CS410 Assignment 3
Matrix Multiplication using MPI

Date released: March/31/2022.
Due date: April/12/2022 (11:59 pm IST).

In this assignment, you will implement **matrix multiplication** (similar to assignment #2) using **MPI programming model.** The matrix multiplication methods that we have seen in class (column partitioning / row partitioning) have some disadvantages; the block column or block row algorithms do more communication than necessary (recall that communication is minimized when the blocks are square and one-third the size of the cache). Cannon's algorithm is hard to generalize to the case where the matrix dimensions aren't divisible by the square root of the number of processors. So, you will implement and experiment with a more practical matrix multiplication algorithm called the "SUMMA" (**S**calable **U**niversal **M**atrix **M**ultiplication **A**lgorithm), which is the one that is used in practice in Scalapack and related parallel libraries. The basic SUMMA algorithm is simple: reorder the 3 nested loops involved in the computation of matrix C as `C=C+AB`. The following code shows the `'kij'` reordering:

```
C = zeros(n,n);
for k = 1 to n
      for i = 1 to n
            for j = 1 to n
                  C(i,j) = C(i,j) + A(i,k) * B(k,j);
```

If you consider the inner two loops, the order of data access in Matrices B and C is by row. Furthermore, the inner two loops are computing an update to the entire C matrix 'n' times with the product `A(i,k) * B(k,j)`. As the computation of C involves updates using the product of a column of matrix A with a row of matrix B, this is also referred to as rank-1 update to C. The SUMMA algorithm parallelizes the inner two loops and executes the 'k' loop sequentially.

The inner two loops can be imagined to be executed by a 2D grid of processors updating the C matrix in parallel. You can assume that the input matrix sizes are `nxn`. You will have to implement the parallel version of the basic SUMMA algorithm using MPI. In this case, you will partition the nxn matrices (A, B, and C) into square blocks of size *n/sqrt(p) * n/sqrt(p)* and form a 2D grid. Each grid will be assigned to one of the *p* processes.

Teams with at least one PG student:

1. You must produce two code versions for group communications such as broadcast and allreduce.
   a. Using blocking version (e.g., MPI_Bcast) of MPI collectives
   b. Non-blocking (MPI_Ibcast) version of collectives and overlap computation with communication.
2. Compare and contrast their performance behaviors in your report.

Your program should accept one argument – the matrix size n (which initializes matrices with random floating point numbers). This is only for simplicity. In reality, the algorithm works for A[m, n] * B[n, k] = C[m, k] matrix.

You will use the IITDH HPC system to compile and run your code.
You may do the project alone or in a group of two students.
You may discuss ideas but do not share the code with another team.

Distribution and collection of assignments will be managed using **GitHub Classroom**. You can accept the assignment at the following URL: https://classroom.github.com/a/03zQI87q

The repository includes the following starter files:

1. **summa.cpp** which is a starter code taken from here.
   You need to fill in the code wherever you see the "YOUR CODE HERE" comment.
2. **Makefile**, which produces these versions of the code:
   a. summa: which does not validate the correctness of your results.
   b. summa_check: which checks to make sure that the results are correct within a reasonable accuracy. This code uses a single process in the end to validate the code. It runs much slower. Use this version only for checking the correctness of your code. DO NOT use this for collecting the performance data.
   c. **[PG students only]**
      i. summa_nb: which uses non-blocking collective communication APIs.
      ii. Summa_nb_check: like summa_check but for your non-blocking version.
3. **submit.sbatch**, which is a job submission script.
   a. The example script uses 64 MPI processes (#SBATCH -n 64) spread over 2 nodes (#SBATCH --nodes=2), where each node has 32 MPI processes (#SBATCH --ntasks-per-node=32). Each MPI process is allocated one CPU core (#SBATCH --cpus-per-task=1).
   b. If you want a different configuration, which is highly recommended for your development purposes, below is an example of using 36 MPI ranks on 2 nodes for solving a 720 * 720 matrix:

```
#!/bin/bash
#SBATCH --job-name=MPI_JOB            # Job name
#SBATCH --mail-type=END,FAIL          # Mail events (NONE,
BEGIN, END, FAIL, ALL)
#SBATCH --mail-user=email@iitdh.ac.in # Where to send mail
#SBATCH --nodes=1                    # Run on 2 nodes
#SBATCH --ntasks-per-node=16          # 18 per node
#SBATCH --cpus-per-task=1             # 1 cpu per task
#SBATCH --time=00:10:00               # Time limit
hrs:min:sec
#SBATCH --output=test_%j.log   # Standard output and error
log
#SBATCH --partition=test   # Queuename
```

```
#SBATCH -n 36   # Must be a perfect square

pwd; hostname; date
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
srun ./summa_check 720
date
```

c. Below is an example of using 4 MPI ranks on 1 node to solve a 100 * 100 matrix:

```
#!/bin/bash
#SBATCH --job-name=MPI_JOB            # Job name
#SBATCH --mail-type=END,FAIL          # Mail events (NONE,
BEGIN, END, FAIL, ALL)
#SBATCH --mail-user=email@iitdh.ac.in # Where to send mail
#SBATCH --nodes=1                    # Run on 12 nodes
#SBATCH --ntasks-per-node=4          # 18 per node
#SBATCH --cpus-per-task=1             # 1 cpu per task
#SBATCH --time=00:10:00               # Time limit
hrs:min:sec
#SBATCH --output=test_%j.log   # Standard output and error
log
#SBATCH --partition=test   # Queuename
#SBATCH -n 4    # Must be a perfect square

pwd; hostname; date
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
srun ./summa_check 100
date
```

When you accept the assignment, it will provide you with a clone of a GitHub repository that contains the following starter files:`Makefile`, `interactivejob.sh`, `submit.sbatch`, `summa.cpp`.

Experiments

When you compile your code, make sure to use -O3 optimization. All of the execution measurements described below should be performed with this version of your executable.

1) For an input matrix of size n = 6720, vary the number of MPI processes over p = [2^2=4,3^2=9,4^2=16,5^2=25, 6^2=36,7^2=47, 8^2=64]. Use the matrix dimension as 6912. 6912 is chosen because the lcm(2,3,4,5,6,7,8) = 840 and 6720 = 8 * 840.
   a. During development, you can choose some other smaller multiple of 840.

2) Plot the speedup versus the number of processes. Compute the parallel efficiency for n=6720. Explain whether the scaling behavior is as expected.
3) **[PG students only]:** Repeat #1 and #2 for both blocking and non-blocking versions of summa.

Analyze the speedup and efficiency of the parallelized code:

Parallel efficiency

Prepare a graph of the efficiency of your parallelization comparing the real times measured. Plot a point for each of the executions. The X axis should show the number of MPI processes. The Y axis should show your measured parallel efficiency for the execution. Parallel efficiency is computed as S/(p * T(p)), where S represents the real time of a sequential execution of your program, p is the number of processors and T(p) is the real time of the execution on p processors. Discuss your efficiency findings and the quality of the parallelization.

Written report

Write a report that describes how your program works. This report should not include your program, though it may include one or more figures containing pseudo-code that sketches key elements of the parallelization strategy. Explain how your program exploits parallelism. For version 2 implementations with blocking, explain how the communication costs would be reduced if you chose to exploit parallelism in this way. Were there any opportunities for parallelism that you chose not to exploit? If so, why? Include all results for the sections described in the Experiments section above.

Submitting your assignment

Your assignment should be submitted in two parts.

- One or more source files containing your different versions of the program. Your submission must contain a Makefile with the following targets:
    - `team`: prints the team members' names on the terminal.
    - `summa`: builds related code and runs your experiments mentioned in Experiments.
    - `summa_check`: builds related code and validates your results.
    - `[PG students only]`
        - `summa_nb`: builds the version of your code that uses non-blocking collectives runs your experiments mentioned in Experiments. Assume that the "`summa`" target uses blocking collectives.
        - `summa_nb_check`: same as `summa_check` but for the non-blocking collectives.

- A report about your program in PDF format. Guidelines for your report: 3 pages won't have enough detail; more than 10 pages will say too much. Plan for a length in between. Make sure that you address all of the requirements specified in the "Experiments" section.

You will submit your assignment by committing your source files, Makefile, batch script, and report to your GitHub Classroom repository. If you have trouble using GitHub and come up against the deadline for submitting your assignment, email the instructor a tar file containing your source code and your report.

Click on the link shared above. This will create a repository in your GitHub account.

1. Clone the repository into your local development environment (on HPC master node) using the **git clone** command.
2. Add all assignment related files that you want to submit to your GitHub local repository using the **git add** command.
3. Save the changes using the **git commit** command
4. Upload the changes to GitHub using the **git push** command
5. Release your changes by first tagging your commit on the local environment using the **git tag -a cs410assignment3 -m "submitting assignment 3"**.
6. Next, push the tag to GitHub with the help of the following command: **git push --tags** \\
   If you want to make changes after you have submitted (repeat the above steps from 1 to 5 and apply modified commands shown below in place of step 4):
   git tag -a -f git tag -a cs410assignment3 -m "submitting assignment 3"
   git push -f --tags

Grading criteria

- 60% Experiments: split as 30% for correctness, 30% for conducting the experiments.
- 10% Program scalability and performance.
- 30% Report. Your grade on the report will consider the quality of the writing (grammar, sentence structure), whether all of the required elements are included, and the clarity of your explanations. Make sure that you have performed all of the experiments requested above and provided the information requested about them in your report. If you haven't, you will lose points!

Using the HPC machine

1. You must be connected to the VPN to access the DGX master node.
   a. Follow these instructions to connect to VPN.
2. `ssh <your id>@10.250.101.100`
3. Log onto 10.250.101.100 with your id and password.

Before submitting the first job, you need to setup password-less access using the following set of commands:

```
Command1:  ssh-keygen -t rsa (user action: press 3 enters)
Command2:  ssh cn01 ssh-keygen -t rsa  (press yes, enter login
passwd, press 3 enters)
Command3:  ssh-copy-id -i ~/.ssh/id_rsa.pub cn01 (enter the login
passwd)
Command4:  ssh cn01 `ssh-copy-id -i ~/.ssh/id_rsa.pub iitdhmaster`
(type yes, enter login passwd)
Check: ssh cn01 hostname  (If above process is correctly followed:
following hostname will be printed without asking for any passwd)
Cn01.iitdh.ac.in
```

Repeat the same step once more this time replacing `cn01` with `cn02`.

Using SLURM on HPC to submit jobs

While you can develop your program on one of the login nodes on HPC, you MUST run your experiments on one of the compute nodes using the SLURM job manager. You can obtain a private node to run your job on by requesting a node for an interactive session using

```
srun --pty --export=ALL --nodes=1 --ntasks-per-node=1
--cpus-per-task=32 --time=00:30:00 bash
```

A copy of this command is included in the file interactive among the provided files in your GitHub Classroom repository.

I strongly recommend developing and testing your code (with small matrix size) on the login node to avoid the surprise of having your editor be killed as your interactive session on a compute node expires.

Your sample repository includes a SLURM batch script submit.sbatch that you can launch with SLURM's sbatch utility. The sample batch script runs the program. You can edit the file to run your code just once or run it multiple times in a sequence of experiments.
Example slurm script (run: `sbatch submit.sbatch`)