TDT4200-PS4 – OpenMP and pthreads

Zawadi Berg Svela and Jacob Odgård Tørring

Deadline: Wed October 20, 2021 at 22:00 on BlackBoard
Evaluation: Pass/Fail

- All problem sets are to be handed in INDIVIDUALLY. You may discuss ideas with max. 1-2 collaborators and note them in the comments on the top of your code file, but code sharing is strongly discouraged. Preferably seek help from TAs rather than get unclear/confusing advice from co-students!
- Code should not use other external dependencies aside from the ones we specified.
- The minimum requirement to pass this assignment is to have Part 1 working and and Part 2a implemented.

1 Introduction

In this assignment, you will parallelize a Double-precision GEneral Matrix Multiplication(DGEMM) operation using OpenMP and Pthreads, and compare the performance with a manual serial and an OpenBLAS's DGEMM implementation.

For those of you who have not done matrix-matrix multiplication in a while, it is a operation between a $m \times k$ matrix A and a $k \times n$ matrix B that produces a $m \times n$ matrix C, where each cell $C[i, j]$ is the dot product/inner product between the i'th row of A and the j'th column of B. In all of the provided code, this is the meaning of the variables A, B, C, m, n and k. See the Figure 1 for a graphical representation of matrix-matrix multiplication. Matrix vector multiplication is covered in the Pacheco text.

You can also check out "Multiplying matrices" at: https://www.khanacademy.org/

About OpenMP

OpenMP is a lightweight API for parallel programming. While its scope and functionality has grown a lot over the years, including tasking (see lecture notes), one of its most important functionalities is to quickly and easily parallelize code sections, like loops.

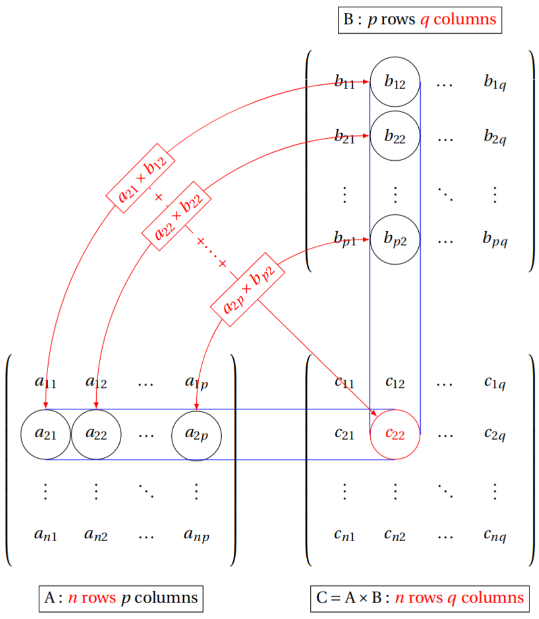

Figure 1: Graphical representation of matrix-matrix multiplication. Keep in mind that this figure uses different letters to indicate the dimensions than in the provided code.

Most of OpenMP's functionality is accessed via compiler directives. They look like this: `#pragma omp <something>`. They generally "attach themselves" to the following scope, so if you had a code section like

```
#pragma omp parallel num_threads(4)
{ //Remember, curly braces (generally) delineate scope
    printf("Thread \%d says hello!", omp_get_thread_num());
}
```

...then the inner scope in the section would execute in parallel across 4 threads.

In order to parallelize a for-loop, you need `#pragma omp parallel for`. You need to ensure that the loop satisfies the requirements of OpenMP, most importantly that all of the loop's iterations can be evaluated when the loop begins. That means to early exit the loop or manually incrementing the loop iterator outside the loop header. The iterations also need to be independent (the lines of code can be interleaved across threads in any arbitrary way) unless locking tools such as "critical sections" are used.

Prerequisites

Linux and Windows/WSL: pthread and openmp should work automatically if you have gcc installed. However, if OpenMP does not work for some reason (e.g., your using clang), it might help to explicitly install the libomp-dev package (the OpenMP package for LLVM).

Mac: The code should work as-is. If it does not, you might need to install Homebrew[1] and use it to install gcc (which has support for OpenMP): `brew install gcc`.

To ensure that you are using the gcc you installed and not clang (the built-in C compiler), you might need to update your PATH environment variable: `PATH="/usr/local/Cellar:"$PATH`.

Provided Files

- Makefile (run **make** to compile the project)
- main.c (the file to edit for this problem set)

Problem Description

This problem set consists of two parts. First you will implement a matrix-matrix multiplication parallelized using OpenMP, and afterwards using Pthreads.

[1]https://brew.sh/

Requirements

The OpenMP and Pthread-sections should perform the calculation correctly. They should only work on their designated result-matrices, i.e. `C_openmp` and `C_pthreads`, as this is used to determine correctness. Both parallelizations should use the `num_threads` global variable to determine the number of threads, and should produce the correct result with an arbitrary number of threads. This variable can be set using the "-t" argument when running the program, and is set to 4 by default.

Part 1 - OpenMP

This part is marked by TODO: OpenMP in the handout code.

- Using the OpenMP API, set the number of threads it should use in parallell section(s).
- Parallellize the outer loop of the matrix-matrix multiplication using OpenMP compiler directives.
- **(Optional)** Try to parallellize the inner-most loop. Watch out for race-conditions!

Part 2 - Pthreads

This part is marked by TODO: Pthreads in the handout code. The section within the `main()` function contains only timing functionality, and this is where you will spawn the threads. The actual computation will be done within the function made as part of this task.

2a - Thread function

These sub-tasks should all be done in the TODO section *outside* of `main()`.

1. Create a new function to be called by `pthread_create()`. It should have the return-type `void *` and take a single argument `int *thread_num_p`.
2. Copy the serial matrix-matrix multiplication code into this function. Adjust it so that given a `thread_num` between 0 and `num_threads - 1`, it computes its part of the output matrix `C_pthreads`. The logic you need to implement will be similar to working with MPI.

2b - Thread spawning

These sub-tasks should all be done in the TODO section *inside* of `main()`.

1. Allocate two arrays. One for the thread-ids (assigned by `pthread_create()` and one for the `thread_num` variables (assigned by you).

2. Create a for-loop set the `thread_num` variable and call `pthread_create()` for each thread.
3. Create a for-loop to join the threads spawned. For each thread, call `pthread_join()` on its corresponding thread id.

Deliverables

- The main.c file