# COMP534: Project3: 2.5D Matrix Multiplication using MPI

Dejan Grubisic

April 30, 2020

**Abstract**

This project implements the general version of the 3D matrix multiplication algorithm called 2.5D matrix multiplication, discovered in 2011. by Edgar Solomonik and James Demmel [1]. This algorithm is communication optimal and it trades some extra memory to alleviate the communication of the 2D-Cannon algorithm. The implementation is based on MPI and the experiments are done for the different number of processors ranging from 4 to 64 for different values of parameter c, which is the number of copies of local sub-matrices. The time results are shown for matrix size 7500x7500. For correctness proving and performance analysis, we used HPCToolkit and jumpshot4.

1. **The problem statement and a short motivation for why solving this problem is useful.**

Matrix multiplication represents an essential operation in linear algebra, which is widely used in scientific computing. Unfortunately, the complexity of such operation is $O(n^k)$, where $k \in [2,3]$, and for large matrices, the execution times could be unacceptably long. To make this computation faster, it is necessary to use an efficient distributed algorithm and to optimize communication as much as we can.

Cannon's algorithm is the first and conceptually the simplest distributed algorithm for matrix multiplication. It is also called the 2D algorithm because it organizes all processors into one layer square with side $\sqrt{P} \times \sqrt{P}$ (Figure 1). In this example there is 9 processors that form matrix 3x3 and each of processors take submatrix of $\sqrt{\frac{N}{P}} \times \sqrt{\frac{N}{P}}$ elements.

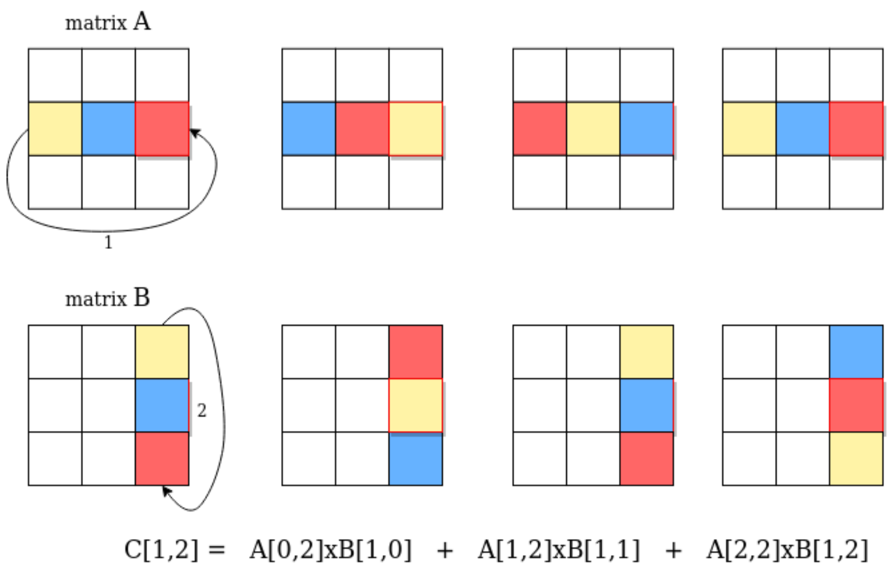


Figure 1: Cannon 2D matrix multiplication

In the first step, every row of the first matrix A and every column of the matrix B is shifted circularly on the left and to the up by row number and column number respectively. After this step, each processor contains submatrices A and B that need to be multiplied to get the first partial result. Once the local matrix multiplication is done, processors just shift to the left submatrices A and B and calculate new partial results that add to the previous result. The algorithm terminates in $\sqrt{P}$ steps, with communication $2 \times \frac{N^2}{P}$. As the bandwidth is sum of all communications per each step we get $O(\frac{N^2}{\sqrt{P}})$.

The main goal of the 3D matrix multiplication algorithm is to lower communication if there is additional memory for each processor. To use 3D algorithm we need to have enough memory to store 3 submatrices of size $\frac{N}{\sqrt[3]{P}} \times \frac{N}{\sqrt[3]{P}}$. In this case, processors are layered in a cube with side $\sqrt[3]{P}$. In the beginning, we assume that starting matrices are stored in the first layer and the first step is to propagate them to other layers (Figure 2). Once each layer gets its copy, they will perform the offset step of Cannon's algorithm but this time the offset will be increased by the layer number so that each processor calculates a different partial result. After we calculate partial results we just need to propagate them back to the first layer and sum up with other partial results.

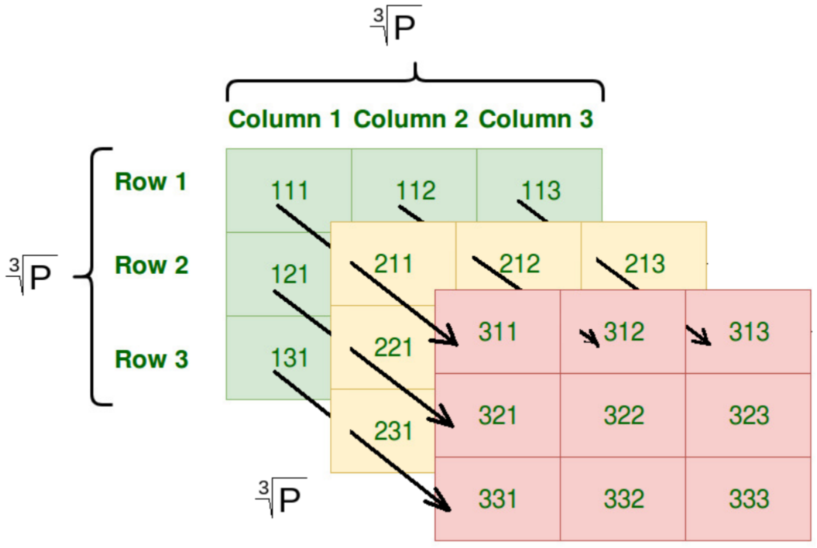


Figure 2: The 3D algorithm for matrix multiplication

The total latency of communication, in this case, is $logP$ as broadcast and reduce step could be done in $log\sqrt[3]{P}$ time and we need O(1) time for circular shifts. The bandwidth, in this case, is $O(\frac{N^2}{\sqrt[3]{P^2}})$ as there are 3 communication steps in which every processor exchange their submatrices.

The problem with the 3D algorithm is that it assumes that each processor has $O(\frac{N^2}{\sqrt[3]{P^2}})$ memory. For large matrix, this is hard to achieve as necessary memory grows proportionally to the matrix size. Luckily, all processors that have only c times more memory than for 2D algorithms can use 2.5D algorithm with optimal communication.

2. **The details of your approach and an explanation of how/why this approach solves your problem.**

The idea behind 2.5D algorithm is to create c layers with $\sqrt{\frac{P}{c}} \times \sqrt{\frac{P}{c}}$ processors. As each layer has $N^2$ elements, each processor gets $c \times \frac{N^2}{P}$ matrix elements. The workflow of the algorithm is the combination of 3D and 2D algorithm. We first have to propagate submatrices like in the 3D algorithm and to shift submatrices in each layer for their offset. After that there are $\sqrt{\frac{P}{c^3}}$ rounds of 2D algorithm for each of c layers to sum up all of $\sqrt{\frac{P}{c}}$ partial results. After the processors sum up all partial results, they aggregate their results in the first layer.

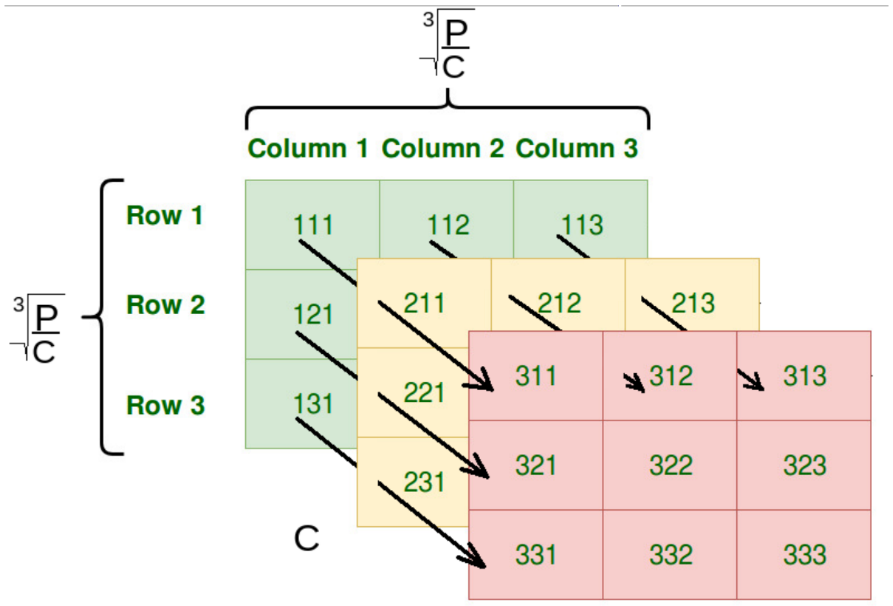


Figure 3: The 2.5D algorithm for matrix multiplication

The latency of 2.5D algorithm is $\Omega(\sqrt{\frac{P}{c^3}})$ as we distribute $\sqrt{\frac{P}{c}}$ computations accross c layers. The bendwidth is $\frac{N^2}{\sqrt{Pc}}$ as we are sending submatrices $\frac{N}{\sqrt{\frac{P}{c}}} \times \frac{N}{\sqrt{\frac{P}{c}}}$, in $\sqrt{\frac{P}{c^3}}$ steps.

## MPI Implementation

MPI program starts with the initialization of ProcessInfo datastructure which stores the information about process topology and ranks for each process inside it. As we already mentioned the process are organized in c layers with each layer having $\frac{P}{c}$ processors. Processes are connected by communicator xy_comm across layers, and with c_comm accross c dimension.

After the setup of ProcessInfo, master thread creates $N$ matrices with random values, divides the matrices into blocks, and distributes them to other processes in the first layer by using function MPI_Scatterv. To divide the starting matrix into blocks, we are defining a new MPI_Datatype called blocktype. Blocktype is obtained from MPI_Type_vector with x and y coordinates set to the number of elements inside a tile, while the third parameter is a distance in memory between each row of the block.

```
MPI_Datatype blocktype;
MPI_Type_vector(info.tile_el_side, info.tile_el_side, info.mat_side,
                MPI_INT, &blocktype);
MPI_Type_create_resized(blocktype, 0, sizeof(int), &blocktype);
MPI_Type_commit(&blocktype);
```

MPI_Type_create_resized function just sets the bounds for each element of blocktype. After calling

MPI_Type_commit, the block type can be used in the rest of the program.

Function MPI_Scatterv takes the address of the distribution matrix, divides it in blocktypes, and send it to different processors into sub_matrix buffer. Arrays *counts* and *disps* determine how many blocktypes are be sent to each processor and at which index the subblock starts in full_matrix (Figure 4).

```
MPI_Scatterv(    &full_matrix[0], &counts[0], &disps[0], blocktype,
                 &sub_matrix[0], info.tile_el, MPI_INT, 0, info.xy_comm);
```

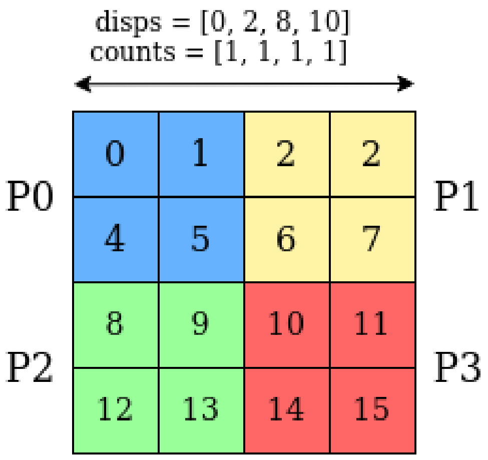


Figure 4: Blocktype

Once all submatrices from A and B are distributed, each processor in the first layer will broadcast copies of their submatrices across communicator c_comm. At this point, all processors have their copy of data, and the next step is to calculate offsets for their submatrices based on their ranks. For submatrix A - shift circular left for the row number and shift right for the $\sqrt{p\_layer}$. For the submatrix B shift circular upper for the column number and shift bottom for $\sqrt{p\_layer}$, where $p_layerisnumberofprocessorsinthelayer.ThesecircularshiftsareimplementedbyfunctionMPI\_Sendrecvwhic$

```
MPI_Sendrecv(&send_buff[0], info.tile_el, MPI_INT, send_to, 0,
               &recv_buff[0], info.tile_el, MPI_INT, recv_from, 0, info.xy_comm, st
swap(send_buff, recv_buff);
```

When processors exchange submatrices, they are ready for sequential matrix multiplication on them. When this step is done, all processors across communicator c_comm has the partial result for one tile. If the $c < \sqrt[3]{P}$, there are $\sqrt{\frac{P}{c^3}}$ additional partial results to be calculated for each processor. To calculate these results the algorithm is making the simple right and bottom circular shift on A and B submatrices and accumulates locally their matrix multiplication.

When all partial results are calculated and summed up locally, the algorithm sends them to the processors from the first layer, where the contributions from all processors from dimension c are summed up. At this point, the final matrix C is calculated and its subblocks are distributed across the processors in the first layer. To send it back we can use function MPI_Gatherv that accepts the same parameters as already explained MPI_Scatterv and combine submatrices to recreate the final matrix in the processor with rank 0.

3. **A description of the experiments you conducted. Give the quantitative and qualitative analysis of your results.**

   Experiments had been conducted on 4, 8, 9, 16, 25, 27, 32, 36, 64 nodes for the appropriate value of c on the architecture given in Appendix(Table 1). Time results are shown in Figure 5. From the given charts, we can conclude that speed mostly depends on the number of processors and the hardware architecture we are using. The value of c doesn't change execution speed in this case and other characteristics of hardware architecture and interconnect overlay the acceleration contributed by c.

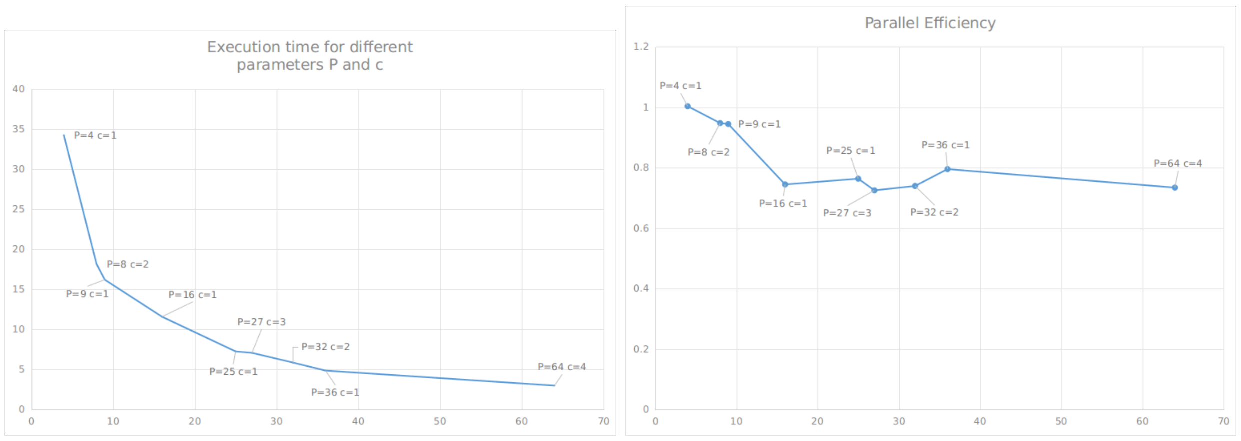


Figure 5: Execution time and Parallel Efficiency

To calculate Parallel Efficiency we calculate matrix multiplication sequentially on one node. From the Parallel Efficiency plot, it seems that for configuration of 4 ranks and c equals to 1 we get a superlinear speedup. The reason for this could be the additional cache memory that we get with each new processor included in the computation. On the other hand, with more ranks, communication increases a little bit so we have a decreasing trend of parallel efficiency.

**Jumpshot4 results**

For visualization of MPI execution and the communication between ranks, we used a tool called Jumpshot4. In Figure 6 we can full execution time with the legend on the top, zoomed the first phase in the middle, and zoomed the middle phase on the bottom. Based on the top figure we can see that the computation is divided into two part-pure lines without color, while there is some communication at the beginning, in the middle, and on the end.

When the program starts, the processor with rank 0 needs to distribute A and B matrices to other processors from the first layer. This is done with MPI_Scatterv colored in pink and we can see the steps pattern that refers to this in the middle figure. Yellow stands for MPI_Barrier and we can see that all ranks from the second layer go straight there while ranks from the first layer go there when they got their submatrix. At this point, the 2.5D algorithm starts. Firstly, ranks calculate their offset and the ranks from the first layer broadcast their submatrices to their neighbors in c dimension (Bcast is colored in light blue). When all ranks get submatrices they do a collective shift colored in green.

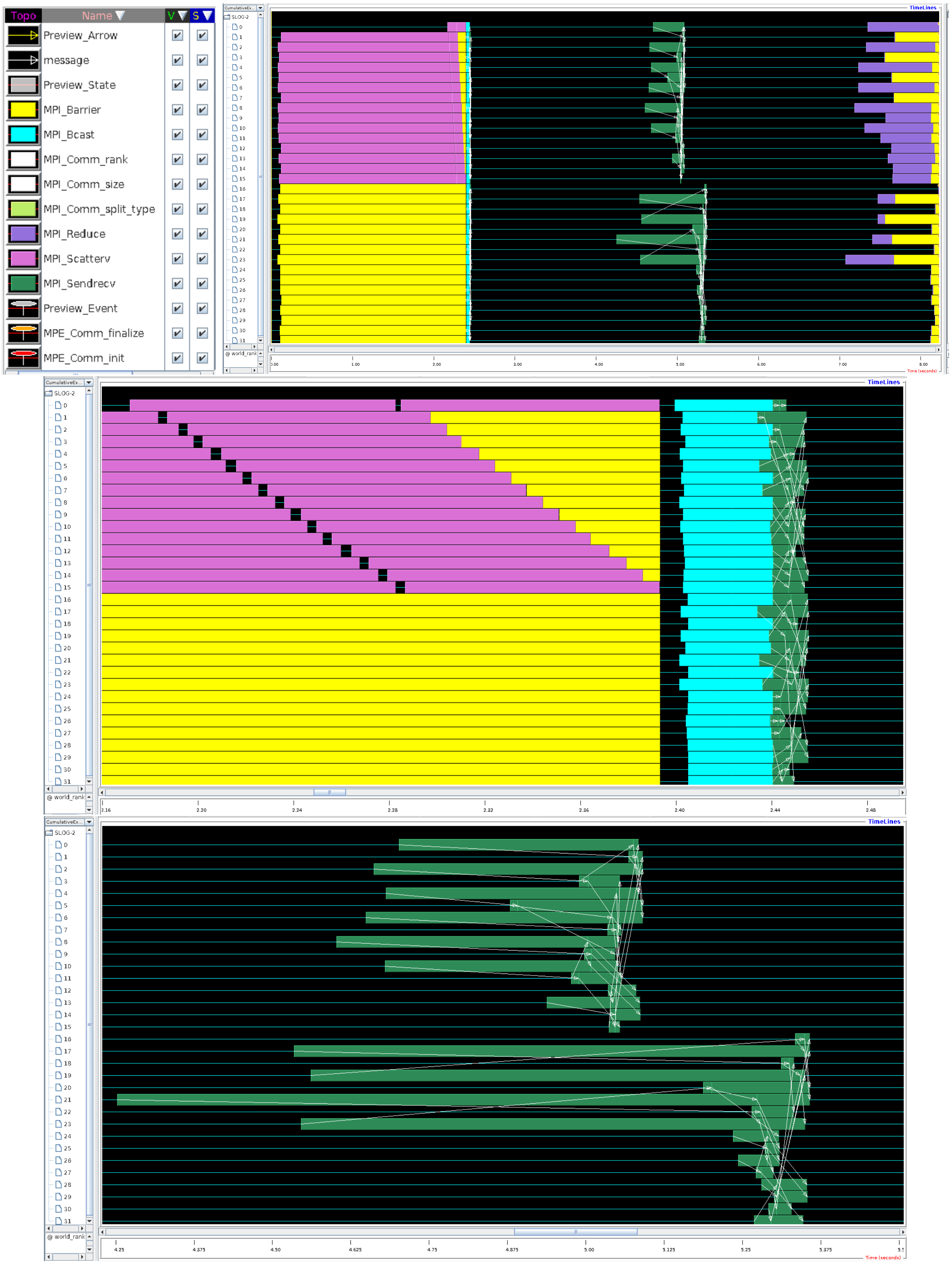
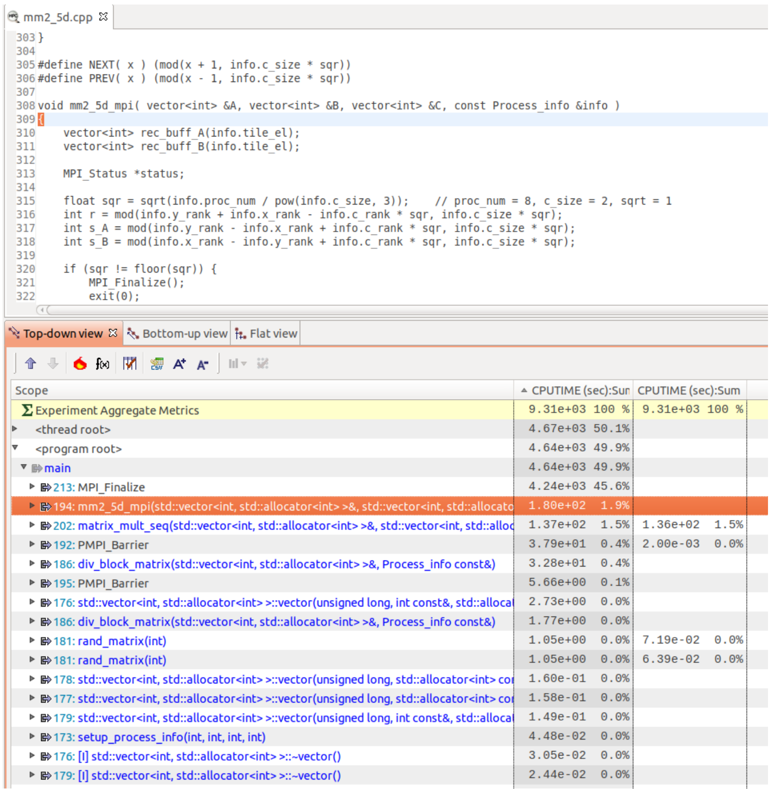


Figure 6: Jumpshot4 results for P=32 N=7500 c=2

At this point, all ranks got their submatrices and they perform local matrix multiplication on them to create the first partial result. Some ranks are going to finish this multiplication before others because some of them are sharing the same core, which is defined by the affinity parameter. This behavior could be seen on the bottom figure in green strides in which faster threads are waiting slower to perform shift operation. After collective shift operation on each layer, ranks are doing again sequential matrix multiplication on new tiles. Finally, ranks sum up partial results from two computation steps and send them back to the ranks at the first layer which sums up the contribution of each layer. This is done in MPI_Reduce colored in violet.

### HPC Viewer results

The experiments with HPCToolKit are done in the same configuration as jumpshoot4 (P=32, N=7500, c=2) with sampling CPUTIME which shows the time spent inside the program. From the HPC Viewer tool (Figure 7), we can see that 51.1% is spent inside the program root that means that MPI runtime takes the rest of the time. Inside mm2_5d function, the program spends 4.44% and the majority of the time is spent inside matrix_mult_seq and matrix_multadd_seq, 21% and 20.9% respectively. This behavior is expected because both of these steps represent the calculation of partial results.



```
}

#define NEXT( x ) (mod(x + 1, info.c_size * sqr))
#define PREV( x ) (mod(x - 1, info.c_size * sqr))

void mm2_5d_mpi( vector<int> &A, vector<int> &B, vector<int> &C, const Process_info &info )
{
    vector<int> rec_buff_A(info.tile_el);
    vector<int> rec_buff_B(info.tile_el);

    MPI_Status *status;

    float sqr = sqrt(info.proc_num / pow(info.c_size, 3));    // proc_num = 8, c_size = 2, sqrt = 1
    int r = mod(info.y_rank + info.x_rank - info.c_rank * sqr, info.c_size * sqr);
    int s_A = mod(info.y_rank - info.x_rank + info.c_rank * sqr, info.c_size * sqr);
    int s_B = mod(info.x_rank - info.y_rank + info.c_rank * sqr, info.c_size * sqr);

    if (sqr != floor(sqr)) {
        MPI_Finalize();
        exit(0);
```

| Scope | CPUTIME (sec):Sum | CPUTIME (sec):Sum |
|---|---|---|
| Experiment Aggregate Metrics | 9.31e+03 100 % | 9.31e+03 100 % |
| <thread root> | 4.67e+03 50.1% | |
| <program root> | 4.64e+03 49.9% | |
| main | 4.64e+03 49.9% | |
| 213: MPI_Finalize | 4.24e+03 45.6% | |
| 194: mm2_5d_mpi(std::vector<int, std::allocator<int> >&, std::vector<int, std::allocato | 1.80e+02 1.9% | |
| 202: matrix_mult_seq(std::vector<int, std::allocator<int> >&, std::vector<int, std::alloc | 1.37e+02 1.5% | 1.36e+02 1.5% |
| 192: PMPI_Barrier | 3.79e+01 0.4% | 2.00e-03 0.0% |
| 186: div_block_matrix(std::vector<int, std::allocator<int> >&, Process_info const&) | 3.28e+01 0.4% | |
| 195: PMPI_Barrier | 5.66e+00 0.1% | |
| 176: std::vector<int, std::allocator<int> >::vector(unsigned long, int const&, std::allocat | 2.73e+00 0.0% | |
| 186: div_block_matrix(std::vector<int, std::allocator<int> >&, Process_info const&) | 1.77e+00 0.0% | |
| 181: rand_matrix(int) | 1.05e+00 0.0% | 7.19e-02 0.0% |
| 181: rand_matrix(int) | 1.05e+00 0.0% | 6.39e-02 0.0% |
| 178: std::vector<int, std::allocator<int> >::vector(unsigned long, std::allocator<int> co | 1.60e-01 0.0% | |
| 177: std::vector<int, std::allocator<int> >::vector(unsigned long, std::allocator<int> co | 1.58e-01 0.0% | |
| 179: std::vector<int, std::allocator<int> >::vector(unsigned long, int const&, std::allocat | 1.49e-01 0.0% | |
| 173: setup_process_info(int, int, int, int) | 4.48e-02 0.0% | |
| 176: [I] std::vector<int, std::allocator<int> >::~vector() | 3.05e-02 0.0% | |
| 179: [I] std::vector<int, std::allocator<int> >::~vector() | 2.44e-02 0.0% | |

Figure 7: HPCToolkit Viewer results for P=32 N=7500 c=2

From the HPC Traceviewer, we can see similar shapes like in jumpshoot4 with much more details regarding call stack(Figure 8). Again we can see a distribution of submatrices at the beginning, first computation(grey colored), collective circular shift, the second computation, and reduction at the end.

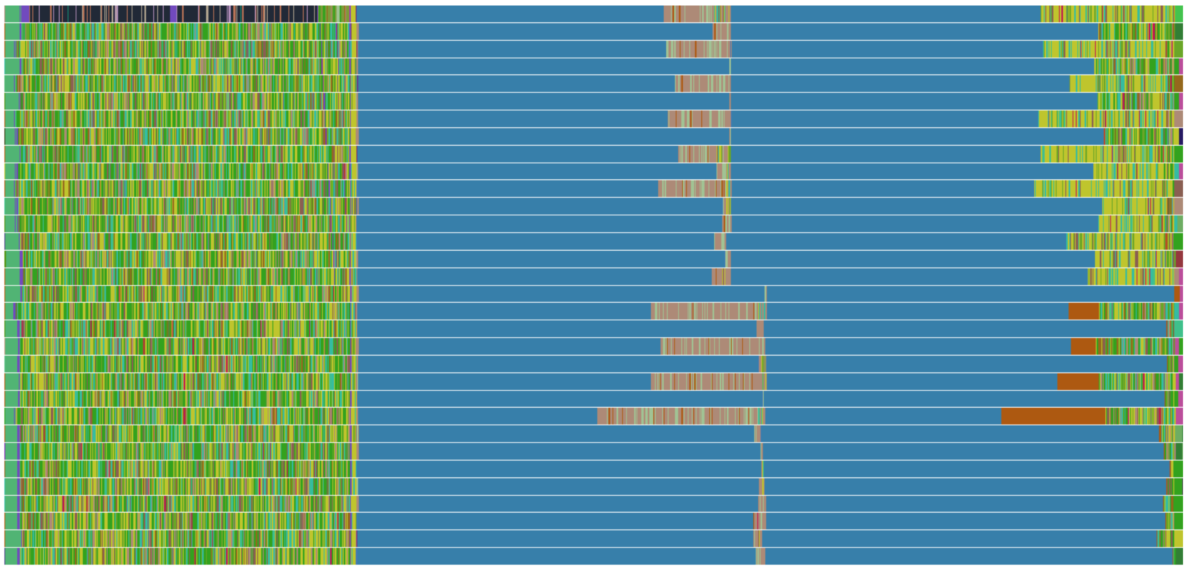


Figure 8: HPCToolkit TraceViewer results P=32 N=7500 c=2

4. **Conclusion** This project implements and analyses the MPI implementation of 2.5D matrix-multiplication. In this report, we showed the 2.5D algorithm, its implementation in the MPI parallel model together with results and visual representation from Jumpshot4 and HPCtoolkit.

   From shown results, we can conclude that tuning parameter c does not particularly accelerate the algorithm on the given architecture. The speed of the application mostly depends on the number of processors and how MPI ranks are scheduled across them. If only one processor is delayed due to slower hardware or conjecture for CPU resources all other threads will stall due to collective shifts that happen before each computation step. Using Jumpshot4 and HPCTraceviewer tool enabled us to see this behavior and understand the cause of performance bottlenecks.

   MPI programming model enables convenient development of distributed applications with many build in details of each primitive that a developer needs to know if we want to achieve the best performance. The MPI Runtimes provides a convenient abstraction for collective communication which includes often-used patterns like distribution on a matrix to block-submatrix and global circular shift for a communicator. However, MPI runtime constitutes about 50% of execution time, and therefore is important to understand it under the hood.

## A Apendix

| Proc | Specifications |
|---|---|
| Architecture: | x86_64 |
| CPU op-mode(s): | 32-bit, 64-bit |
| Byte Order: | Little Endian |
| CPU(s): | 48 |
| On-line CPU(s) list: | 0-47 |
| Thread(s) per core: | 2 |
| Core(s) per socket: | 12 |
| Socket(s): | 2 |
| NUMA node(s): | 2 |
| Model name: | Intel-Xeon(R) CPU E5-2650 v4 @ 2.20GHz |
| L1d cache: | 32K |
| L1i cache: | 32K |
| L2 cache: | 256K |
| L3 cache: | 30720K |
| NUMA node0 CPU(s): | 0-11,24-35 |
| NUMA node1 CPU(s): | 12-23,36-47 |

Table 1: Architecture

## References

[1] Edgar Solomonik and James Demmel, Communication-optimal parallel 2.5D matrix multiplication andLU factorization algorithms, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.pdf