Homework #2

Andrew Kirfman

CSCE-435-100

Due Date: 6/07/17

Part #1: Minimum of a List

Question #1: Complete the function find_minimum so that the program computes the minimum of the list correctly. You will get full points if you add code only to the find_minimum routine.

In this problem, each thread computes the minimum of a subset of a larger list. One of these local minimums will be the global minimum. Threads need to aggregate their results together to produce a global minimum. The key of this portion of the assignment is to ensure that the threads are able to work together to calculate the global minimum in a safe and correct way.

The main question is, how should the program calculate the global minimum using the local minimums that all worker threads calculate? The easiest solution would be to record each local minimum somewhere in memory as the threads finish and then have a parent thread compute the global minimum by finding the minimum of the list of local minimums. The problem with this scheme is that it introduces a serial fraction at the end of the program (specifically, iterating through the list of intermediate results) which reduces the parallel performance/speedup of the algorithm. This serial fraction would only get worse as the number of threads increases.

For example, for two threads, the parent only has to find the minimum of two values (which can be done on most CPUs in just one instruction). However, if you're trying to find the minimum of a large list with 10,000 threads, the parent thread must iterate through a list of 10,000 intermediate local minimums in order to find the global minimum. In essence, the size of the serial fraction is linear with respect to p.

Depending on system scheduling, CPU load, and a multitude of other factors, some threads on the system will finish before others. Some threads may finish faster than others. In the scheme mentioned previously, wherein one thread computes the global minimum, the result cannot be calculated until all threads finish calculating their local minimums. This means that any thread that finishes early will sit idle until the rest have completed. The runtime will be dependent on the slowest thread + the serial fraction at the end to compute the global minimum.

Instead of doing this, what if the threads could compute the result as they were working? Simply maintain a global variable to all threads that contains the intermediate minimum value. As a thread finishes, have it check its minimum against the global minimum. If the thread's local minimum is less than the global minimum, then update the global minimum to equal the thread's local minimum.

There are two challenges with this scheme: 1) how do you ensure mutual exclusion between all of the threads when accessing the global minimum variable, and 2) how do you initialize the global minimum if you aren't able to do so from inside of another function (as was the case with this assignment).

The first challenge is an example of the mutual exclusion problem. The portion of code that

updates the global minimum is a critical section. This means that only one thread should be able to access this section of code at a time. To ensure mutual exclusion, I protected this segment of code with a mutex lock. Threads entering the critical section will either gain access to the lock (if it was previously unlocked) or will wait until the lock becomes available. When finished with the critical section, any thread that has the lock will unlock it.

For the second challenge, the issue is that I cannot compare a local minimum value to a global minimum if the global minimum is not initialized. The first thread to run the critical section will attempt to compare its local minimum against an indeterminate value. Doing so will result in undefined behavior.

In order to prevent this, I keep track of a variable that represents the number of threads which have entered the critical section, named count (this was provided to us in the source code). The first thread to finish calculating its local minimum will find that count is equal to zero. It will then update the global minimum to be equal to its own local minimum (without doing any comparisons) and then increment the count. Any thread afterwards will find count != 0 and as a result will compare their local minimum with the global minimum (since, at this point in the program's execution, the global minimum has been initialized to a definite value).

Using this method, once all of the threads exit, the value for global minimum will be available immediately since it was already computed during thread execution.

The code that I created and filled into the find_minimum function is as follows: (this can also be seen in the file list_minimum.c which is included in my submission!)

```
// Lock the mutex to access to global minimum variable.
pthread_mutex_lock(&lock_minimum);

// If we are the first thread to run, then we need to initialize the global minimum variable.
if (count == 0)
{
    minimum = my_minimum;
}
// Otherwise, perform a comparison and update the global minimum if necessary.
else
{
    if (my_minimum < minimum)
    {
        minimum = my_minimum;
    }
}

// Increment count since a new thread has accessed the critical section
```

```
count = count + 1;

// Each thread should unlock the mutex when it is done!
pthread_mutex_unlock(&lock_minimum);
```

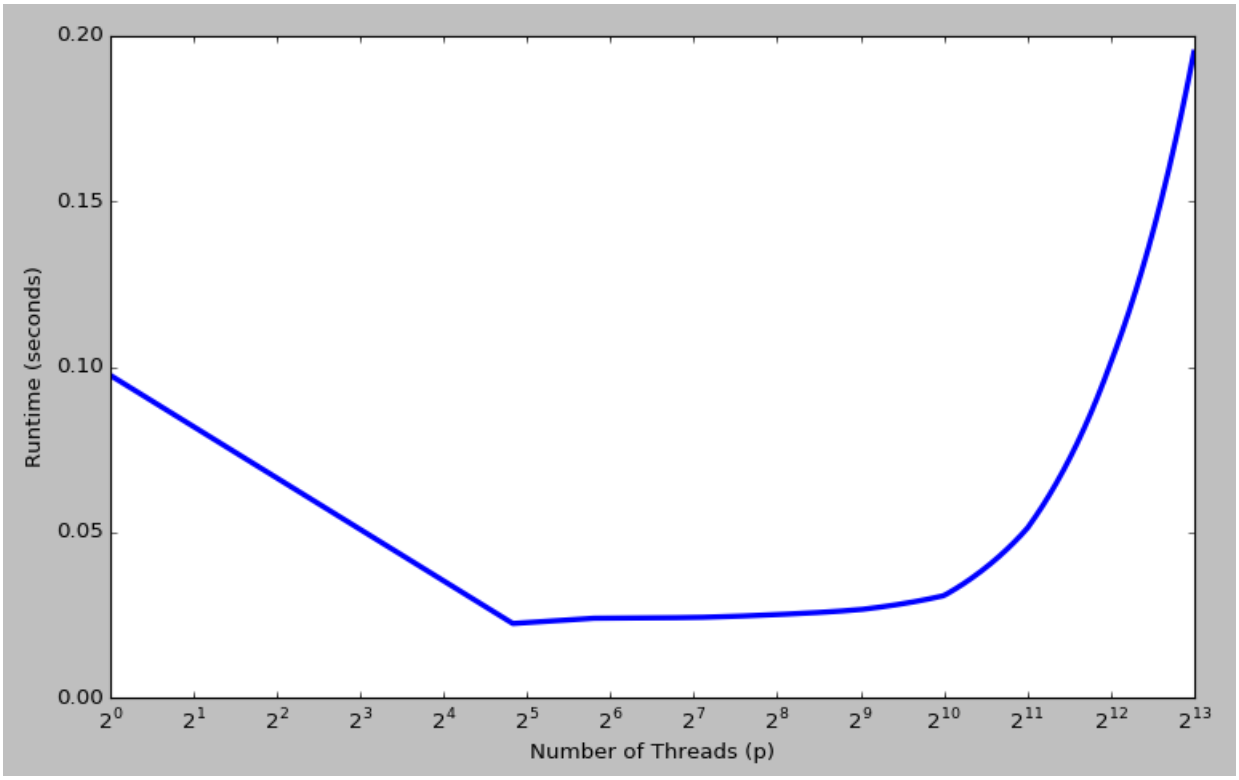
Note: As specified in the problem description, I did not add any new logic to any other function than the find_minimum function (so that I would be elligible to receive full credit). However, I did change the random number seed in the main function from "srand48(0)" to "srand48(time(0))". The original version produced the same random number sequence each time it ran. Adding the time function in fixed this issue, generating a different number each time the program was run.

Question #2: Execute the code for $n = 2 \times 10^8$ with p chosen to be 2^k, for $k = 0, \cdots, 13$. Plot execution time versus p to demonstrate how time varies with the number of threads. Use a logarithmic scale for the x-axis. Plot speedup versus p to demonstrate the change in speedup with p.

Table #1: Table of runtimes for varying p, $n = 2 \times 10^8$

Number of Threads (p)	Runtime (seconds)
1	0.0974
2	0.0502
4	0.0298
8	0.0239
16	0.0173
32	0.0240
64	0.0241
128	0.0243
256	0.0252
512	0.0267
1024	0.0309
2048	0.0512
4096	0.1011
8192	0.1949

Figure #1: p vs. Runtime for list_minimum.c

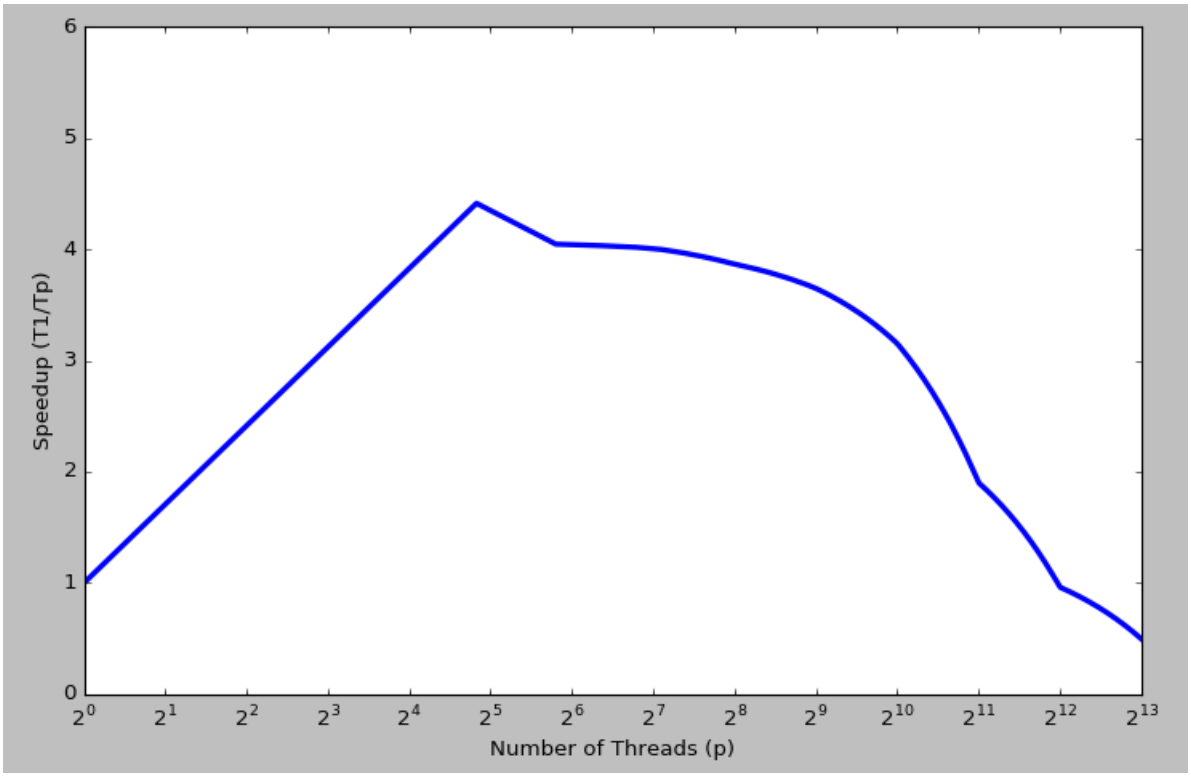


Observations: The runtime to calculate the minimum in a list decreases proportional to the number of threads up until $p = 16$. Afterwards, the runtime begins to increase, eventually growing larger than the runtime to make the calculation using only 1 thread.

Table #2: Speedup Data for list_minimum.c

Number of Threads (p)	Speedup ($S = T_1/T_p$)
1	$0.0974/0.0974 = 1.000$
2	$0.0974/0.0502 = 1.940$
4	$0.0974/0.0298 = 3.268$
8	$0.0974/0.0239 = 4.075$
16	$0.0974/0.0173 = 5.630$
32	$0.0974/0.0240 = 4.058$
64	$0.0974/0.0241 = 4.041$
128	$0.0974/0.0243 = 4.008$
256	$0.0974/0.0252 = 3.865$
512	$0.0974/0.0267 = 3.648$
1024	$0.0974/0.0309 = 3.152$
2048	$0.0974/0.0512 = 1.902$
4096	$0.0974/0.1011 = 0.963$
8192	$0.0974/0.1949 = 0.499$

Figure #2: Speedup vs. p



Question #3: Give reasons for the observed variation in the execution time as p is varied from 1 to 8192.

In this program, each thread calculates the minimum of a of a list that has approximately n/p elements in it (specifically, the portion of the big list of size n that was assigned to the thread on startup). After the calculations are performed, each thread updates the global minimum variable if that thread's local variable is less than the current global minimum. This portion of the program is serial in that only one thread can update the global variable at a time. The mutual exclusion of updates to the global variable is enforced through the use of a pthread mutex lock.

In order to update the global minimum before terminating, a thread must lock the mutex, update the global variable, and then unlock the mutex. While one of the threads is updating the global minimum, any other threads that are ready to access the same section of code must wait until the accessing thread is done (this is done by putting the waiting thread to sleep). When the accessing thread finishes, it wakes up one of the waiting threads and gives it control of the lock.

All of these lock and unlock operations (along with sleeping and waking up threads) introduce overhead into the program. The fact that all threads must do this before exiting, and that only one thread can do it at a time introduces a major serial bottleneck. While threads are waiting, they are not doing any useful work!

As the number of threads increases, the probability that they will run into each other (i.e. that they will want to access the lock at the same time) grows. The proportion of time spent waiting continues to increase as the number of threads increases (with the amount of useful work remaining the same).

On top of this, since this program uses pthreads, its execution space is limited to one board on ada. This means that there are a maximum of 20 CPU cores capable of running threads at any time. When the thread count is considerably larger than the number of available cores, it could take a long time for any of them to be scheduled (as is definitely the case at 8196 threads). The thread switches (and associated overhead) required to perform scheduling of all of the threads complicates this issue further.

Finally, the size of the list being checked is massive (in this instance, 200000000 integers = more than 700 megabytes, assuming that each word is 4 bytes. The size would be double if the words were 8 bytes long). In comparison, the caches on individual CPUs are rather small. In being shuffled around by the scheduler, newly scheduled threads could inadvertently evict the cache members of other threads that those sleeping threads were still using. If this was the case, then once the evicted thread was scheduled again, it would have to refill the cache, incurring many page faults in order to do so. This could potentially serve to greatly increase execution time due to the sluggishness of the memory hierarchy.

The three factors mentioned are most likely the cause of the significant slowdown that is witnessed as the number of threads increases beyond 1000. Interestingly, the overhead grows so large that it takes longer to run the program with tons of threads than it would be if you just used one. (I guess it goes to show that throwing tons of workers at a problem doesn't always make it better)

Part #2: Barrier

Question #4: Complete the function barrier_simple to implement a barrier among the threads. You will get full points if you add code only to the barrier_simple routine.

The thread barrier in this portion of the assignment functions as a synchronization mechanism for an arbitrary number of threads. When a thread reaches the barrier, it is put to sleep. No thread is woken up again until all (or a specific number of) threads reach the barrier. Afterwards, all threads continue through the barrier at the exact same time.

I implemented this functionality by adding code to the function: barrier_simple(). Each thread that calls barrier_simple first increments the count variable. This variable indicates the number of threads that have called barrier_simple since the start of the program. In order to prevent race conditions, this increment operation is protected by a mutex lock. Any thread must lock the mutex before incrementing, and then threads must unlock the mutex when done.

Inside the critical section, a check is also performed to see if the current count variable is equal to the number of threads. If it is not, this means that there are more threads that still need to reach the barrier before any waiting threads can continue. Consequently, the currently running thread will be put to sleep using pthread_cond_wait(). This makes the thread sleep until some condition wakes it up.

If the count is equal to the number of threads, that means that the current thread is the last thread to access the function. Based on the definition of the barrier, now that everyone has reached the barrier function, it is time to wake every other thread up and allow them to continue. The last thread uses the function pthread_cond_broadcast() to wake any sleeping threads. These threads are then woken up and continue to the end of the barrier_simple() function.

The code that I created for this function is as follows. It is also included in my submission within the file: barrier.c

```
void  barrier_simple ()
{
    // Lock the mutex so that count can be incremented.
    pthread_mutex_lock(& lock_barrier );

    // Increment count
    count = count + 1;

    // If this is true, it means that you are the last thread!
    if (count == num_threads)
    {
        // Signal all threads to tell them to wake up
        pthread_cond_broadcast (&cond_barrier );
```

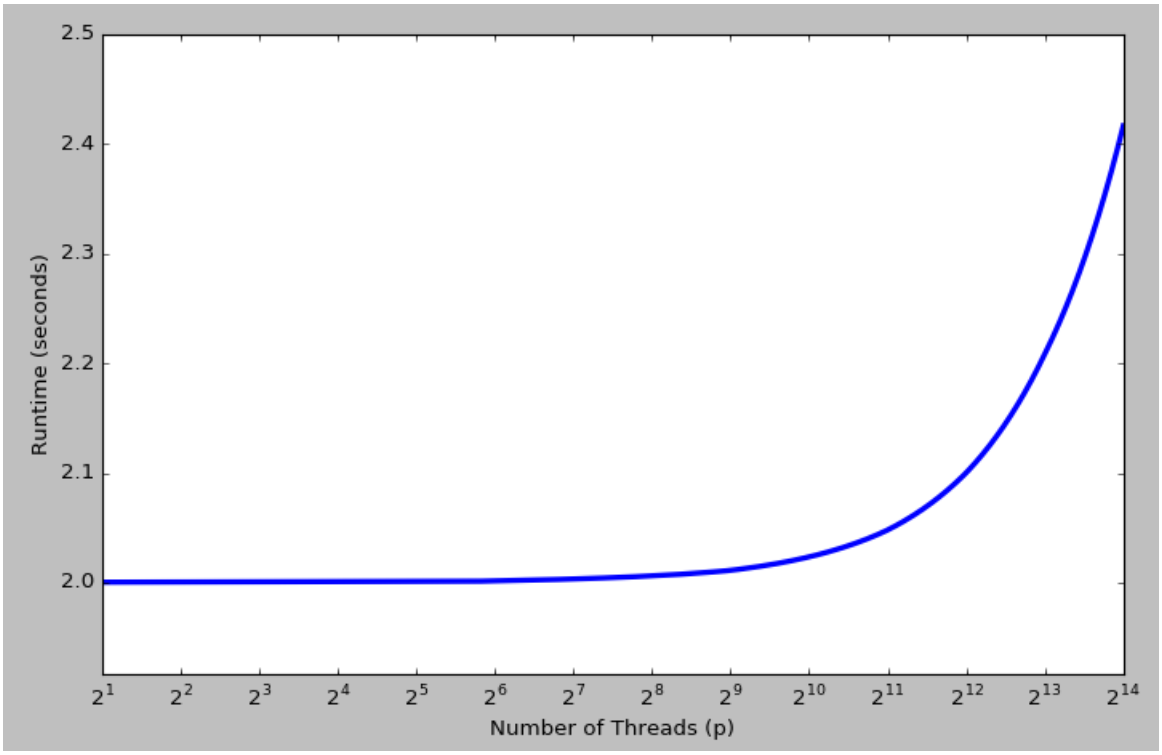
```
        // Unlock the mutex
        pthread_mutex_unlock(& lock_barrier );

        return;
    }

    // Put the thread to sleep until the others arrive
    pthread_cond_wait (&cond_barrier , & lock_barrier );

    pthread_mutex_unlock(& lock_barrier );

    return;
}
```

Question #5: Execute the code for $p = 2^k$ for $k = 1, \cdots, 14$. Plot execution time verses p to demonstrate how time varies with the number of threads. Use logarithmic scale for the x-axis.

Table #3: Barrier Data for Varying p

Number of Threads (p)	Runtime (seconds)
2	2.0004
4	2.0003
8	2.0003
16	2.0006
32	2.0008
64	2.0015
128	2.0034
256	2.0062
512	2.0112
1024	2.0235
2048	2.0480
4096	2.1002
8192	2.2085
16384	2.4165

Figure #3: p vs. Runtime for barrier.c

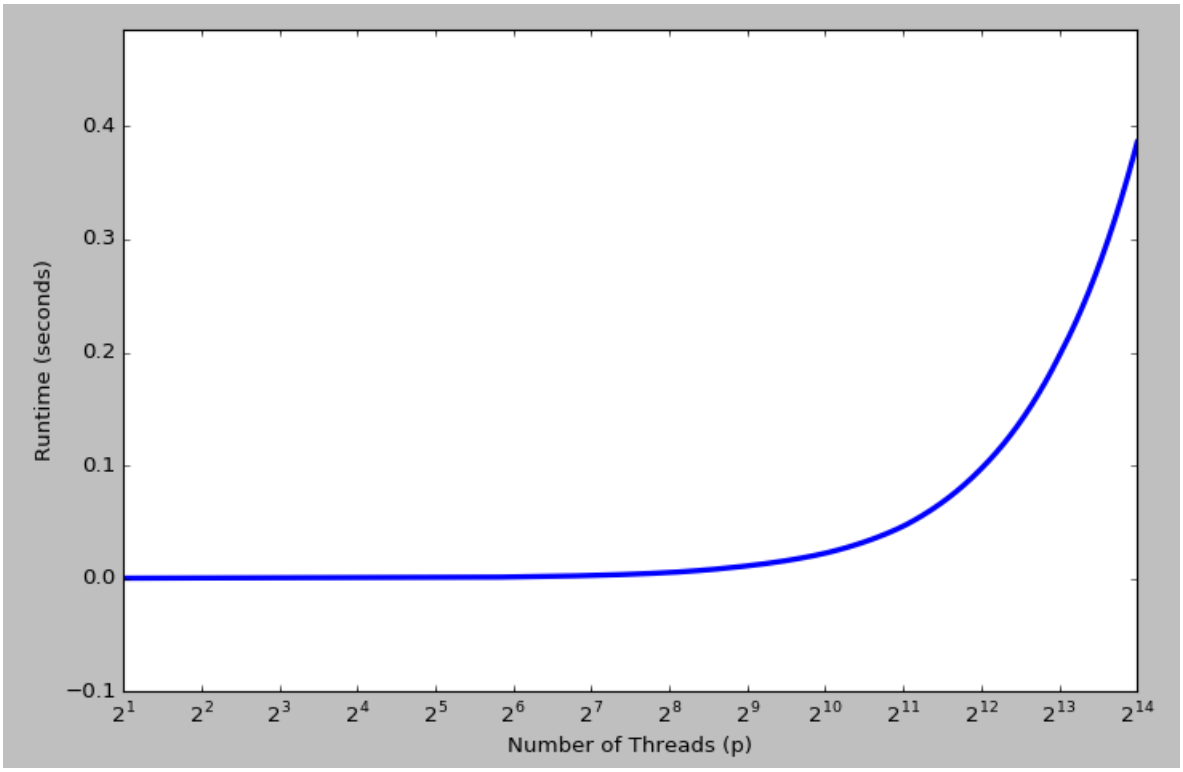


Question #6: Set sleptime.tv_sec = 0 and sleptime.tv_nsec=0 in csce435.h and run the experiments from the previous step again. Plot execution time verses p. How would you characterize the growth in time with p? What is the reason for such a growth?

Table #4: Barrier Runtime vs. p for tv_sec = 0

Number of Threads (p)	Runtime (seconds)
2	0.0003
4	0.0002
8	0.0003
16	0.0005
32	0.0008
64	0.0014
128	0.0027
256	0.0053
512	0.0112
1024	0.0224
2048	0.0463
4096	0.0971
8192	0.1970
16384	0.3859

Figure #4: p vs. Runtime (log scale)



With the sleep removed from the work() function, the program in barrier.c essentially devolves into a bunch of threads that synchronize with themselves before terminating. Since there is no useful work being done (and there's no sleeping either), almost all time spent in execution is spent performing the act of synchronization at the barrier.

From Table #4, shown above, a linear pattern seems to be apparent. Every time the number of threads doubles, the runtime also doubles. The log graph shown above doesn't necessarily show this relation fully, so a linear graph is provided below in Figure #5 (the homework handout asked for a log graph, so I'll just provide both for the sake of being complete). Figure #5 demonstrates a near perfect linear relationship between number of threads and runtime.

Each thread, during its execution does the following things:

- Starts at the beginning of start_func
- Call the work() function (does nothing in this instance)
- Call barrier_simple()
  - lock mutex
  - increment count
  - either wait or broadcast to wake up other threads
  - return
- Terminate with pthread_exit

Since the barrier simple function is almost entirely protected by a mutex lock, only one thread

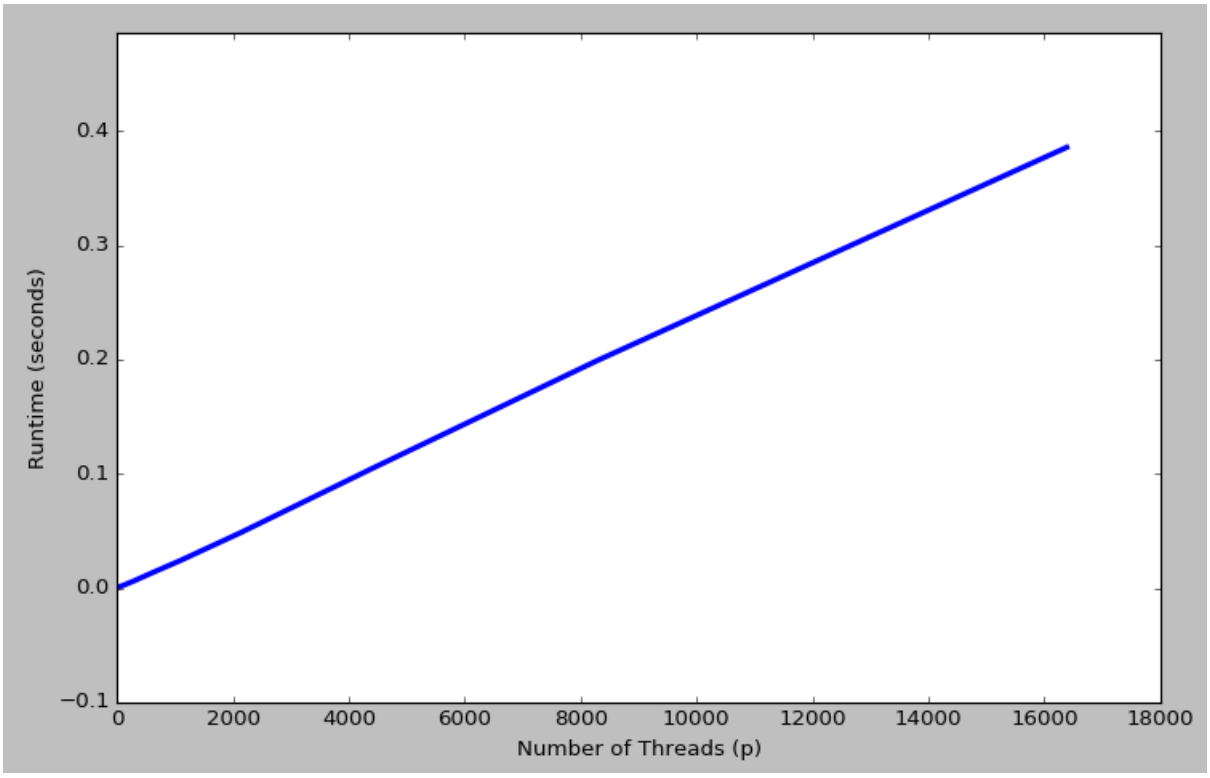
can be executing it at a time. All other threads are either hanging, waiting for control of the lock or have already made it through the critical section and are waiting on the conditional variable.

Because only one thread can run through the barrier function at a time, this portion of the program is essentially completely serial, with each thread executing one after the other in a non-deterministic order, sleeping before and after running the critical section of barrier_simple().

As a result of this serial execution, adding new threads to the program doesn't introduce very much overhead that will slow down existing threads (i.e. increasing the thread count doesn't reduce the speedup as significantly as in previous examples that have been enumerated in this assignment). Even scheduling overhead isn't a problem because only one CPU core is in use while all of the threads are waiting to execute the barrier simple function.

Consequently, the only costs of using $p+1$ threads instead of p threads are creating the thread + the time to execute the barrier simple function + the time to terminate the thread. Because these execution steps won't interfere with other threads significantly, this time is simply added on to the existing runtime, resulting in a linear increase.

Figure #5: p vs. Runtime (linear scale)



Part #3: List Statistics

Question #7: Modify the program in list_minimum.c so that it computes the mean and standard deviation of the list elements instead of the minimum. Name the new program file list_statistics.c. You may define global variables mean and standard_deviation that will store the values. You will get full credit only if the mean and standard deviation of the list are computed by threads before exiting the thread routine.

The mathematical formulas for computing mean and standard deviation are as follows:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \quad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Note that the equation for the standard deviation requires knowledge of the mean. In order to do this, my program implements the following series of steps:

1. Each thread begins by summing up every member within its sublist. At the end of this step, it adds this to a global variable named mean. (The mean variable's final purpose will be to contain the mean of all of the elements of the list. Right now, it's just being used to contain sums) After every thread performs this action, the mean variable contains the sum of every single element in the list. This section is protected by a mutex lock so that there are no race conditions when trying to access the global variable.
2. The last thread to perform the summing of the list also computes the final value for the mean by running: mean = mean / list_size. After this point, the global variable named mean contains the average of all of the elements in the list.
3. After the mean is calculated, all threads are synchronized at a barrier (very similar to the one which was implemented in part #2 of this assignment). All of the threads wait here until the last thread makes it to this point. The barrier is necessary because calculations for standard deviation require that the mean be calculated beforehand.
4. After exiting the barrier, every thread iterates through its own portion of the list again, computing a part of the standard deviation calculation. Specifically: $(x_i - \mu)^2$ for each x_i in that thread's sublist.
5. Once the thread has calculated this information, it adds it to a global variable. This global variable, named standard_deviation, will eventually contain the final standard deviation. For now, it just contains the sums that all of the individual threads are running. It is protected by a mutex lock to avoid race conditions.
6. The last thread to execute this function finishes the standard deviation calculation by executing: standard_deviation = sqrt(standard_deviation / list_size). The global variable standard_deviation now contains the actual standard deviation of the list.
7. Finally, the threads are synchronized at another barrier. This one ensures that the standard deviation is fully calculated before any threads are allowed to exit the thread function.

Note: I also modified the main function so that it computes the mean and standard deviation with only one thread. I use these values as comparisons at the bottom of the main function in order to ensure correctness. (i.e. if the true mean or true standard deviation are incorrect, then an error message is printed out.)