San José State University
Department of Computer Engineering

CMPE 180A
Data Structures and Algorithms in C++
Fall 2020

Instructor: Ron Mak

Assignment #14

Assigned: Tuesday, November 24
Due: Friday, December 4 at 11:59 PM
Canvas: Assignment 14: Multithreading
Points: 190

Multithreading
The purpose of this assignment is to give you some experience writing multithreaded programs and some basic practice with the 2011 C++ multithreading library.

Part 1: Multithreaded Borwein π algorithm
Recall the Borwein π algorithm from Assignment #9 that used the MPIR multiple precision library. For Part 1 of this assignment, you will write a multithreaded version of the algorithm to compute π to one million digits.

Set $a_0 = 6 - 4\sqrt{2}$ and $y_0 = \sqrt{2} - 1$. Then iterate

$$y_i = \frac{1 - \sqrt[4]{1 - y_{i-1}^4}}{1 + \sqrt[4]{1 - y_{i-1}^4}}$$

$$a_i = a_{i-1}(1 + y_i)^4 - 2^{2i+1} y_i (1 + y_i + y_i^2)$$

and the a_i will converge quartically towards $1/\pi$.

Examine the algorithm carefully:

- What parts can be computed simultaneously in different threads?
- Can you distribute the work among the threads for optimal simultaneity?
- What are the shared data and critical regions?
- How must the threads be synchronized?
- Can computing at the end of one iteration overlap with computing at the start of the next iteration?

Use MPIR and the 2011 C++ multithreading library to compute and print one million digits of π with the Borwein algorithm. Record the total time to compute the digits, but do not include the printing time. Compare your multithreading compute time with a single-threaded implementation of the algorithm, which you can take from your solution to Assignment #9. Compare your π with Dr. Evil's at http://3.14159265358979323846264338327950288419716939937510582097494459 2.com/index314159.html

Note 1: Ten iterations are required to compute a million digits.

Note 2: If you were unable to successfully install MPIR, you can do this part of the assignment by using the built-in **`long double`** scalar datatype to implement the Borwein algorithm and compute 19 digits of π. Your timings will not be as dramatic. In fact, your multithreaded implementation may run slower due to threading overhead.

```
#include <iostream>
#include <iomanip>
#include <math.h>

using namespace std;

int main()
{
    cout.precision(25);

    double one = 1.0, three = 3.0;
    long double longone = 1.0L, longthree = 3.0L;

    cout << "     double 1/3 = " << one/three << endl;
    cout << "long double 1/3 = " << longone/longthree << endl;

    cout << endl;

    cout << "     double pi = " << acos(-1.0) << endl;
    cout << "long double pi = " << acos(-1.0L) << endl;
    cout << "     actual pi = 3.14159265358979323846264 3..." << endl;

    return 0;
}


     double 1/3 = 0.3333333333333333148296163
long double 1/3 = 0.3333333333333333333423684

     double pi = 3.1415926535897931159979634
long double pi = 3.1415926535897932385128089
     actual pi = 3.1415926535897932384626433 ...
```

Part 2: Multithreaded quicksort

Recall how the quicksort algorithm works by recursively sorting subarrays. For each subarray (initially the entire array), choose a pivot element and partition the subarray into two smaller subarrays on either side of the pivot. The pivot is now in its proper place in the final sorted array. Recursively quicksort each of the smaller subarrays.

Quicksort is ideal for multithreading because each subarray can be sorted by a separate thread, and each thread works on a different part of the entire array. Instead of making a recursive call to quicksort a subarray, a thread takes over to do that sort.

Here is one approach:

- Use a "work queue" to keep track of subarrays to be sorted. Each element of the queue is a "work item" pair <*left index*, *right index*> which represents the left index and right index of a subarray to be sorted. Initialize the work queue with a work item that represents the entire array.
- Create a "thread pool" of N sorting threads. For this assignment, you will try pools of different sizes to see how using more threads affects performance.
- Each sorting thread runs the same function. In a loop, the function first attempts to pop a work item off the work queue. If the queue is empty, the function must wait for the queue to become nonempty. Once it gets a work item and thereby acquires a subarray, it proceeds to choose a pivot and partition the subarray into two smaller subarrays on either side of the pivot. The function next pushes two work items onto the work queue that represent the two smaller subarrays. It then loops back to attempt to pop another work item (which in the meantime other sorting threads are entering) off the work queue.

Important considerations:

- What are the shared data and critical regions?
- How must you synchronize the sorting threads?
- How do you ensure that all the sorting threads stay working until the entire array is sorted? A thread should not terminate when it encounters an empty work queue, because the queue may be empty only temporarily.
- How do you ensure that the threads are actually working simultaneously? Poor multithreading: (1) one thread takes over and does all the work, or (2) the threads take turns working so they do all the work sequentially instead of simultaneously.
- Threads must yield to one another. Look at the producer-consumer example. Each sorting thread is both a producer and a consumer.
- Multithreading incurs overhead. How fine-grained should the synchronization be? If you overdo it, the overhead can overwhelm any advantages to multithreading.
- How do you prevent deadlocks? If your program freezes, that's the cause?
- How do you know when the sorting is complete? (As noted above, an empty work queue does not necessarily mean that the sorting is done.)

Generate an unsorted array or vector of one million integer values. If you use a vector, a simple means to generate unsorted values:

```
#include <algorithm>

vector<int> data;

data.reserve(DATA_SIZE);
for (int i = 0; i < DATA_SIZE; i++) data.push_back(i);
random_shuffle(data.begin(), data.end());
```

Use your multithreaded quicksort to sort the array. Use thread pools of sizes N = 1, 2, 3, ... 8. For each thread pool size, compute the total amount of time to sort the data.

Before and after sorting the data, verify that your data is or isn't sorted:

```
bool is_sorted(vector<int>& data)
{
    for (int i = 1; i < data.size(); i++)
    {
        if (data[i] < data[i-1]) return false;
    }

    return true;
}
```

Not for CodeCheck
CodeCheck will not be used for this assignment.

Submission into Canvas
Submit into Canvas a copy of your C++ program(s) and a text file of its output:
Assignment 14: Multithreading

Rubric
Your program will be graded according to these criteria:

Criteria	Max points
Part 1: Borwein π algorithm	**90**
• Good multithreading design with good work distribution among the threads.	• 20
• Good use of the 2011 C++ multithreading library components and API.	• 20
• Timing comparison with a single-threaded implementation	• 20
• Correct value of π was computed. (Last 10 digits should be 57794 58151.)	• 30
Part 2: Quicksort	**100**
• Good multithreading design with good work distribution among the threads.	• 20
• Good use of the 2011 C++ multithreading library components and API.	• 20
• Timings for each thread pool size.	• 20
• Data correctly sorted for each thread pool size.	• 40

Academic integrity

You may study together and discuss the assignments, but what you turn in must be your individual work. Assignment submissions will be checked for plagiarism using Moss (http://theory.stanford.edu/~aiken/moss/). **Copying another student's program or sharing your program is a violation of academic integrity.** Moss is not fooled by renaming variables, reformatting source code, or re-ordering functions.

Violators of academic integrity will suffer severe sanctions, including academic probation. Students who are on academic probation are not eligible for work as instructional assistants in the university or for internships at local companies.