Spring 2025

CS 133 Lab 2

CPU w/ MPI: General Matrix Multiplication (GEMM)

Due Date: 5/1/25 2:00PM PST

Description

Your task is to parallelize general matrix multiplication (GEMM) with MPI based on the sequential implementation provided in Lab 1. Specifically, for matrix multiplication $C[I][J] = A[I][K] \times B[K][J]$, you will need to implement two functions using MPI:

```
void GemmParallelBlocked(const float a[kI][kK], const float b[kK][kJ], float c[kI][kJ]);
```

You should select the block size with the best performance when you submit this lab. You are welcome to add any other optimization to further increase the performance. *However, you must not use any OpenMP pragma in your code.* Please describe every optimization you performed in the report.

You can assume the matrix sizes are multiples of 1024 (e.g. 1024x2048 (A) * 2048x4096 (B) = 1024x4096 (C)). You can assume the number of processors is a power of 2.

Caution: All three matrices are created and initialized ***ONLY*** at processor 0 (see `lib/main.cpp` for details). That means your code needs to explicitly send (parts of) matrices `a` and `b` from processor 0 to the other processors. After the computation, processor 0 should collect the data from all processors and store it in matrix `c`. The correctness and performance will be evaluated at processor 0.

How-To

Build and Test GEMM

We have prepared the starter kit for you at BruinLearn/file/labs/lab2.zip. Log in to your AWS instance or use the terminal of your own computer to run the following commands:

```
cd lab2
./setup.sh
make gemm
./gemm
```

It should run but show “Your answer is INCORRECT!”. Currently, we only provide support for Ubuntu. If you are running another operating system, please replace “`./setup.sh`” with the commands to install **MPICH** on your operating system (e.g., `nix-shell -p mpich gcc boost` if you are using Nix on macOS) or install an Ubuntu virtual machine.

Create Your Own GEMM with MPI

If you have successfully built and run the baseline, you can start to create your own GEMM kernel. The provided starter kit will generate the test data and verify your results with a black-box implementation.

Your task is to implement an MPI parallel version of blocked GEMM. Your program could be based on your parallelization from Lab 1. You should edit `mpi.cpp` for this task.

To test your own implementation of GEMM:

```
make [np=8] # use 8 processors
```

If you see something similar to the following message, your implementation is incorrect.

```
Diff: 1.04936e+06
Your answer is INCORRECT!
```

Your code's performance will be shown after `Perf:` in GFLOPS.

Create an AWS Instance for Performance Experiments

Please refer to the discussion section slides and create an `m5.2xlarge` instance with Ubuntu 22.04 AMI.

Tips

- We will use `m5.2xlarge` instances for grading.
- If you develop on AWS:
    - To resume a session in case you lose your connection, you can run `screen` after login. You can recover your session with `screen -DRR`.
    - You should ***stop*** your AWS instance if you are going back and resume your work in a few hours or days. Your data will be preserved but you will be charged for the EBS storage.
    - Stopped instances still accumulate small fees due to storage. If you are completely done with your instance, you should therefore terminate it. After termination, ***data and any setup on the instance will be completely lost***.
- We recommend you to develop on your own computer and perform experiments on AWS to save your credits. You can upload your code to AWS with `scp`, `rsync`, or `git`.
- You are recommended to use ***private*** repositories provided by GitHub to back up your code. ***Never put your code in a public repo to avoid potential plagiarism.*** To check in your code to a private GitHub repo, create a repo first.

```
git branch -m upstream
git checkout -b main # skip these two lines if you are reusing the folder in Lab 1
... // your modifications
git add mpi.cpp
git commit -m "lab2: first version"  # change commit message accordingly
# please replace the URL with your own URL
git remote add origin git@github.com:YourGitHubUserName/your-repo-name.git
git push -u origin main
```

- You are recommended to `git add` and `git commit` often so that you can keep track of the history and revert whenever necessary. If you move to an AWS instance to perform the experiments, just `git clone` your repo.
- You can run the sequential GEMM by `make gemm && ./gemm sequential`. Be aware that this is very slow (~10 mins on `m5.2xlarge` instances).
- ***Make sure your code produces <u>correct</u> results!***

Submission

You need to report the performance results of your MPI implementation on an `m5.2xlarge` instance. Please express your performance in GFlops and the speedup compared with the sequential version. In particular, you need to submit a brief report which summarizes:

- Please briefly explain how the data and computation are partitioned among the processors. Also, briefly explain how communication among processors is done.
- Please analyze (theoretically or experimentally) the impact of different communication APIs (e.g. blocking: `MPI_Send` , `MPI_Recv` , buffered blocking: `MPI_Bsend` , non-blocking: `MPI_Isend`, `MPI_Irecv`, etc). Attach code snippets to the report if you verified experimentally. Please choose the APIs that provide the best performance for your final version.
- Please report the performance on three different problem sizes (1024^3, 2048^3, and 4096^3). If you get significantly different throughput numbers for the different sizes, please explain why.
- Please report the scalability of your program and discuss any significant non-linear part of your result. Note that you can, for example, `make np=8` to change the number of processors. Please perform the experiment `np=1, 2, 4, 8, 16, 32`.
- Please discuss how your MPI implementation compares with your OpenMP implementation in Lab 1 in terms of the programming effort and the performance. Explain why you have observed such a difference in performance (Bonus +5).

You also need to submit your optimized kernel code. Do not modify the testbed code `main.c`. Please submit on Gradescope. You will be prompted to enter a Leaderboard name when submitting, please enter the name you want to show to your classmates. You can submit as many times as you want before the deadline to improve your performance. *<u>Same lab 1, you must submit a zip file by copying your `lab2-report.pdf` to the `lab2` directory and `make zip` on the m5.2xlarge instance.</u>* `make zip` will automatically generate `result.log` for you. Your final submission should contain and only contain these files individually:

```
├ mpi.cpp
├ result.log
```

```
└ lab2-report.pdf
```

File `lab2-report.pdf` must be in PDF format exported by any software. Gradescope will automatically extract the files from the zip file and it is expected behavior.

Grading Policy

Your submission will be automatically sent to our AWS Batch queue (m5.2xlarge) for evaluation on Submission Format and Performance. Your last Submission Format score and Performance marks will be final if there is no cheating. The correctness score will only be **partially displayed** before the deadline and may be adjusted during the manual grading. There are other correctness tests of your solution which are not included in the starter kit repo. You can see the performance of other students on the Leaderboard. Although rare, you could request for manual regrade if there is a significant discrepancy between the performance on your instance and Gradescope.

Submission Format (10%)

Points will be deducted if your submission does not comply with the requirement. In the case of missing reports, missing codes, or compilation errors, you will receive 0 for this category.

Correctness (50%)

Please check the correctness of your implementation with different number of processors and problem sizes, including but not limited to 1, 2, and 4, and 1024^3, 2048^3, and 4096^3, respectively. Your code will be inspected -- if it is not an MPI implementation, you will receive 0.

Performance (25%)

Your performance will be evaluated based on the performance of problem size **4096^3** with **np=4**. Performance points will be added only if you have a correct implementation, so please prioritize correctness over performance. Your performance will be evaluated based on the ranges of throughput (GFlops). We will set ranges after evaluating all submissions and assign the points as follows (Ranges A+ and A++ will be defined after all the submissions are made):

- Range A++, better than Range A+ performance: 25 points + 5 points (bonus)
- Range A+, better than Range A performance: 25 points + 3 points (bonus)
- Range A GFlops [125, 165): 25 points
- Range B GFlops [85, 125): 20 points
- Range C GFlops [45, 85): 15 points
- Range D GFlops [5, 45): 10 points
- Range E GFlops [0.19, 5): 5 points
- Lower than range E [0, 0.19): 0 points

Report (15%)

Points may be deducted if your report misses any of the sections described above.

Academic Integrity

All work is to be done individually, and any sources of help are to be explicitly cited. Any instance of academic dishonesty will be promptly reported to the Office of the Dean of Students. Academic dishonesty, includes, but is not limited to, cheating, fabrication, plagiarism, copying code from other students or from the internet, or facilitating academic misconduct. We'll use automated software to identify similar sections between **different student programming assignments**, against **previous students' code**, or against **Internet sources**. Students are not allowed to post the lab solutions on public websites (including GitHub). Please note that any version of your submission must be your own work and will be compared with sources for plagiarism detection.

Late policy: Late submission will be accepted for **24 hours** with a 10% penalty. No late submission will be accepted after that (you lost all points after the late submission time).