Parallel Computing Assignment 1

Rachel Hwang

January 28, 2014

1 Final Design

While my serial implementation of the algorithm did not differ much from my original design plans, my parallel implementation changed notiably.

First, I eliminated the need for the k-loop entirely simply by giving each thread its own k variable and managing when it would be imcremented. As stated in my original design document, each thread handled one row (doing the i-loop) at a time, and the current row is kept track of using a counter variable. Whenever the counter exceeds the n value, a thread knows that all rows for this k value have been taken, and so the thread increments it's k variable and waits for the other threads to finish.

In my original design, I simply spawned and joined threads for every k interation, which would have offered significantly worse performance given the overhead of creating threads, as was discussed in class. In my final implementation, I instead used the same set of threads for the entire run of the algorithm, using pthread barriers to keep things synchronized. Two barriers are needed. One barrier waits until all work threads are finished. The second waits for the main thread to reset the row counter to 0. Using two barriers ensures that no thread moves along to the next k interation before the others and that the counter is reset only once all work for one k iteration is done, but before the next one starts. Below are the relevant snippets of code. The second snippet is the work loop of the thread function (doing a single row) and the first snippet is the main loop which keeps the counter, the only central information, value set properly. This implementation keeps shared information to a minimum by relying on the logic of the program to update each thread's information periodically. Notice that the matrix a need not be locked since threads can never be working on the same row at the same time.

```
// From floyd_parallel.c main()

  // Spawn threads
  thread(num, n, a, thr, thr_data);

  // Floyd-Warshall Algorithm
  k = 0;
  while(k < n) {
    pthread_barrier_wait(&b1);
    k++;
    pthread_mutex_lock(&c_lock);
    counter = 0;
    pthread_mutex_unlock(&c_lock);
    pthread_barrier_wait(&b2);
  }

  // Join threads
  for (i = 0; i < num; i++) {
    pthread_join(thr[i], NULL);
  }
```

```
// From funcs.c thr_row()
 while (1) {
   // get next row from counter
   pthread_mutex_lock(&c_lock);
   i = counter;
   counter++;
   pthread_mutex_unlock(&c_lock);


   if (i < n) {
     if (i != k) {
       for (j = 0; j < n; j++) {
         if (j != k) {
           if ((a[i*n+k] + a[k*n+j]) < a[i*n+j])
             a[i*n + j] = a[i*n+k] + a[k*n+j];
         }
       }
     }
   } else {
     k++;
     pthread_barrier_wait(&b1);
     pthread_barrier_wait(&b2);
     if (k == n) {
       pthread_exit(NULL);
     }
   }
 }
```

These major changes aside, I also eliminated my transposition step from both the serial and parallel programs, as I found that the performance increase did not seem worth the cost of maintaining two copies of the matrix and the added complexity of the program. Unfortunately, I was not able to collect data for the transposed version.

2 Parallel Overhead

Table 1: Comparision of serial and parallel code with t = 1 (milliseconds)

n	serial	parallel
16	0.0287 ± 0.0004	0.8232 ± 0.0198
32	0.1949 ± 0.0011	1.4215 ± 0.0439
64	1.3129 ± 0.0296	3.7102 ± 0.0242
128	9.0796 ± 0.3020	16.4220 ± 0.2927
256	26.4610 ± 1.3115	61.2174 ± 2.5181
512	192.9074 ± 2.0926	310.8201 ± 2.0641
1024	1494.7179 ± 9.9010	2207.7326 ± 23.1666

The overhead incurred in the parallel implementation behaved about as expected. In the first graph, it is evident that overhead cost is more or less flat, which is what one would expect from my implementation. Threads are spawned only once, and so the expensive set of thread creation operations take approxiately the same ammount of time at any n value as long as the t value is fixed.

Looking at the second graph, which shows the ratio of serial to parallel implementation performance, the flat cost is reflected in the upward relative performance trend. As n increases, the flat threading overhead cost remains the same, and therefore composes a smaller and smaller percentage of overall

performance. Thus, the parallel implementaion does better with larger input sizes. My hypothesis was correct in that regard. The increase in performance is sharp initially, then slowly flattens out as the proportion of overhead time to total time becomes smaller and smaller, thus less significant.

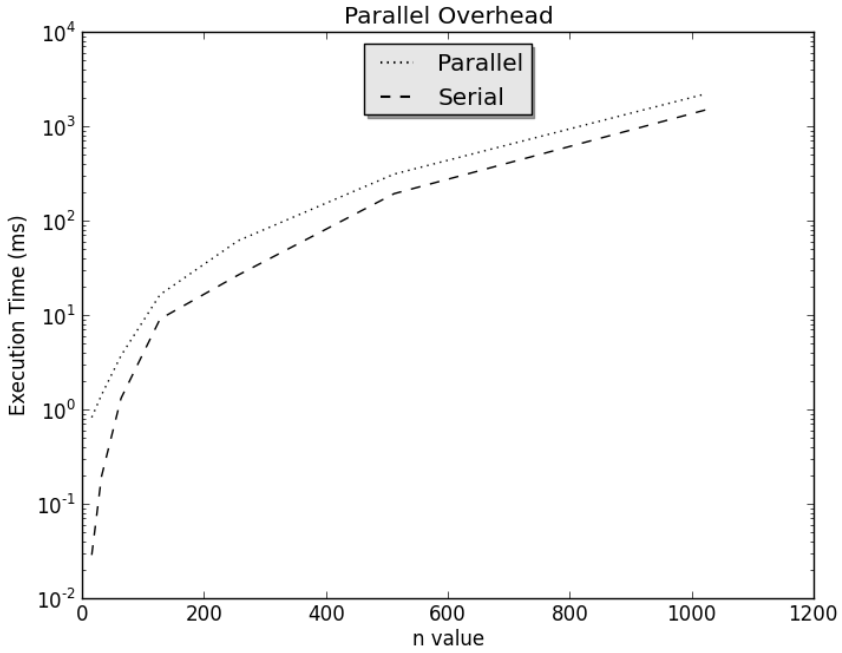

Figure 1: This figure illustrates the flat overhead cost of using threads. The parallel and serial performance difference stays approximately the same for all n values

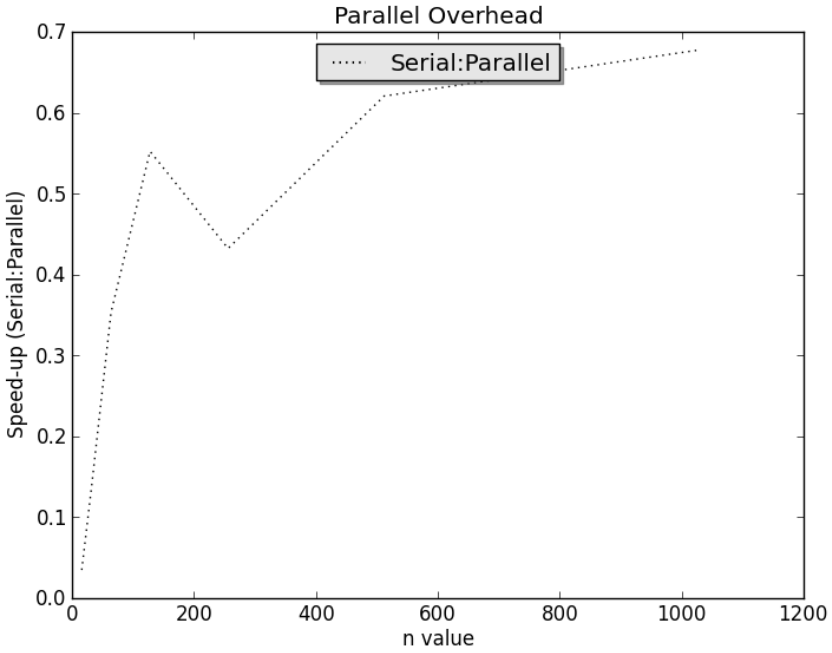

Figure 2: This figure illustrates how overhead becomes less significant as n and overall time increases.

I was unable to determine why the speedup dips at n=256, despite taking averages of multiple runs.

3 Parallel Speedup

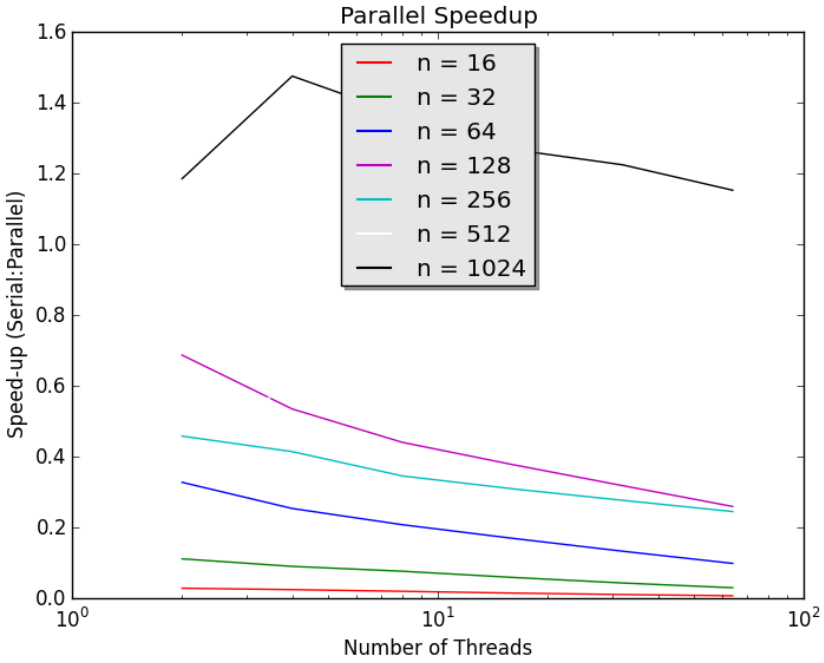

Figure 3: This figure illustrates the performance of the parallel implementation at different t values. Only the largest n value affords any speedup!

The overall performance boost from parallelizing was smaller that I expected. Only n=1024 provided any improvements in performance. All smaller input sizes meant worse than serial execution time for all t values, which surprised me. For all input sizes, there was also a downward thrend in performance as t increases. This means that the additional overhead cost of adding one more thread is generally negates the benefits of increased parallelization. This suggests that my implementation is inefficient, although it may be possible that performance would continue to improve with larger n values. Given more time, I would also continue testing increasing n values.

While I was correct in hypothesizing that performance would only improve above a certain n value, I did not expect it to be so large, nor did I expect the overall performance increases to be so minimal. The best n and t values do not even offer a 2x speedup! I was quite disappointed.

By Amdahl's law, this means that my parallel implmentation was only able to parallelize a small portion of my code, only about 45% when n=1024, t=4 ($1.5 = \frac{1}{1-p+p/4 \to p} \approx .44$). In other words, the infrastructure I added in order to implement the threaded version took a significant ammount of time compared to the time saved by threading. I suspect that the many extra conditional statements in the main loop of my thread function may be responsible for the extra time, as well as the two barriers.

Had I extra time, I would try an entirely different parallel implementation, perhaps using pthread mutexes instead of barriers to lock entire threads. Simply blocking control to threads entirely may be more time effective.

Table 2: Parallel code performance for different values of n and t

n	t	time
16	2	1.0198 ± 0.0347
16	4	1.1857 ± 0.1710
16	8	1.4747 ± 0.4326
16	16	1.9846 ± 0.9600
16	32	2.7913 ± 1.8279
16	64	4.1501 ± 3.4678
32	2	1.7534 ± 0.0441
32	4	2.1675 ± 0.4172
32	8	2.5584 ± 0.6586
32	16	3.3169 ± 1.4338
32	32	4.5361 ± 2.7567
32	64	6.5999 ± 5.2572
64	2	4.0023 ± 0.0461
64	4	5.1694 ± 1.1798
64	8	6.3098 ± 1.8838
64	16	7.7438 ± 2.9847
64	32	9.8697 ± 5.0361
64	64	13.3250 ± 8.9931
128	2	13.2069 ± 0.1317
128	4	16.9690 ± 3.7643
128	8	20.6017 ± 6.0038
128	16	24.0271 ± 8.1626
128	32	28.5269 ± 11.8930
128	64	35.0065 ± 18.1782
256	2	57.7942 ± 13.5519
256	4	63.9056 ± 11.4191
256	8	76.6207 ± 20.2666
256	16	85.6262 ± 24.9048
256	32	95.6942 ± 32.6077
256	64	108.1073 ± 41.1169
512	2	399.4916 ± 125.8115
512	4	327.6973 ± 114.6437
512	8	349.0607 ± 98.5818
512	16	361.9362 ± 96.9322
512	32	379.2740 ± 101.0530
512	64	403.5378 ± 112.9007
1024	2	1261.2786 ± 23.3571
1024	4	1013.4842 ± 260.1000
1024	8	1095.4018 ± 248.2997
1024	16	1179.2270 ± 271.8279
1024	32	1220.8479 ± 264.2468
1024	64	1296.9553 ± 307.0843

- Exercise 2
  1. Safety. The "bad thing" avoided is patrons being served out of order.
  2. Liveness. The "good thing" ensured is that things will eventually come back down.
  3. Liveness. The "good thing" ensured is that one process enters its critical section, and both are not simply waiting forever.
  4. Safety. The "bad thing" avoided is that an interrrupt will never go more than a second unacknowledged.
  5. Liveness. The "good thing" ensured is that a message will be printed eventually.
  6. cost of living
  7. Safety. The "bad things" avoided are immmortality and government bankruptcy.
  8. Liveness. The "good thing" ensured is that you will eventually indentify man from harvard.

- Exercise 3
  Bob and Alice can use a two can system in which they each have a can on their window sil. Alice and Bob begin by setting his can upright. If Bob's can is upright, he pulls Alice's string and enters the yard. When he's finished, he crosses the yard and resets Alice's can and leaves the yard. If Alice wants to use the yard, she must first check if her own can is down. If not, she is free to use to yard. She enters the yard, pull's Bob's can and reset's it when she is finished. Essentially, each person's can represents their right to claim the yard. Each person basically locks the other while using the yard. and unlocks the other person's can, but not their own, when they are down. This ensures that each person uses the yard only when the other person has signaled that is time for them to do so by resetting their can.

- Exercise 4
  If the initial state of the switch is known to be off, the prisoners select a leader. This leader only ever turns the switch on. If the switch is already on, he does nothing. Each other prisoner turns the switch off if it is on and does nothing if it is off already. The leader merely counts the nubmer of times he has flipped the light on. After the (p-1)th switch on, where p is the number of prisoners, he knows that every other prisoner must have visited the room and flipped the switch once. and can claim as such.

  If the initial state of the switch is unknown, the strategy is the same except everyone (minus the leader) must flip the switch on twice. The leader waits until she has flipped the switch off 2p-2 times. At 2p-2 on lights, either every (non-leader) person has turned on the light twice or every (non-leader) - 1 has touched the light and it was initially on.

- Exercise 5
  The prisoner at the back of the line cannot be saved. His job is to transmit binary information to the rest of the group. Beforehand, the prisoners can agree that "blue" will mean that there are an odd number of hats, and "red" will mean an even number.

  Let us show that this method works by induction. Base case: the penultimate man in line will recieve information, even or odd blue hats, and since he can see all the hats before him, can derive the color of his own hat.

Assuming that for any man, all the hat colors before him have been stated correctly by the time his turn is reached, the man will know the color of every hat apart from his own (and excluding the last man), since he can also see all the hats before him. If he knows whether there are an odd or even number of blue hats in total, he can add the number of blue hats already called to the number of blue hats he sees. If this sum has the property (odd/even) that the last man named, then this man's hat must be red. If the sum does not, his hat must be blue. By induction, this strategy allows all but the last prisoner to know his hat color, so long as P¿2.

- Exercise 7

$$\frac{1}{1-p+p/2} = S_2$$
$$1 - 1.5p = \frac{1}{S_2}$$
$$p = \frac{2(1-1/S_2)}{3}$$

This p value is the proportion of parallizable program. So this can be plugged back into Amdahl's law like so:

$$p = \frac{2(1-1/S_2)}{3}$$
$$S_n = \frac{1}{1-(2(1-1/S_2)/3) + \frac{(2(1-1/S_2)/3)}{n}}$$

- Exercise 8
We want a speedup for at least 5x in order for the multiprocessor to be worth it. Using Amdahl's law, for S = 5 and n = 10:

$$5 = \frac{1}{1-p+p/10}$$
$$p = 8/9$$

Therefore the portional of an application that is parallizable must be at least $\approx 89\%$ to be worth the multiprocessor machine.

- Exercise 11
Mutual Exclusion: satisfied. Since the outer loop invariant only allows threads to exit the loop if it is their turn, and turn can only have one value, only one thread may exit the lock at one time. All other threads are placed in the inner loop to wait until the lock is not busy. Thus, only one thread can take its turn at a time and so thread critical sections can be never allowed to overlap if lock is called before and after. Starvation-free: not satisfied. Consider the case where several threads call lock at about the same time. Thread 1 gets the turn and threads 2 and 3 are left waiting in the inner loop. Once thread 1 calls unlock, thread 2 jumps and grabs the turn. When the lock is next free, thread 1 gets back into the waiting loop and if thread 3 is unlucky, could once again grab the turn before thread 3 has an opportunity. There is nothing preventing thread 3 from never getting a turn, and so if conditions are unfavorable, it may be starved out.

Deadlock-free: satisfied. So long as unlock is called exactly as many times as lock is called, there will never be any dead lock. This code has no competing conditions which would allow that to occur. As in our discussion of Mutual Exclusion, only one thread may have its turn at once; the remaining threads are left waiting in the inner loop. Thus, as soon as the lock is unlocked, the next thread may proceed to grab control and the process continues. So any thread which calls lock must either make it through or be stalled in the inner loop, a waiting process that has a definite endpoint so clong as unlock is eventually called.

- Exercise 14
  sdf

- Exercise 15
  sdf

- Exercise 16
  sdf