HPC, MPI task report

Jan Kwiatkowski

30.06.2024

Solution structure and used MPI optimizations

Elements of my solution, with the description of used MPI optimizations:

- Communication.
  In my solution I only use vectors (not arrays). Therefore every time the vector is sent I use two MPI send calls. The first one sends one integer - the size of the vector. The second one sends the vector itself.

- Spreading input data.
  Process 0 reads data from files and then sends it to other processes. To optimize this step I've used asynchronous commands *MPI_Isend* and *MPI_Irecv*.

- Summa2D.
  For each layer Summa2D has to be computed. Each layer has a square process grid assigned to it. These processes are split into different communication channels. Such channel is created per each row and column of the process grid. This solution allows the processes to use asynchronous broadcast (*MPI_Ibcast*) inside their row and column channels. It is much faster than sending data between each pair of processes inside the row / column with *MPI_Isend*.

- ColSplit and AllToAll.
  After division of the initial matrix into layers and computing Summa2D within each layer, ColSplit and AllToAll steps must be performed. Firstly, each process sorts its result matrix from Summa2D in column-major order. Secondly AllToAll step is performed by using *MPI_Alltoallv* and creating a new communication channel per each fiber. This MPI function gives a visible speedup compared to naive communication.

- Computing the answer (-g flag).
  In order to compute the answer each process has to count how many values bigger than g they obtained. Then the results from each process must be summed up. In order to achieve that in a clear and fast way, the *MPI_Reduce* with *MPI_SUM* option is used.

- Printing the matrix (-v flag).
  Despite the fact that this option will only be used for testing I've decided to implement it in a fast way using *MPI_Gatherv* instead of slower synchronous communication between process 0 and each other process.

- Summary.
  In each possible fragment of the code the most suitable MPI command was used. The most important optimization was splitting the processes into dedicated communication channels which allowed me to use *MPI_Ibcast* and *MPI_Alltoallv*.

Numerical intensity

Let $\mathrm{nnz}(M)$ - number of non-zero elements in M, p - number of processes, l - number of layers, L - latency, B - bandwidth. I will estimate the intensity of computation (note that it does not consider the part of assigning the data to processes). Assuming that the non-zero elements will divide roughly evenly between processes we can compute the estimation of the numerical intensity.

As for the computational complexity by $flops$ we denote the number of multiplications needed for computing AB (like in the given paper). We can see that no two identical multiplications occur in Local-Multiply, and then some computation is needed for merging layers and fibers. The estimated numerical complexity is $O\Big((flops/p) \cdot (\log(p/l) + \log l)\Big) = O\Big((flops/p) \cdot \log p\Big)$.

The communication complexity is:

- each process gets data of size $(\mathrm{nnz}(A) + \mathrm{nnz}(B))/p$;
- in Summa2D there are $\sqrt{p/l}$ stages of broadcast, so we get

$$\begin{aligned}\text{latency } &= O\Big(L \cdot \log(p/l) \cdot \sqrt{p/l}\Big)\\ \text{bandwidth } &= O\Big(B \cdot (\mathrm{nnz}(A) + \mathrm{nnz}(B))/p) \cdot \sqrt{p/l}\Big) = O\Big(B \cdot (\mathrm{nnz}(A) + \mathrm{nnz}(B))/\sqrt{pl}\Big)\end{aligned}$$

  There is a logarithm in latency because of usage of binary trees in broadcast (like shown in the lecture).

- in AllToAll each process sends $O(flops/p)$ data to l processes in total so

$$\begin{aligned}\text{latency } &= O\Big(L \cdot l\Big)\\ \text{bandwidth } &= O\Big(B \cdot flops/p\Big)\end{aligned}$$

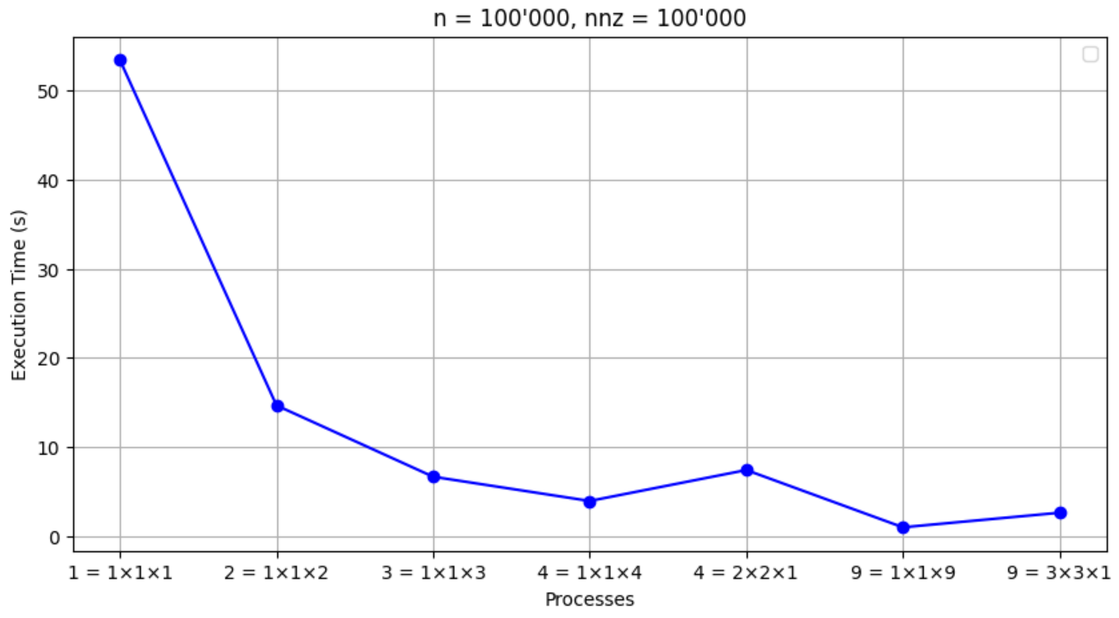
With those values and given p, l, L, B we can easily estimate the total numerical complexity.

Benchmarks

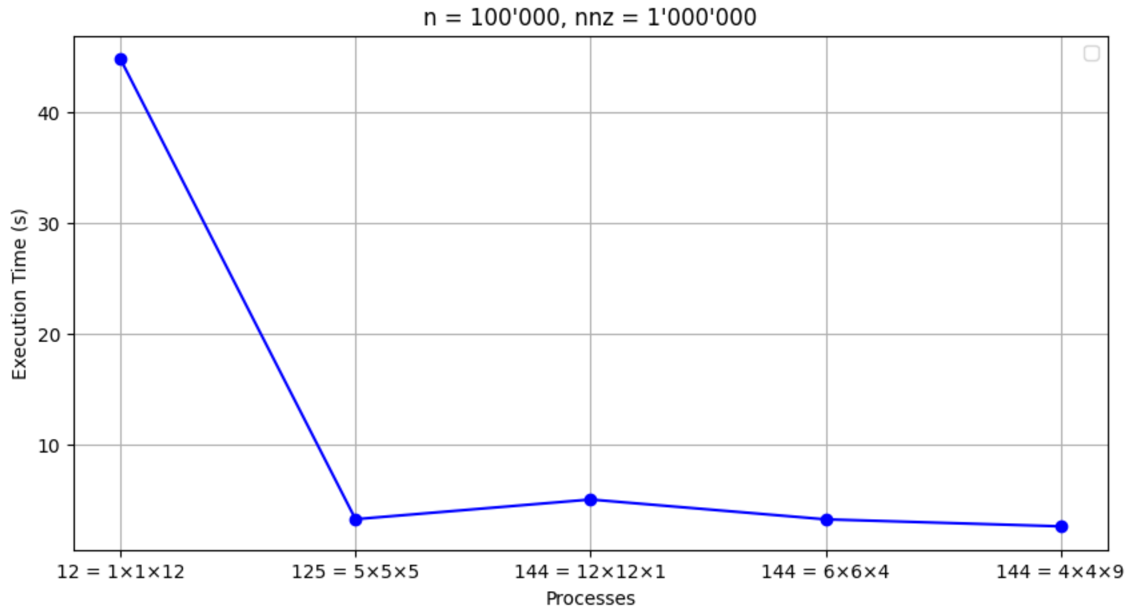
Here are the charts with some of the benchmarks I've conducted. Blue line present the average runtime of Summa3D. Let n be the size of the matrix side and nnz the number of non-zero elements. The notation $p_r \times p_c \times l$ denotes the situation with l layers of processes grids, each grid made of $p_r \times p_c$ processes. If $l = 1$ then the 2d variant was called.

For example the notation $250 = 5 \times 5 \times 10$ means a situation where 250 processes were divided into 10 layers, each creating a grid of processes of size 5×5.

In each execution tasks per node value was as close as possible (but not greater) than 24.

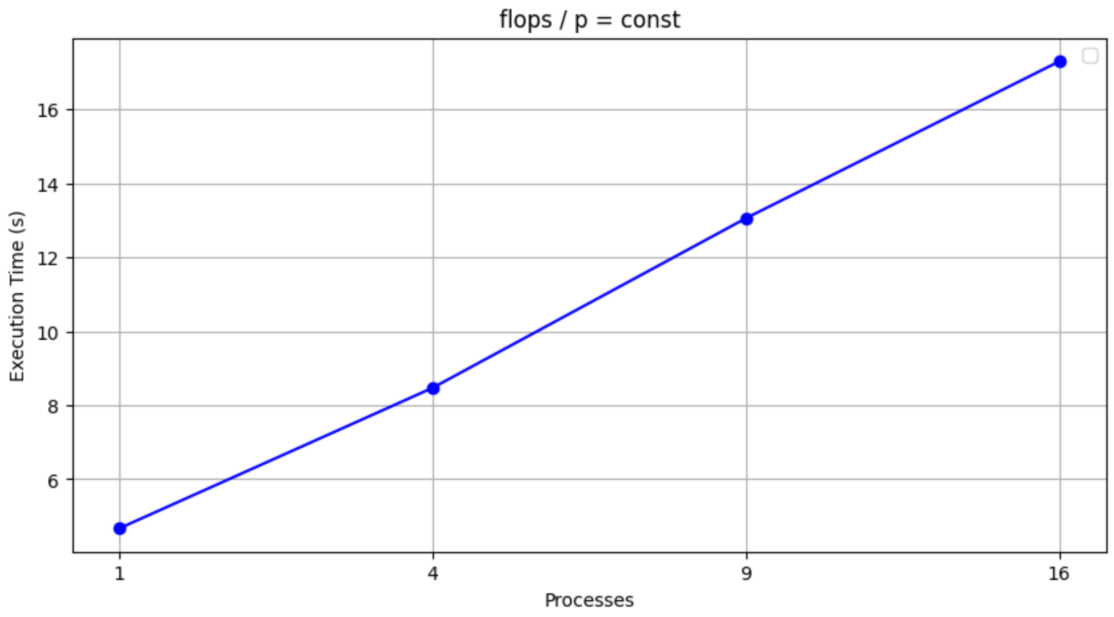


Drawing 1: The first benchmark.



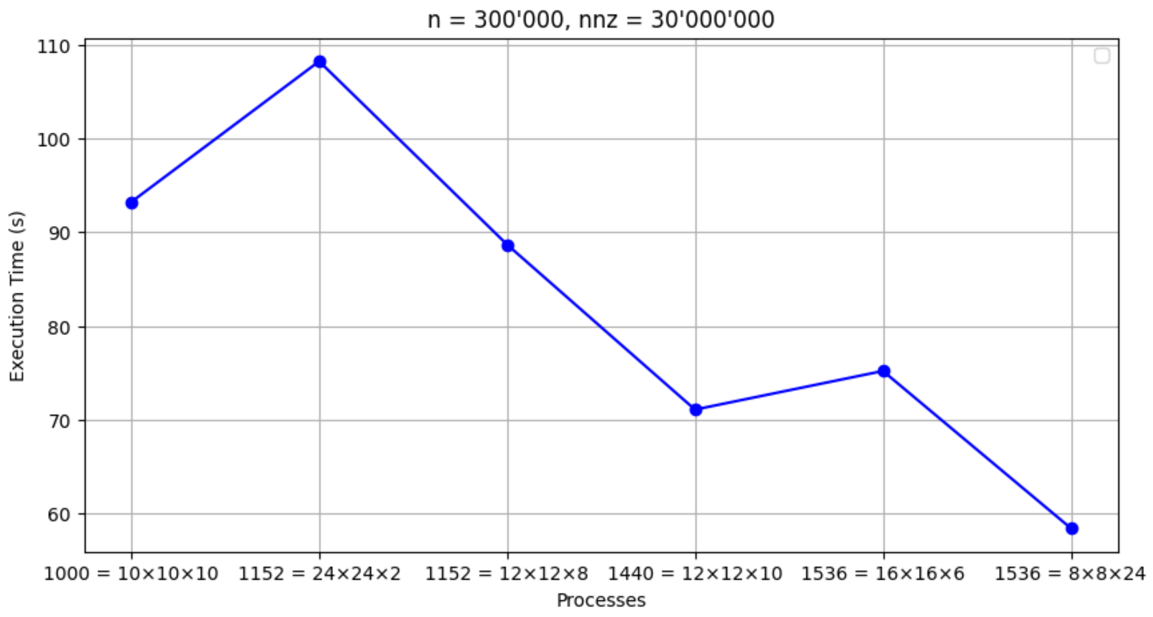
Drawing 2: The second benchmark.

Drawings 1 and 2 shows, that the problem is strongly scalable (i. e. we can see $+/-$ linear speedup when working with more processes on the same problem size) **however** one have to be cautious while choosing the exact values of p_r, p_c and l. In particular it seems like the program is faster if there are more layers and less processes in the grid.



Drawing 3: The third benchmark.

Drawing 3 shows that it is not so nice for weak scalability. Here the work per process remains the same, however the time needed to compute the resulting matrix grows significantly with the number of processes. In this example for one process there is around 10^8 operations to be performed.



Drawing 4: The fourth benchmark.

Drawing 4 shows some more execution times for a very big matrix. Here the time of splitting the input between processes is around 38 seconds in each considered variant (the rest of the time is proper computation).