CPTS 223 Advanced Data Structure C/C++ Fall 2024

MA4: Parallel Programming and Heaps

1 Learning Objectives

At the conclusion of this programming assignment, participants should be able to:

- Implement solutions to parallel algorithms (OpenMP);
- Implement solutions to percolate down and percolate up algorithms for binary heaps.

2 Prerequisite

Before starting this programming assignment, participants should be able to:

- Analyze a basic set of requirements and apply parallel design principles for a problem
- Describe and analyze binary heaps
- Edit, build, and run programs through a cross-platform environment (CMake)
- Read §6.3 Binary Heaps of Textbook and slides on Canvas. After reading, you will understand:
  - Structure property of binary heaps;
  - Heap-order property of binary heaps;
  - Basic heap operations, including percolation-up and percolation-down.

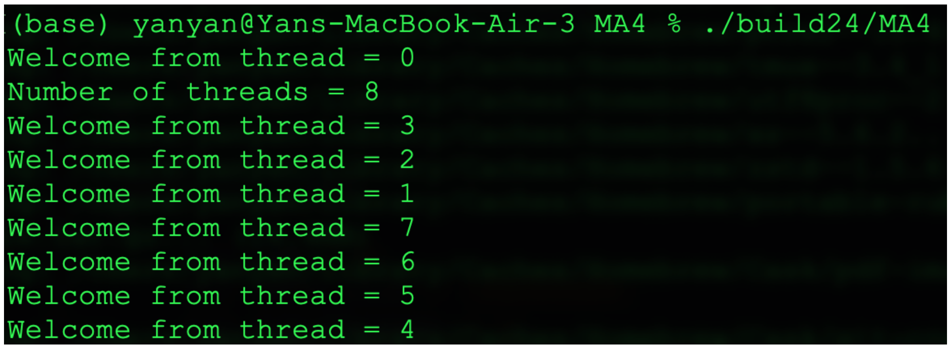
3 Overview and Requirements

3.1 Overview

For this assignment, you will start with the starter code template provided on Canvas. You are required to implement "TODO" functions in "OpenMP.h" and "Heap.h". After that, run test cases in "main.cpp" (should not need to change anything already in "main.cpp").

OpenMP is a widely-used standard library for multithreaded parallel programming in C/C++. You will be given a computation task to parallelize using OpenMP, breaking it down into multiple threads to improve execution efficiency. This part will cover fundamental OpenMP directives, such as `#pragma omp parallel`. Furthermore, you will learn your first mutual exclusion in OpenMP that deals with the possible *race condition* (i.e., multiple threads attempt to update the same memory location at the same time). You will observe and analyze the impact of different thread counts on execution time.

Heaps use the percolate operation as a critical procedure in inserting and removing elements to ensure that the heap properties (either min-heap or max-heap) is maintained. We have learned that for binary heaps, percolation-up and percolation-down restores the heap-order property. You will implement "percolateUp" and "percolateDown" functions, so that after inserting a new element or removing the smallest element, the binary heap still satisfies its heap-order property.



```
(base) yanyan@Yans-MacBook-Air-3 MA4 % ./build24/MA4
Welcome from thread = 0
Number of threads = 8
Welcome from thread = 3
Welcome from thread = 2
Welcome from thread = 1
Welcome from thread = 7
Welcome from thread = 6
Welcome from thread = 5
Welcome from thread = 4
```

Figure 1: Example running result of `hello_word()` in the starter code for testing OpenMP multi-threads. This figure shows that there are 8 threads available on this computer.

3.2 Requirements

3.2.1 OpenMP (50 pts)

You will implement the two functions that use the OpenMP library with OpenMP directives: `calSum()` and `calMax()`. Before implementing these two functions, we can use the provided `hello_world()` function to test if the OpenMP library is properly installed and configured. See Appendix A for "how to install/configure OpenMP" if you have not done so. Figure 1 shows the example running result when OpenMP is properly functioning. The specific requirements for the two TODO functions are:

- (25 pts) `calMax()` requires finding the maximal value from an input array using OpenMP with at least two threads.
- (25 pts) `calSum()` requires calculating the sum of an input array using OpenMP with at least two threads.
- For both functions, you may declare a local variable to store the result for each thread and aggregate all local variables to the final result. For example, on the slides page "Sum using OpenMP" in the chapter of parallel programming, we see that a "local_sum" variable is declared within the parallel code, and each thread will keep their local result in this variable temporarily until aggregating to a global variable "sum". You may follow this principle.
- Moreover, you are also required to use a proper synchronization functionality of OpenMP (e.g., critical, atomic, lock, etc.) to set mutual exclusion when aggregating the local result safely.

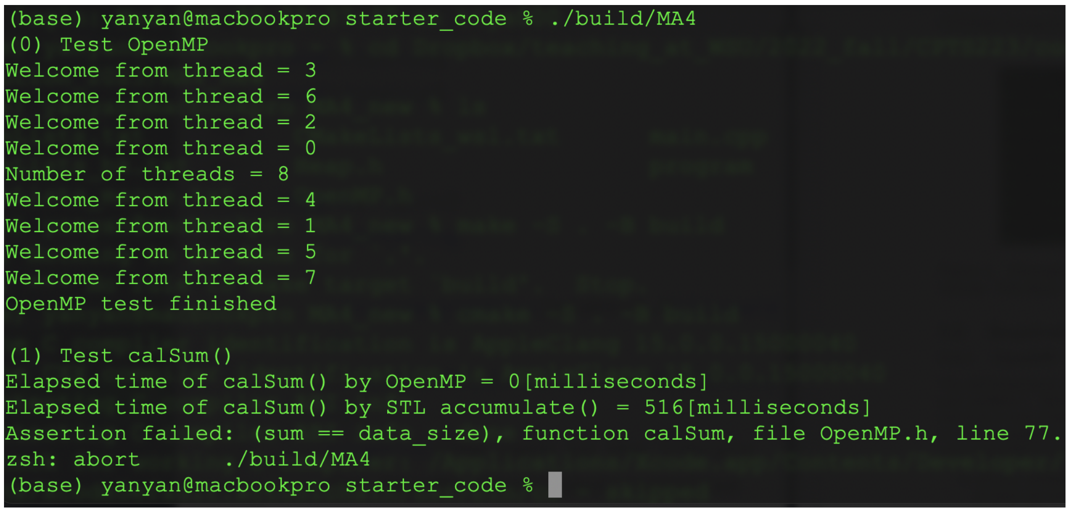
Expected running results. For both functions, the starter code provides test functions which evaluate the time spent by your OpenMP implementation to process the input data. On the other hand, it will compare this running time with that spent by C++ STL function `std::max_element()`[1]. We should expect the running time by OpenMP implementation (parallel) is less than that by C++ STL (serial).

3.2.2 Binary Heaps (50 pts)

In a min-heap (the smallest item at the root), the underlying implementation is an array (`std::vector`) and the first item in the array is a placeholder as we mentioned in the class. The starter code provides partial implementation for binary heaps in "Heap.h", which is slightly different from that in Textbook. By referring the textbook, you may need to adjust it a little bit to fit into the template.

- (25 pts) `percolateDown()`: refer to Figure 6.12 in Textbook. It is called by `pop()` when deleting data.
- (25 pts) `percolateUp()`: refer to Figure 6.8 in Textbook. This function adds an element and perform percolation up to restore the heap-order property.

[1]You may refer to this link or this link for documentation of `std::max_element()`.



```
(base) yanyan@macbookpro starter_code % ./build/MA4
(0) Test OpenMP
Welcome from thread = 3
Welcome from thread = 6
Welcome from thread = 2
Welcome from thread = 0
Number of threads = 8
Welcome from thread = 4
Welcome from thread = 1
Welcome from thread = 5
Welcome from thread = 7
OpenMP test finished

(1) Test calSum()
Elapsed time of calSum() by OpenMP = 0[milliseconds]
Elapsed time of calSum() by STL accumulate() = 516[milliseconds]
Assertion failed: (sum == data_size), function calSum, file OpenMP.h, line 77.
zsh: abort      ./build/MA4
(base) yanyan@macbookpro starter_code %
```

Figure 2: Running result from the starter code.

3.2.3 Test Cases

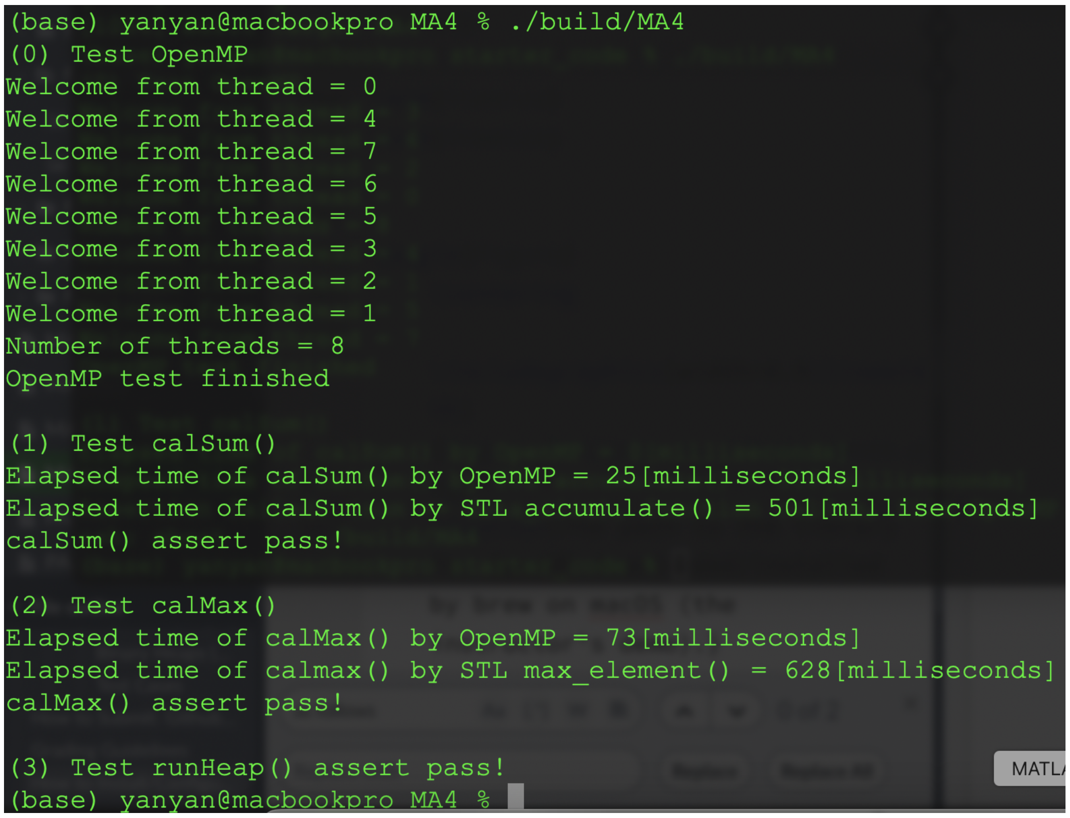
The "main.cpp" contains a number of test cases.

- The `calSum()` will print the sums computed by OpenMP and STL `accumulate()`, as well as the running time, respectively. If your implementation is correct, the two values should match and assert will pass. Additionally, we can examine the speedup rate of OpenMP over the serial program. Now it fails.
- The `calMax()` will print the max value found by OpenMP and STL `max_element()`, as well as the running time. If your implementation is correct, the two values should match and assert will pass. Additionally, we can examine the speedup rate of OpenMP over the serial program. Now it fails.
- The `runHeap()` function will push a number of random values into the heap and pop the values one by one. It compares the popped values with the values sorted by STL. If your implementation is correct, the assertion function used in the loop in "main.cpp" should pass. Currently, it fails.

See Figure 2 and Figure 3 for the running result from the starter code and expected code, respectively.

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "MA3"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "MA3" and place all MA3 files in the "MA3" directory. All files for MA3 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).
2. You should submit at least the following files:
   - the header files: "Heap.h", "OpenMP.h".
   - the main C++ source file: "main.cpp".
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.
3. You can refer the example GitHub template project for how to use CMake at `https://github.com/DataOceanLab/CPTS-223-Examples`.



```
(base) yanyan@macbookpro MA4 % ./build/MA4
(0) Test OpenMP
Welcome from thread = 0
Welcome from thread = 4
Welcome from thread = 7
Welcome from thread = 6
Welcome from thread = 5
Welcome from thread = 3
Welcome from thread = 2
Welcome from thread = 1
Number of threads = 8
OpenMP test finished

(1) Test calSum()
Elapsed time of calSum() by OpenMP = 25[milliseconds]
Elapsed time of calSum() by STL accumulate() = 501[milliseconds]
calSum() assert pass!

(2) Test calMax()
Elapsed time of calMax() by OpenMP = 73[milliseconds]
Elapsed time of calmax() by STL max_element() = 628[milliseconds]
calMax() assert pass!

(3) Test runHeap() assert pass!
(base) yanyan@macbookpro MA4 %
```

Figure 3: Running result from the expected code.

4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.

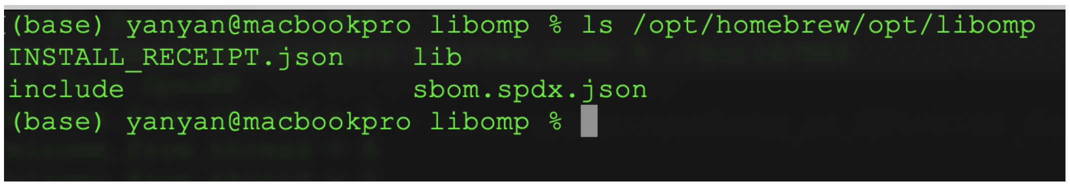
5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.



```
(base) yanyan@macbookpro libomp % ls /opt/homebrew/opt/libomp
INSTALL_RECEIPT.json    lib
include                 sbom.spdx.json
(base) yanyan@macbookpro libomp %
```

Figure 4: The path to the library for OpenMP downloaded/installed by brew on macOS (the instructor's demo).

A How to Install/Configure OpenMP

- Linux/WSL system: If you already installed gcc/g++ compiler, then the compiler typically includes the OpenMP library and supports OpenMP by default. When building the project with CMake, we need a slightly different C++ Flags in "CMakeLists.txt" (this has been included in "CMakeLists_linux.txt" of the starter code) as follows:

  ```
  set(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -Wall -fopenmp")
  ```

  The new C++ Flag `-fopenmp` above will enable the compiler to use OpenMP library.

- macOS: unfortunately, OpenMP library is not included in Xcode by default (even if you installed gcc/g++ already), so you have to install OpenMP manually. To this end, follow the two steps below to install and configure OpenMP from Terminal:

  - use "brew"[2] for installing the OpenMP library.

    ```
    brew install libomp
    ```

    See this 1-min YouTube video for a demo of this Terminal operation.

  - use a modified C++ Flags in "CMakeLists.txt" (this has been included in "CMakeLists.txt" of the starter code) as follows

    ```
    set(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -Wall -Xpreprocessor -fopenmp
    -lomp -I/opt/homebrew/opt/libomp/include -L/opt/homebrew/opt/libomp/lib")
    ```

    Please note that the above paths, i.e., "/opt/homebrew/opt/libomp/include" as well as "/opt/homebrew/opt/libomp/lib", are derived from the instructor's computer. Figure 4 showcases this path for OpenMP library.

[2]See how to install "Homebrew" if you do not have one on your macOS at https://brew.sh/, under "Install Homebrew".