Assignment 3: 2.5D Matrix Multiplication

Aniket Gattani

Program Overview:

The program implements the 2.5D algorithm in the following manner:

1. Create a 3D matrix of size $\sqrt{p/c} * \sqrt{p/c} * c$ grid using MPI routine
2. Create communicators along i, j, k directions of the grid to communicate
3. Initialize buffers A,B and C of size $n/\sqrt{p/c} * n/\sqrt{p/c}$ using a random number seed using the rank of each processor in the front face of the grid.
4. Perform the modified Cannon's algorithm - 1 Initial Shift and $\sqrt{p/c^3} - 1$ shifts for each processor layer.
5. Reduce the sum of the matrices across all layers to the front face of the grid
6. Print execution time and maximum wait time (max total time spent by a process amongst all while waiting to send or receive data)
7. Verify the results, if required, by performing Cannon's along the front face of the grid. Each block verifies its result and the overall result from each is reduced to rank 0.

Bandwidth and Latency Lower Bounds:

It has been proven that the lower bounds for bandwidth and latency are:

- W = Ω (#arithmetic operations / $\sqrt{M}$)
- S = Ω (#arithmetic operations / $M^{3/2}$)
- #arithmetic operations = $(1 + \sqrt{(p/c^3)} - 1) * (n/\sqrt{p/c})^3 = n^3/p$

  The matrix multiplication occurs once immediately after the Cannon's initial shift and $\sqrt{(p/c^3)} - 1$ times as iterations in Cannon's algorithm for each processor. Every matrix multiplication comprises of $(n/\sqrt{p/c})^3$ because the matrix is of size $(n/\sqrt{p/c})$

- M = cn^2/p Since M is the maximum amount of memory, utilized by a processor at any point during algorithm execution.
- So, $W_{2.5D} = \Omega (n^2/\sqrt{pc})$ and $S_{2.5D} = \Omega (\sqrt{(p/c^3)})$

As per the execution of the algorithm,

1. Initial Broadcast along k direction for A, B: We send 2 messages of size $(n/\sqrt{p/c})$

   W = 2*(cn^2/p) and S = 2* log(c)
2. Initial Cannon's shift: We send and receive both A and B according to the corresponding shifted ranks. Only 2 messages are passed per processor.

   W = 2*(cn^2/p) and S = 2
3. Cannon's iterations: Both A and B are exchanged in every iteration

   W = $(\sqrt{(p/c^3)} - 1) * 2 * (cn^2/p)$ and S = $(\sqrt{(p/c^3)} - 1) * 2$

4. Finally we reduce when c >1 along k direction. This is same as broadcast except that there is only 1 message reduced i.e. the matrix C

   W = 2*(cn^2/p) and S = log(p)

Summing all of this, the total bandwidth and latency are:

$W_{2.5D} = 3*(cn^2/p) + 2*n^2/\sqrt{pc} = O(n^2/\sqrt{pc})$ because $c <= p^{1/3}$

$S_{2.5D} = O(\sqrt{p/c^3} + log(c))$

Thus, the algorithm achieves theoretical lower bounds and strong scaling when $\sqrt{p/c^3} >> log(c)$
From this we also realize that the lower bounds are achieved only when available memory >= $3 * (cn^2/p) * sizeOf(int)$

Collective Communication:

The program makes use of collective communication along each of the dimensions.

1. First of all, all nodes in the front face of the cube(k==0) broadcast the values of Aij, Bij along the k direction. Hence, all nodes in the same (i,j) should be in a group (kcomm)
2. We first perform the circular shift as in Cannon's algorithm using collective groups. For this all nodes with similar (i,k) and similar (j,k) should be in communicative groups, icomm and jcomm.
3. As we are performing Cannon's iterations in every kth layer, we again make use of the icomm and jcomm ($\sqrt{(p/c^3)} - 1$) times.
4. Finally, we reduce the matrices from different layers into the front facing ones and perform a MPI_SUM of the values. For this we again use the kcomm communication group. This reduction is in place so as to avoid multiple buffers and save on memory.

Allowed Processor Counts:

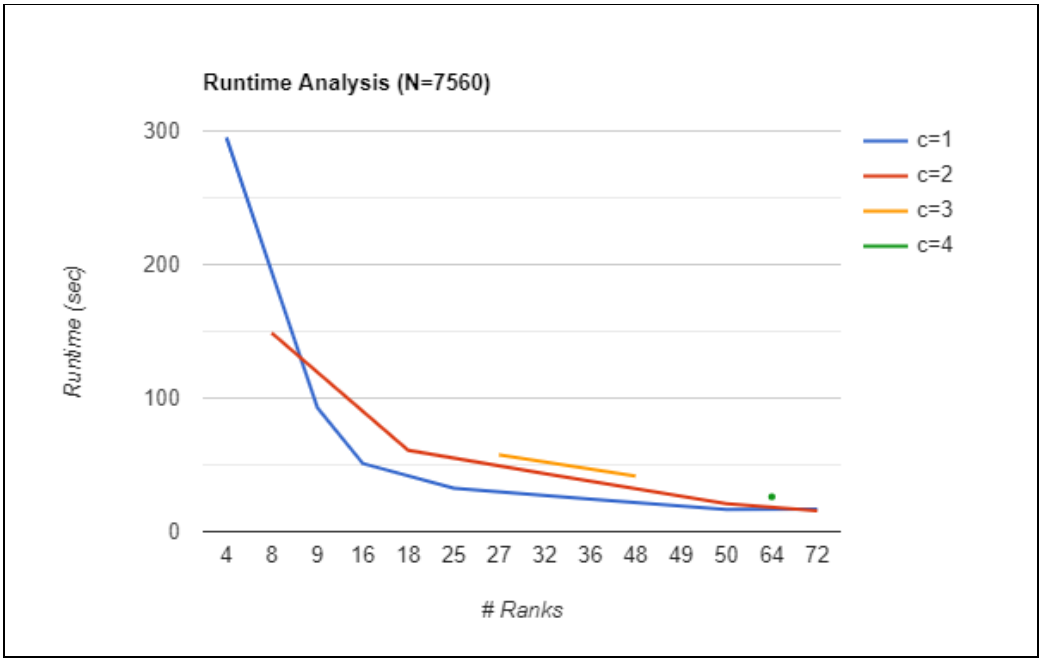
First we should understand what are the allowed values of c and how they can impact the performance.

1. The 2.5D matrix lays out the processors on a $\sqrt{p/c}$ * $\sqrt{p/c}$ * c grid. Hence, $\sqrt{p/c}$ should be an integer. This implies that (p/c) should be a perfect square. Also c < $p^{1/3}$. This means that the bounds of c are between [1 , $p^{1/3}$].
2. For our use case, NOTS only provides core counts from 1-72. Now we need to compare Cannon's algorithm with the 2.5D matrix. Now for Cannon's c=1. This means that p should be a perfect square. So for c=1 only perfect squares are allowed.
3. We are also dividing our data in square blocks of size $n/\sqrt{p/c} * n/\sqrt{p/c}$. This means that n should be divisible by $\sqrt{p/c}$. Hence, I have taken n = 7560. This allows us to run Cannon's algorithm for all values of p (from $\sqrt{p} = 1$ *to* $\sqrt{p}$ = 64 when c=1.
4. From the above constraints:
   a. For c=1, possible p = 1, 4, 9, 16, 25, 49, 64
   b. For c=2, possible p = 8, 18, 32, 50, 72
   c. For c=3, possible p = 27, 48
   d. For c=4, possible p = 64

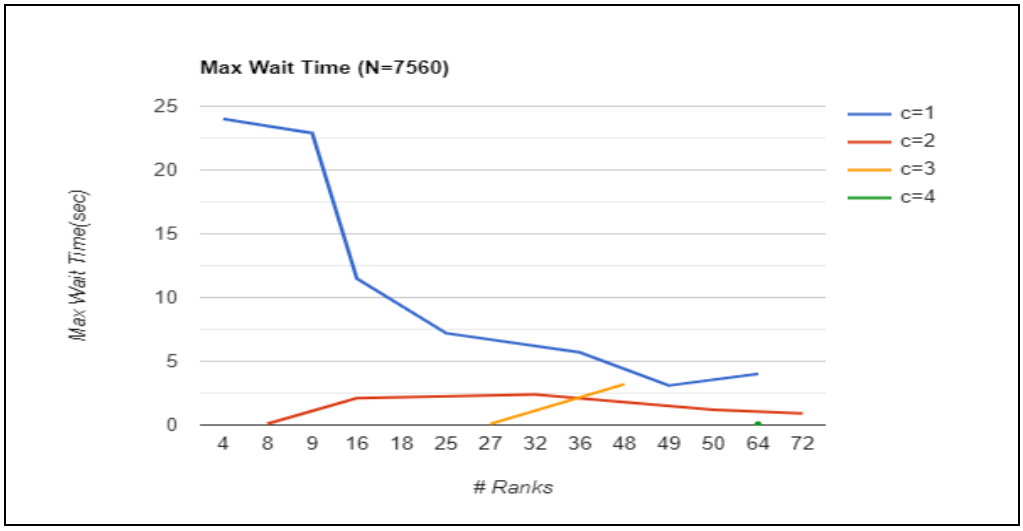
5. From the above latency and bandwidth lower bound analysis we know that available memory >= $3 * (cn^2/p_{min}) * sizeOf(int)$. On NOTS, the processors usually have roughly around 30000 KB (roughly 30MB for 24 core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz) of caches available. This means that

   $(p_{min} / c) = (3*(7560*7560)*4)/(30000*1024) = 23$ approx

6. This means that when (p_{min} / c >= 23) the memory is sufficient to hold all the 3 buffers and so computation is no longer the bottleneck but bandwidth and latency is.



(Fig. 1) Runtime (in sec) vs Number of MPI ranks (N=7560)



(Fig. 2) Max Wait Time(in sec) vs Number of MPI ranks (N=7560)

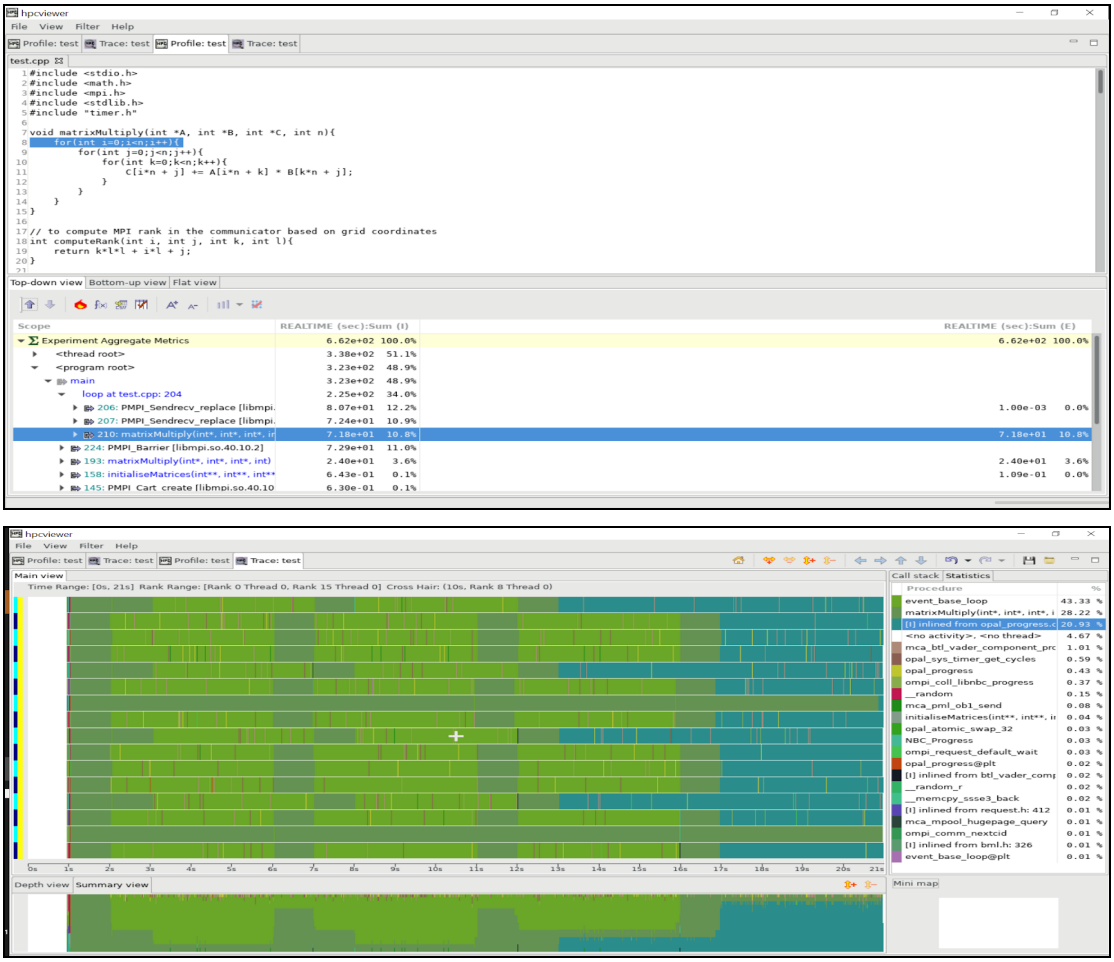
Performance Runs:

1. From Fig.1, for c=1, it is evident how increasing the number of processors decreases the runtime. When p < 23 (lower bound on computation), we almost observe perfect strong scaling because the memory isn't sufficient to hold all 3 matrices A,B and C . With increasing p, the processors require lesser DRAM fetches and perform better.
   Another interesting fact here is that after p >= 23,, we do not observe significant improvement. This is because after p >= 23, bandwidth decreases but latency increases by the same factor $\sqrt{p}$. Minor differences are also due to the fact that the NOTS uses NUMA nodes and different scheduling of the processes can lead to varying results (I observed when using 16 core vs 24 core architectures, latency cost varies by a factor of 2 sometimes). So the overall results will vary according to the network topology.

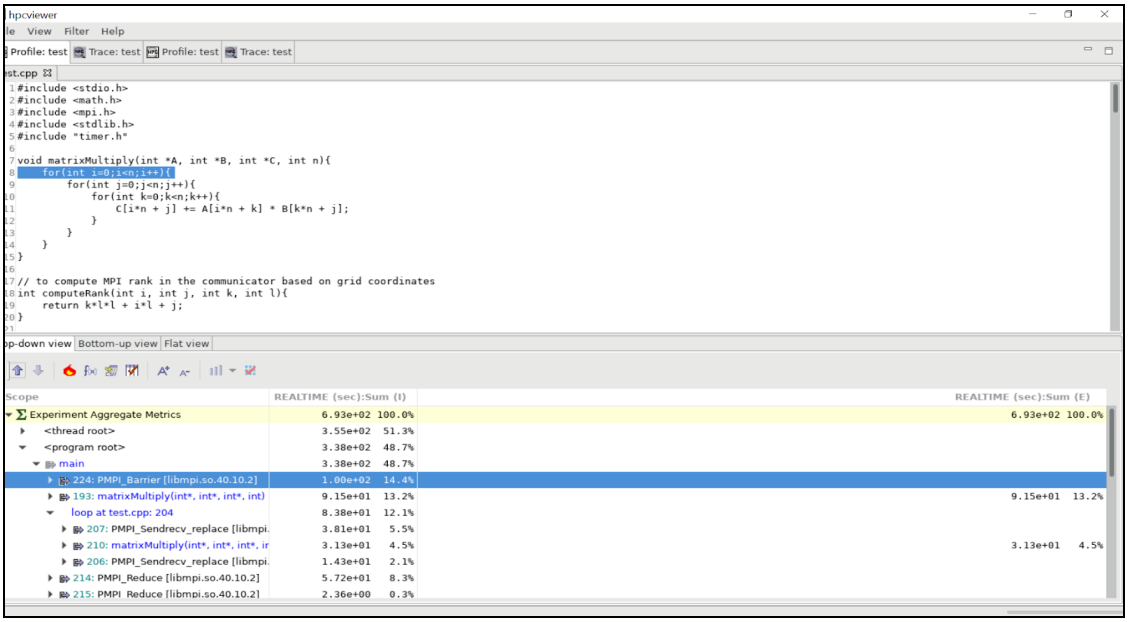
2. With c>=2, increasing processor counts helps us in faster computation and that is evident from the graph. One interesting part here is the cutdown in terms of latency costs as opposed to when c=1. Although performance wise, this is not beneficial because computation lower bounds are not achieved except for the data point (p=72, c=2 and p=50, c=2). We can observe a dip here in runtime due to the fact that computation lower bound is achieved p/c >= 23. Latency cost when (p=49,c=1) is cut down to 22% using (p=50,c=2) . This shows us that we definitely get the benefit of the 2.5D algorithm when the Cannon's algorithm is bounded by latency and bandwidth

3. One interesting fact here is the data point (p=64, c=4). Although p/c < 23, the runtime and the latency costs are significantly smaller than when (p=64, c=1) by 98%. This shows us how expensive communications and using the 2.5D has a significant reduction in the runtime.

Time Analysis:



(Fig 3 & 4) Execution and Trace profile when N=3600, p=16, c=1. Time = 21s



(Fig 5 & 6) Execution and Trace profile when N=3600, p=18, c=2. Time = 19s

The above screenshots are taken for the parameters (N=3600, p=16, c=1) and (N=3600, p=18, c=2). As is evident from the first one, only 28% of our time is spent in computation and the rest is spent waiting to send or receive events as is specified by 'event_base_loop'.

As opposed to when c=2, we see that matrix multiplication is the major contributor (34%). Some of the time is also spent here at MPI_Barrier. This is because the number of Cannon's iterations are not divided evenly between different processor layers ($\sqrt{p/c}\ \%\ c\ !=0$). So the first layer takes 0 iterations whereas the inner one takes 1 iteration.

However, the important point to note is that reducing the communication actually reduces the time spent in moving the data around and waiting for it. Another evident fact here is the profiles of these runtimes. The first runtime has 4 short computations followed and each is followed by a long wait time. When c=2, the computations are spread out and consume a larger time but the communications are reduced and we see only 2 long wait times. These computations are not memory bound since it is sufficient to hold all 3 buffers. So lower bounds for bandwidth and latency do apply here and hence we see the reduction.

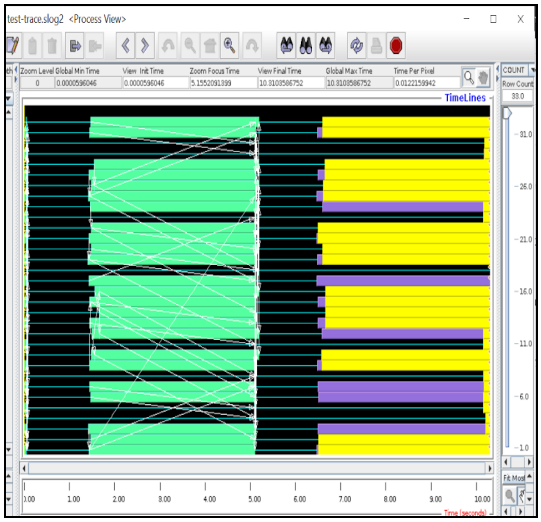


Fig 7. Process View of Jumpshot

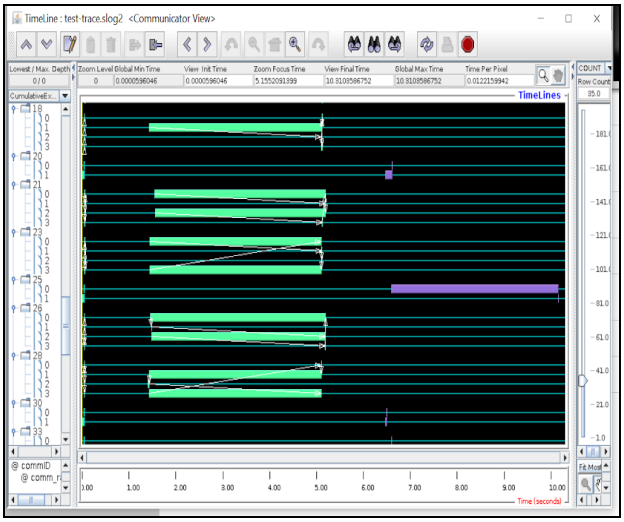


Fig. 8 Communicator view of Jumpshot

Jumpshot Analysis:

1. The snapshot is carried out for the parameters: (N=3600, c=2 and p=32). Each processor uses collective communications and is visualized using jumpshot
2. Figure 7 shows all the collective communications effectively:
    a. The initial broadcast by the processors (in light blue) in the front face of the grid is very small and constitutes very short time in the snapshot. This is because the broadcast is of the order log(c).
    b. The next send and receives by the processors for performing Cannon's algorithm (Greenish blue)
    c. MPI reduce along the front face of the grid (Purple)
    d. MPI Barrier to finally synchronize all processes (Yellow)
3. Finally I checked the collective communications in depth (Fig. 8)
    a. MPI_Bcast involves only 2 processes because c=2 which is visible in the figure with only 2 processes communicating in the kth direction. (communicator rank = 25)
    b. Cannon's shifts involve the cyclic shifts of all processes along ith and jth directions and involves 4 processes which are visible from the figure. (communicator rank = 26)
    c. Finally the reduce occur over the same communicator as the broadcast
    d. Finally the barrier is called using the MPI Comm World (communicator = 0) and involves all 32 processes.