this case we took advantage of the available logical cores, for 12 and 24 processors we pinned the processors with its logical pair within same socket, for 48 processor we used the entire node.

Node # proc	Actual P(L,N)	Actual $\frac{NP_1}{P(L,N)}$	Model P(L,N)
HT GPU 12	868.295	1.08	939.130
HT GPU 24	1534.826	1.22	1878.261
HT GPU 48	2265.736	1.66	3756.521

Table 3.6: Performance parameters for Jacobi solver using physical and logical cores from a GPU node

3.3 Scalability of Jacobi Solver

Considering the workload constant, as shown in Fig. 3.2 this behavior is explained by AMDAL'S LAW [3] which states that serial code and communication overhead decreases the scalability, besides the GPU has a greater slowdown factor than THIN nodes because of its greater serial time and communication overhead which plays an important role for the scalability.
For this reason using 48 processors with hyperthreading outperforms 48 processors using two GPU nodes due to a less communication overhead.

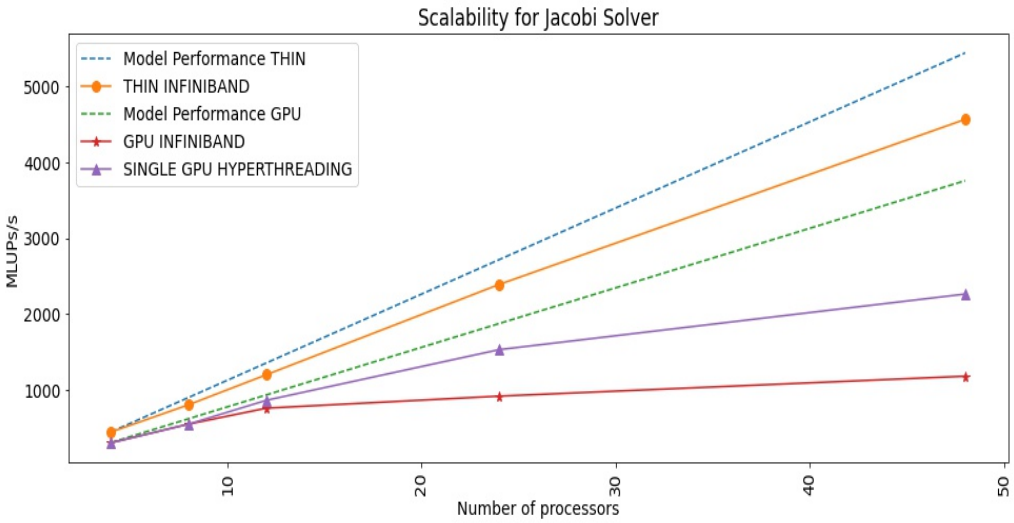

Figure 3.2: Comparison between GPU and THIN scalabilities

Bibliography

[1] CHOPP, D. L. (2019). *Introduction to High Performance Scientific Computing (First ed.).* SIAM - Society for Industrial and Applied Mathematics.

[2] CHAI, L., LAI, P., JIN, H. W., & PANDA, D. K. (2008). *Designing an Efficient Kernel-Level and User-Level Hybrid Approach for MPI Intra-Node Communication on Multi-Core Systems.* 2008 37th International Conference on Parallel Processing. https://doi.org/10.1109/icpp.2008.16

[3] AMDAHL, G. M. (2007). *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* 1967, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California. IEEE Solid-State Circuits Newsletter, 12(3), 19–20. https://doi.org/10.1109/n-ssc.2007.4785615