CS410 Assignment 2
Matrix Multiplication using Pthreads and OpenMP

Date released: Feb/20/2022.
Due date: March/11/2022 (11:59 pm IST).

In this assignment, you will implement **matrix multiplication** using **shared-memory programming models of Pthreads and OpenMP.** 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`.

0) You will have to implement the basic SUMMA algorithm using OpenMP and Pthreads (This is version 0. ver0_omp.c, ver0_pt.c (or.cpp)).
1) You will have to implement the parallel version of the basic SUMMA algorithm using OpenMP and Pthreads (This is version 1. Call them ver1_omp.c and ver1_pt.c (or .cpp))
2) You will have to improve the version 1 implemented previously by choosing a block size of 'b' appropriately (value of 'b' chosen from 1 to $n/\sqrt{p}$). Here, you can assume that n is divisible by $\sqrt{p}$. (This is version 2. Call them ver2_omp.c and ver2_pt.c (or .cpp))
3) You will use the BLAS library routine `dgemm` to compute the update C=C+AB and compare the execution time with version 1 and version 2 implementations (This is version 3. Call them ver3_omp.c and ver3_pt.c (or .cpp)).

Teams with at least one PG student:

4) Find the parallel BLAS library routine for doing matrix multiplication, use it to compute C=C+AB, and compare its running time to your version 2 implementation (This is version 4.).

Your program should accept two arguments:

- First, the matrix size n (which initializes matrices with random floating point numbers).
- Second, a value specifying the number of threads to be used.

You will use the IITDH DGX 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/fkvT2TVx

When you accept the assignment, it will provide you with a clone of a GitHub repository that contains this problem statement only.

Experiments

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

1) For version 0, vary the size of your matrices from n=720, 1440, and 2160 and measure the runtime with one thread.
2) For version 1, for each matrix size of n mentioned previously in 1), vary the number of threads from p=4,9,16, 25, 36. Plot the speedup versus the number of threads. Compute the efficiency. Explain whether the scaling behavior is as expected.
3) For version 1 OpenMP implementations set n=250, 500, 1000, 2000, 4000 and p=2, 4, 8, 16, 32.
    a. use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup and efficiency.
    b. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup and efficiency.
4) For version 2 and version 3, set p=36 and vary b sizes to find the best b that yields the fastest execution time.

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 cores / threads. 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.

Note: Don't make the mistake of using the execution time of a one thread version of your parallel implementation as the time of a sequential execution. The execution time of the one-thread parallel implementation (T_1) is typically larger than that of the original sequential (T_s) implementation. When you compute parallel efficiency, always use the performance of a sequential code as a baseline (T_s); otherwise, you will overestimate the value of your 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.
    - `exp1`: builds related code and runs your experiments mentioned in Experiments 1).
    - `exp2`: builds related code and runs your experiments mentioned in Experiments 2).
    - `exp3`: builds related code and runs your experiments mentioned in Experiments 3).
    - `exp4`: builds related code and runs your experiments mentioned in Experiments 4).

- 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, 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 DGX 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 cs410assignment2 -m “submitting assignment 1”**.
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 cs410assignment2 -m ”submitting assignment 2”
   git push -f --tags

Grading criteria

- 60% Experiments: split as 20% for correctness, 40% 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 DGX 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>@10.250.101.55`
3. Log onto 10.250.101.55 with your id and password.

Using SLURM on DGX to submit jobs

While you can develop your program on one of the login nodes on DGX, 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 vector 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 sort 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`)

```
#!/bin/bash
#SBATCH --job-name=CILK_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 a single node
#SBATCH --ntasks-per-node=1           # Single task
#SBATCH --cpus-per-task=32            # 32 CPUs per task
#SBATCH --time=00:05:00               # Time limit hrs:min:sec
#SBATCH --output=serial_test_%j.log   # Standard output and error
log
pwd; hostname; date

set -ex
make all
make runSortCilk CILK_NWORKERS=32 SIZE=1000000000 CUTOFF=10
date
```

The default queue for the DGX machine is "v100". If we don't specify any queue name, v100 will be chosen. If v100 becomes overloaded, there is another queue "a100", which you can use. To

explicitly specify the queue name, add the following extra line to batch script you submit to the sbatch command:

```
#SBATCH --partition=a100
```