Winter 2019

CS 133 Lab 2

CPU w/ MPI: General Matrix Multiplication (GEMM)

Description

Your task is to parallelize general matrix multiplication (GEMM) with MPI based on the sequential implementation we have provided. Specifically, for matrix multiplication $C[I][J] = A[I][K] \times B[K][J]$, you will need to implement a blocked matrix multiplication function 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 may not use any OpenMP pragma in your code.* Please describe every optimization you performed in the report.

Caution: All three matrices are created and initialized ***ONLY*** at processor 0 (see `main.cpp` for details). That means your code need to explicitly send (part of) matrices `a` and `b` from processor 0 to 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 assessed at processor 0.

Tip

- We will use `m5.2xlarge` instances for grading.
- You can assume the matrix sizes are multiples of 1024 (e.g. 1024x2048 * 2048x4096 = 1024x4096)
- You can assume the number of processors is a power of 2

Preparation

Create an AWS Instance

Please refer to the tutorial slides and create an `m5.2xlarge` instance with a Ubuntu 18.04 AMI. Please use your AWS educate classroom account for this lab.

Run Baseline GEMM

We have prepared the host code for you at GitHub. Log in to your instance and run the following commands:

```
git clone https://github.com/UCLA-VAST/cs-133-19w -o upstream
cd cs-133-19w/lab2
./setup.sh
make gemm
./gemm
```

It should finish in a few seconds with a huge error since you haven't implemented anything.

Tips

- To resume a session in case you lose your connection, you can run `screen` after login.
- You can recover your session with `screen -DRR` if you lost your ssh connection.
- You should ***stop*** your 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 for $0.10 per GB per month (with default settings).
- You should ***terminate*** your instance if you are not going to back and resume your work in days or weeks. ***Data on the instance will be lost***.
- You are recommended to use ***private*** repos provided by GitHub. ***Do not put your code in a public repo.***

Create Your Own GEMM with MPI

If you have successfully launched an instance and run the baseline, you can start to create your own GEMM kernel. The provided code will generate the test data and verify your results against a baseline fast GEMM implementation.

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

Tips

- To check in your code to a ***private*** GitHub repo, create a repo first.

```
git branch -m upstream
git checkout -b master
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 master
```

- 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 a new instance, just `git clone` your repo.
- Run `make test` to re-compile and test your code.
- If `make test` fails, it means your code produces wrong result.
- ***Make sure your code produces correct results!***

Submission

You need to report your 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 the 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 report if you verified experimentally. Please choose the APIs that provides the best performance for your final version.

- Please provide the performance on three different problem sizes (1024^3, 2048^3, and 4096^3). If you get significantly different throughput number 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 compare 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 will need to submit your optimized kernel code. Please do not modify or submit the host code. Please submit to CCLE. Please verify the ***correctness*** of your kernel before submission.

Your final submission should be a tarball which contains the following files:

```
<Your UID>.tar.gz
└ <Your UID>
  ├ mpi.cpp
  └ lab2-report.pdf
```

You should make the tarball by copying your `lab2-report.pdf` to the `lab2` directory and running `make tar UID=<Your UID>`. If you made the tarball in other ways, you should put it in the `lab2` directory and check by running `make check UID=<Your UID>`.

Grading Policy

Submission Format (10%)

Points will be deducted if your submission does not comply with the requirement. In case of missing reports, missing codes, or compilation error, you might 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.

Performance (25%)

Your performance will be evaluated based on the performance of problem size **4096^3**, with **np=4**. The performance point will be added only if you have the correct result, so please prioritize the correctness over performance. Your performance will be evaluated based on the ranges of throughput (GFlops). We will set five ranges after evaluating all submissions and assign the points as follows:

- Better than TA's performance: 25 points + 5 points (bonus)
- Range A GFlops: 25 points
- Range B GFlops: 20 points
- Range C GFlops: 15 points
- Range D GFlops: 10 points
- Speed up lower than range D: 5 points
- Slowdown: 0 points

Report (15%)

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