ECE 382N: Lab 1 Report

Tianda Huang
Sayam Sethi

25 September 2023

Contents

1 MPI Implementation

We build upon the provided implementation of the `test_mm` code. The entire matrix (of size $m \times m$) is divided into $p \times p$ square blocks each of size $\frac{m}{p} \times \frac{m}{p}$. Processor i obtains the row of this smaller matrix. We then compute the matrix multiplication as follows,

$$\begin{aligned}
\mathbf{A} \times \mathbf{B} &= \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & \cdots & B_{pp} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1p} \end{pmatrix} \times \begin{pmatrix} B_{11} \\ B_{21} \\ \vdots \\ B_{p1} \end{pmatrix} & \cdots & \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1p} \end{pmatrix} \times \begin{pmatrix} B_{1p} \\ B_{2p} \\ \vdots \\ B_{pp} \end{pmatrix} \\ \begin{pmatrix} A_{21} & A_{22} & \cdots & A_{2p} \end{pmatrix} \times \begin{pmatrix} B_{11} \\ B_{21} \\ \vdots \\ B_{p1} \end{pmatrix} & \cdots & \begin{pmatrix} A_{21} & A_{22} & \cdots & A_{2p} \end{pmatrix} \times \begin{pmatrix} B_{1p} \\ B_{2p} \\ \vdots \\ B_{pp} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ \begin{pmatrix} A_{p1} & A_{p2} & \cdots & A_{pp} \end{pmatrix} \times \begin{pmatrix} B_{11} \\ B_{21} \\ \vdots \\ B_{p1} \end{pmatrix} & \cdots & \begin{pmatrix} A_{p1} & A_{p2} & \cdots & A_{pp} \end{pmatrix} \times \begin{pmatrix} B_{1p} \\ B_{2p} \\ \vdots \\ B_{pp} \end{pmatrix} \end{pmatrix} \quad (1) \\
&= \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pp} \end{pmatrix}
\end{aligned}$$

Each processor i computes the resultant matrix for row i. This requires each processor to have the entire matrix **B** but since the calculation for each block will be done sequentially, the processor needs to only store the j^{th} column at any time instant. Ideally, the processor just needs to store the k^{th} block of the j^{th} column when computing the k^{th} block of the i^{th} row. However, this would lead to larger message passing overheads. Each processor would need to broadcast all of its blocks to all cores all at once since different blocks will be required by different processors at different times. Instead, we use `MPI_Allgather` and gather the entire column at once for all processors. This reduces the message passing complexity as well since MPI takes the topology into account and all cores perform message passing almost simultaneously. As a result, this approach additionally reduces the stress on send and receive buffers.

Once we are done with the matrix computation, each processor performs the sum of its own row, and then we perform a `MPI_Reduce` to obtain the final sum. In the debug mode, each processor sends its matrix to the `root` processor for printing. We used this approach since `MPI_Barrier` wasn't working across different nodes.

2 Cilk Implementation

For Cilk we follow a similar blocking style. We store all the input matrices and output matrix on the "root" node. Then, obtaining the number of Cilk workers with $p =$`__cilkrts_get_nworkers`, we split the matrix into $p \times p$ blocks of size $\frac{m}{p} \times \frac{m}{p}$.

Each block of the result is computed by a dot product of a block-row and block-column; each dot product computed as a `cilk_spawn` task.

After the matrix computation, we compute the sum of all elements by separating into block-rows. Each block-row is summed by a separate `cilk_spawn` task and the resulting partial sums are reduced into a total sum by the root task.

The motivation for $p \times p$ blocking is to create enough small tasks such that each processor is busy, while reducing the `cilk_spawn` overhead by keeping each task as large as possible.

3 Work Division

The work was split almost equally between both the students. We had a back-and-forth division of labour where one fixed the code and added additional functionality and then the other student would add more functionality and fix any bugs that might have been introduced.