**Meng Yang – CSS503 A – Program2**

**Ⅰ. DOCUMENTATION**

**Technique used to extend the Sleeping-Barbers Program**

The technique used in the assignment is monitor, a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met.

**Synchronization Strategies:**

There are generally three techniques we can use to synchronize a multi-task program: Mutex Lock, Semaphore and Monitor.

**Mutex Lock:** A hardware level strategy where a process must acquire a lock before entering a critical section, and releases the lock when it exits the critical section. The advantage of Mutex Lock is it doesn’t require context switch, which may take considerable time. The disadvantage is it involves busy waiting (spinlock), which wastes CPU cycles.

**Semaphore:** A Operating System level strategy where an integer variable S is accessed only through two standard atomic operations: wait() and signal(). When one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value. The advantages of Semaphore include: no busy waiting involved, no signals are lost. Its disadvantages: it’s complex to implement, and system calls are required.

**Monitor:** A language level synchronization constructin which a condition variable provides a method by which a monitor function can block its execution until it is signaled to continue. The advatages of Monitor include: it allows only one process to be active at a time, and it works across multiple tasks. Its diadvantages are: not every programming language support it, and there are chances where signals could be lost.

In my implementation, each barber and customer is a thread that has several condition varables, i.e..

**Ⅱ. SOURCE CODE**

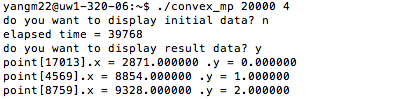
Please find the source code in file Shop.h, Shop.cpp and driver.cpp uploaded.

**Ⅲ. EXECUTION OUTPUT**

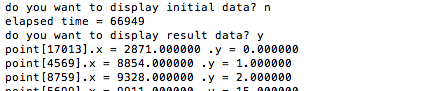
The performance improved. Below I have attached 3 cases for your perusal.

Case1: 20,000 4

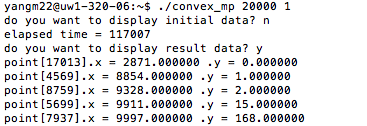
Performance improvement = 117007 / 39768 = 2.94



Case2: 20,000 2

Performance improvement = 117007 / 66949 = 1.75 

Case3: 20,000 1



**Ⅳ. DISCUSSIONS**

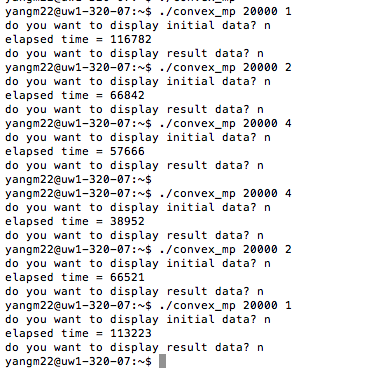
Data Parallelism is done in divide\_and\_conquer() of convex\_mp.cpp. This method recursively divides the points into subsets until there are two or more points in each subset. If the number of processes is bigger than 1, in my case, numOfProc is 2 or 4, then a pipe is created between parent and child. The child is responsible for working on the right queue, while the parent working on the left.

Something worth mentioning is in the parent code, we need to declare an empty queue s3 to receive the data received from child. I first tried making s1 do this job, however, the outcome contains far more points than expected. This is because every time the recv() is called, the existing points in the queue remain there. It took me a long time figuring this out.

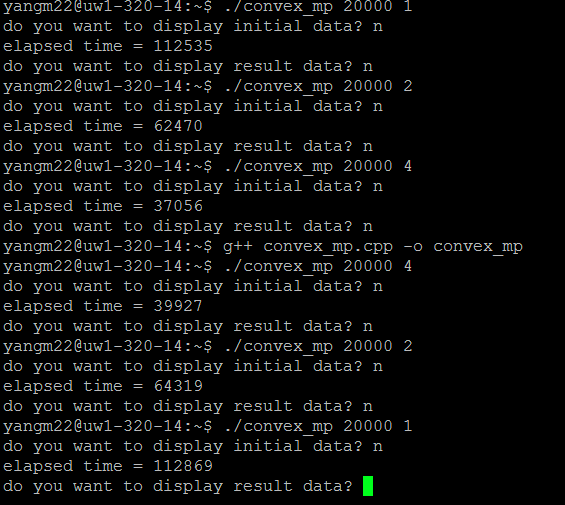
Also, in parent block, the recv(pipeFD[WRITE], s2) needs to be called after the recursive call, or the performance wouldn’t improve. This is because reading from the pipe is a blocking process. If there’s nothing to read from the pipe, the parent would wait until there’s data available. By making the recursive call before reading, we’re making sure that the parent doesn’t have to wait.

**Limitations:**

1. When testing in the order of 4 cores, 2 cores and then 1 core (4-2-1), as illustrated above, the performance is as expected. However, when run in the order of 1-2-4, the performance of 4 cores is only slightly better than that of 2 cores. Nevertheless, when I tested on another machine another time, this issue didn’t show up. This is illustrated in the following two snapshots.



To my understanding, there is memory leak in the other parts of this program that lead to this problem. According to my test, each point creates a memory leak of 24 bytes. Besides, the performance of the CPU isn’t consistent. Since the way we calculate the elapsed time is making time stamps between the two moments, there’s a great chance that during this period of time, the CPU is doing other tasks, which makes the test result instable.



1. The efficiency when running with 1 core isn’t exactly twice as that of with 2 cores or 4 times as that of with 4 cores. It’s around 1.75 and 2.95 because a reasonable amount of execution time is spent on making copies of the variables and allocating data to when forking new processes. This is considered as overhead of execution. The process of splitting up data set, processing the parts and recombining them creates some overhead, too.
2. There is always an oscillation in the performance performed on parallelization because it is based on the performance of the machines we run the program on.

**Possible Improvements:**

1. Instead of creating processes, we can create multiple threads to increase the performance. Since threads are within processes, there should be less overhead created.
2. Apply better methods of elapsed time calculation.
3. Fix the memory leak.