HW3: Implement Distributed Matrix Multiplication using MPI, OpenMP, and CUDA

CSCE 654: Supercomputing
Due: `Monday, Oct 6 @ 11:59pm CT`
Total Points: 50

Academic Integrity & AI Tools

- **No LLM-generated code.** You may *not* use large language models (LLMs) to generate any code or scripts for this assignment (including source files, headers, build files, sbatch scripts, plotting scripts, etc.). If we determine that the code was generated by an LLM, the assignment will receive a **zero**.
- **Permitted use of LLMs.** You may use LLMs *only* to help you understand concepts (e.g., CUDA syntax, cuBLAS interfaces, algorithmic ideas) or to improve the clarity/grammar of your written report.
- **No copying from others or the internet.** Do not copy code from other students, prior solutions, or online sources (blogs, GitHub, Stack Overflow, etc.). We use automated and manual plagiarism checks across submissions and public sources.
- **Consequences.** Suspected violations will receive a zero and may be referred under university academic-integrity procedures.
- **Late policy.** Submissions lose **20%** of the total possible points for each late day, **except** when extensions are granted under university policy.

Objective

In this homework you will implement distributed dense matrix multiplication using the **SUMMA (Scalable Universal Matrix Multiplication Algorithm)**. The computation will be distributed across a square process grid. Local matrix multiplications will be performed using both CPU (OpenMP) and GPU (CUDA) implementations (you can use the code developed in homework 1 and 2).

Learning Outcomes

By the end of this assignment, you will:

1. Understand how to distribute global matrices across a 2D process grid.
2. Implement the SUMMA algorithm for distributed matrix multiplication.
3. Offload local matrix multiplications to GPUs.
4. Verify distributed results by reconstructing the global matrix.
5. Report performance and correctness.

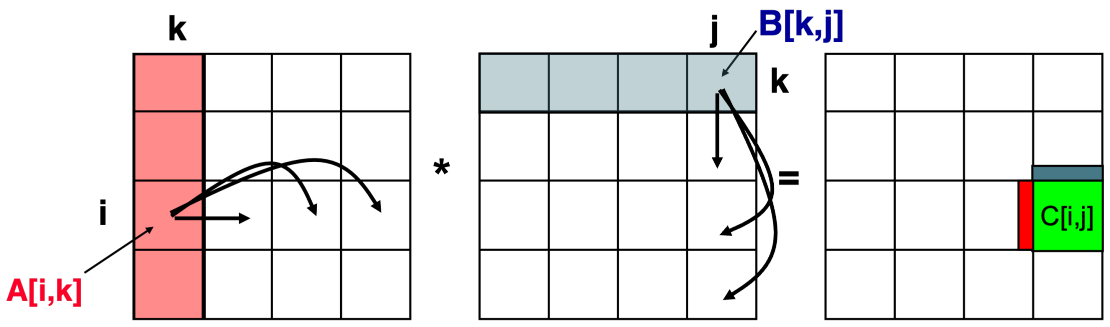

Figure 1: Example of a step in SUMMA. Note that all processes in the shaded regions broadcast simultaneously.

Background: The SUMMA Algorithm

SUMMA is a distributed-memory algorithm for multiplying two dense matrices $C = A \times B$.

Key Idea

- Assume the number of processes P is a square number. We arrange processes as a square $\sqrt{P} \times \sqrt{P}$ process grid. Let P_{ij} be the rank of the process at the ith row and jth column of the grid.
- For simplicity, we assume that A, B, and C are all $N \times N$ square matrix.
- P_{ij} owns local blocks A_{ij}, B_{ij}, C_{ij}. We assume that rows and columns are uniformly divided (as much as possible). For example, if $N = 16$ and $P = 4$, each process will get a 2×2 submatrix.
- Multiplication can be expressed as

$$C = \sum_{k=0}^{\sqrt{P}-1} A[:, k] \times B[k, :].$$

- At step k:
  1. Broadcast A_{ik} across process row i.
  2. Broadcast B_{kj} across process column j.
  3. Compute partial product locally and accumulate into C_{ij}.

Pseudocode

```
for k = 0 to P-1:
    Broadcast A_local(i,k) along process row i
    Broadcast B_local(k,j) along process column j
    C_local(i,j) += A_local(i,k) × B_local(k,j)
```

1 Problem 1 (10 points)

Extend the provided `LocalMatrix` class to distribute global matrices A, B across the process grid. You are already given code to create a square process grid ($\sqrt{P} \times \sqrt{P}$ processes). Assume that all matrices are $N \times N$ square matrices.

1. Implement data distribution and ensure your data distribution works when N is not divisible by P. We assume that rows and columns are uniformly divided among processes (as much as possible). For example, if $N = 16$ and $P = 4$, each process will get a 2×2 submatrix. If $N\%P \neq 0$, some process will get one extra row and column, similar to the MPI programming exercise. Store the local block in row-major order.
2. Initialize global matrices A and B using simple deterministic functions (e.g., $A(i, j) = i + j$, $B(i, j) = i - j$).

2 Problem 2 (10 points)

Implement SUMMA using MPI broadcasts and local DGEMM with OpenMP. For local DGEMM, you can use the code developed in Homework 1. You will need to write a function to add matrices to update the output.

3 Problem 3 (10 points)

Implement SUMMA with local DGEMM running on GPUs (CUDA kernel from Homework 2). You will need to write a function to add matrices to update the output. The distributed code will be the same as in Problem 2.

4 Problem 4 (10 points)

Develop a correctness checker as follows:

- Gather distributed A, B, C to rank 0.
- Reconstruct the global matrices.
- multiply them with your correct OpenMP code and verify the results.

5 Reporting (10 points)

Submit a report including:

- Screenshots or logs showing then verification results (max error). Only the cpu version is fine. For the verification task, use $N \sim 1000$ and $P = 4$ with 8 threads per process.
- Use $N \sim 5000$ for this experiment. Any other value of N is fine as well. For the CPU code, run experiments with 8 threads per process. Then change the number of processes as follows (P=4, P=16, and P=64). We will only use square processes. Note that NERSC has 128 cores per node for the CPU partition. Show the results in your report and discuss the performance. Submit the corresponding SLURM scripts. Submit the corresponding SLURM scripts as part of your submission.

- For the GPU code, run experiments with 8 threads per process (this does not matter for local multiplication). Use $N \sim 5000$ for this experiment. Any other value of N is fine as well. Then change the number of processes as follows (P=4 and P=16). Note that NERSC has 4 GPUs per node. Use 1 MPI rank per GPU. Show the results in your report and discuss the performance. Submit the corresponding SLURM scripts as part of your submission. If necessary, you may change your GPU implementations from HW2.