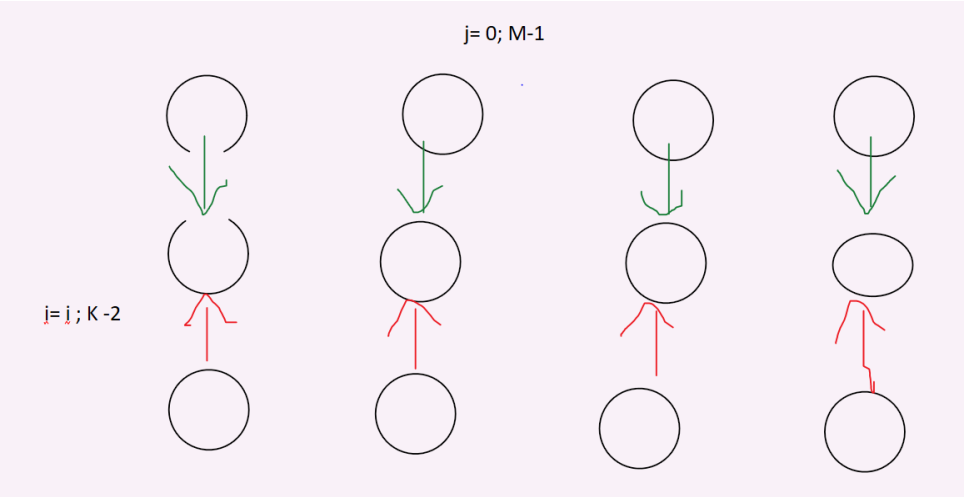
Cyndy Ishida

Homework #3

Problem 1:

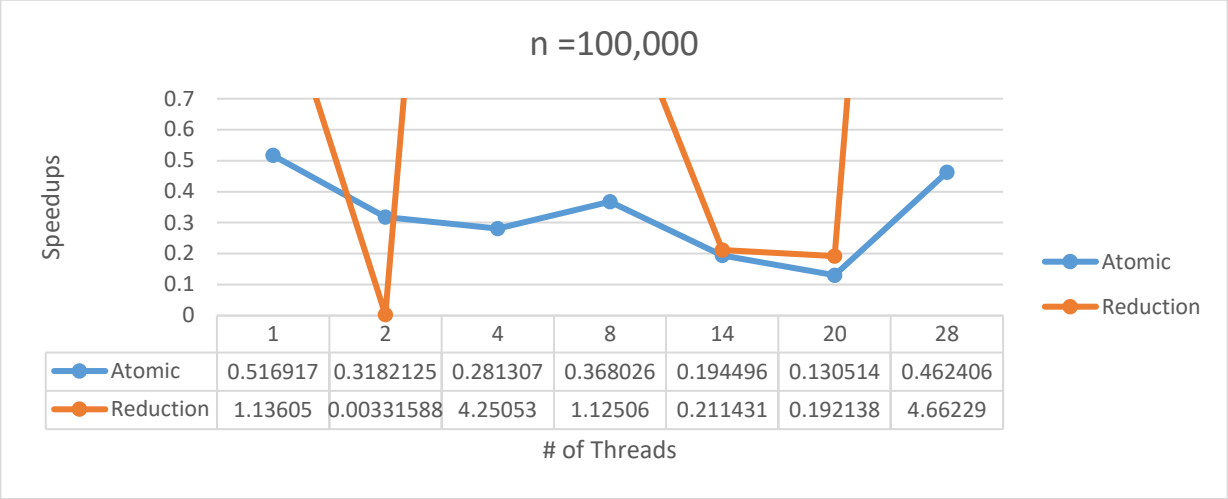
i. S1 flow dependent to S3, its ok, in loop
S2 flow dependent to S3, its ok, in loop
S1 flow dependent to S3
S2 anti dependent to S3



ii. Yes.
Because All of the “i” iterations are able to start in dependently from each other.

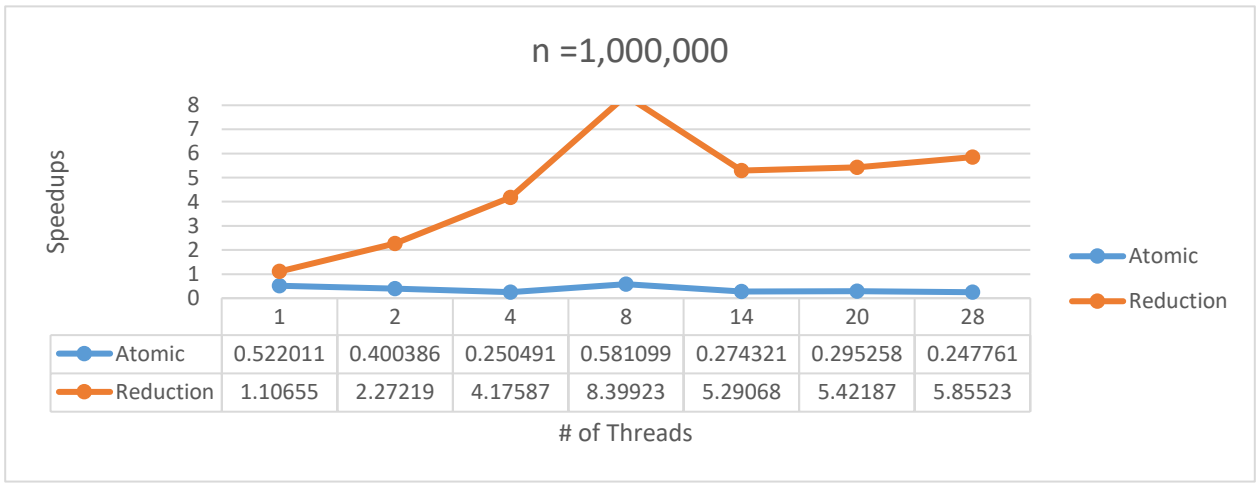
```
# pragma omp parallel for default (none) private(i, M , a1, a2, j,A) shared(K)
for (int i = 1; i <K-1; i++)
    for (int j = 0; j <M; j++)
    {
        int a1 = A[i-1][j];
        int a2 = A[i+1][j];
        A[i][j] = (a1+a2)/2;
    }
```

Problem2: Monte Carlo PI Approximation



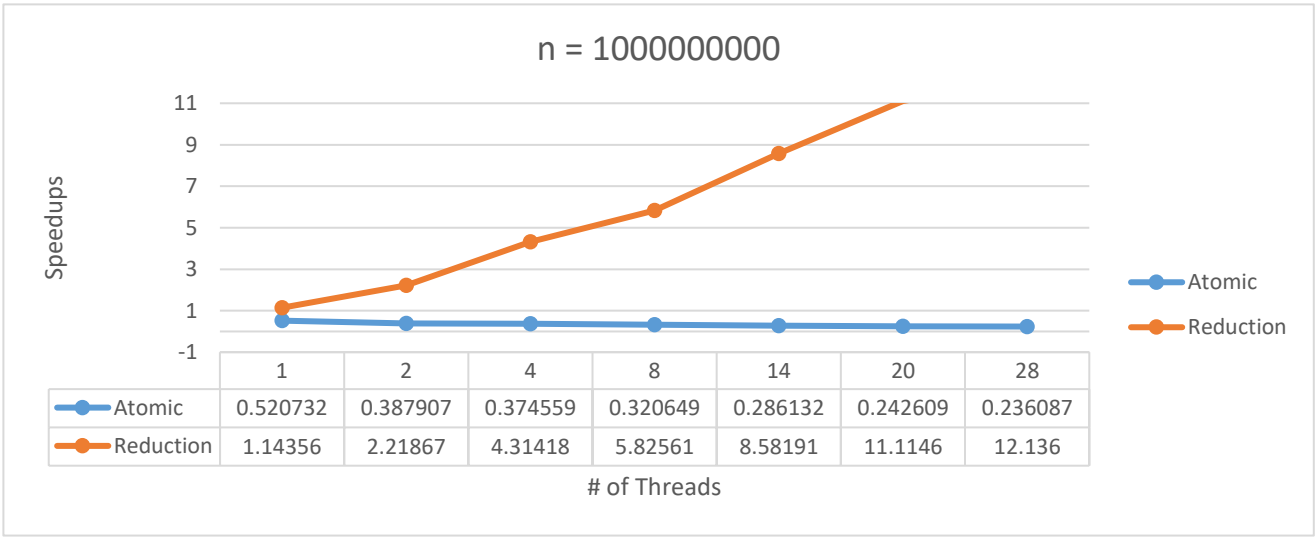
	1	2	4	8	14	20	28
Atomic	0.516917	0.3182125	0.281307	0.368026	0.194496	0.130514	0.462406
Reduction	1.13605	0.00331588	4.25053	1.12506	0.211431	0.192138	4.66229

Figure 1



	1	2	4	8	14	20	28
Atomic	0.522011	0.400386	0.250491	0.581099	0.274321	0.295258	0.247761
Reduction	1.10655	2.27219	4.17587	8.39923	5.29068	5.42187	5.85523

Figure 2

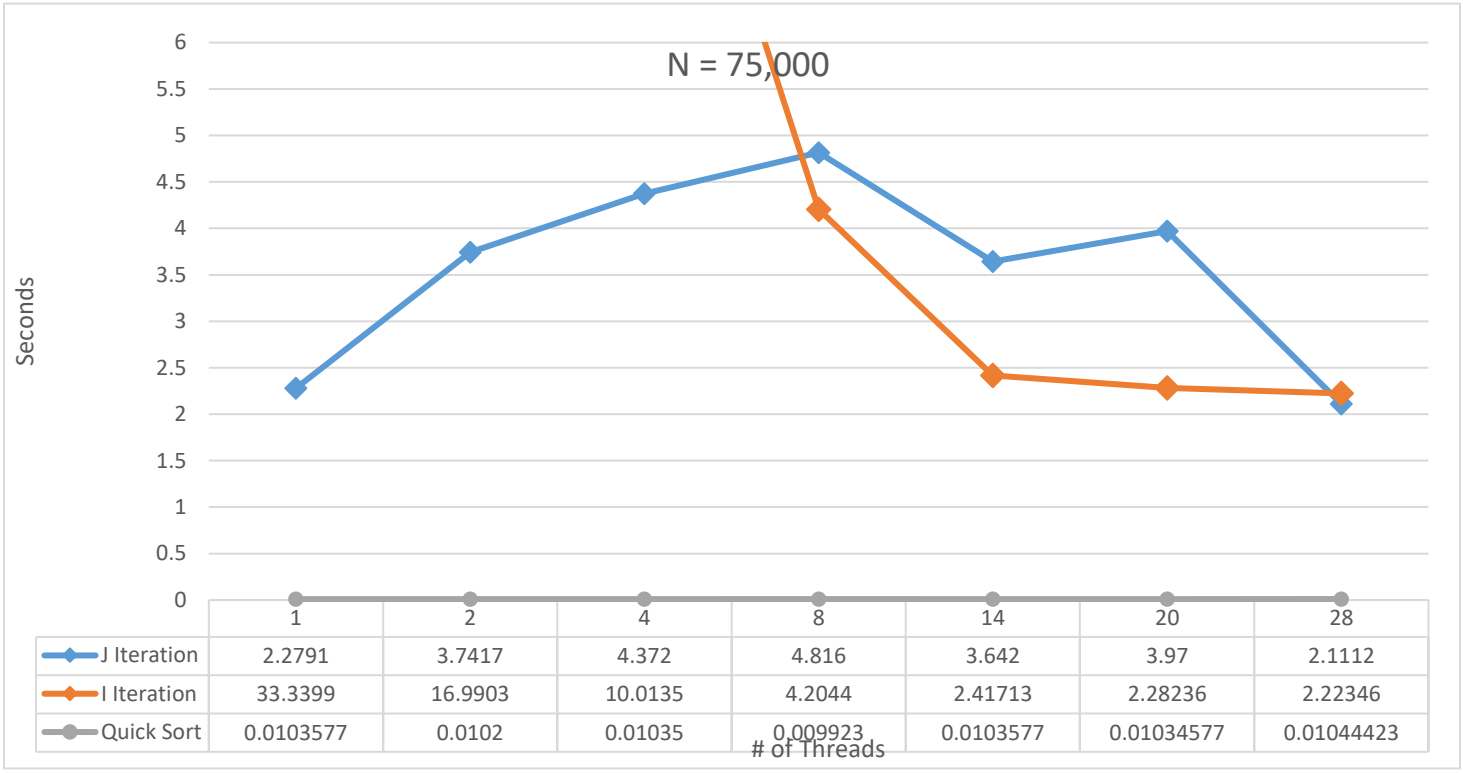


	1	2	4	8	14	20	28
Atomic	0.520732	0.387907	0.374559	0.320649	0.286132	0.242609	0.236087
Reduction	1.14356	2.21867	4.31418	5.82561	8.58191	11.1146	12.136

Figure 3

ii.) For the first n size, shown in figure 1, the relative speedups to the sequential code are within the 0 and 1 seconds. On the chart, it can be shown that the reduction directive has a more significant speedup, which means it's the more efficient way of parallelizing source code. For n at 1 million, the speed up range from 0 to 8 and the reduction speed up slope is growing linearly to the number of threads where the atomic is staying relatively constant. Finally, for n at 1 billion the significance change is between 0 and 12. It is clear to see that as n grows, so does the efficiency of using the reduction directive in openMP in comparison to the atomic directive. For the random seed, I used the i variable since it is a private variable for each thread. I originally tried to do the seed as the thread id, but it would sometimes result in an invalid computation, where choosing i, always generates valid approximations.

Problem3 : Count Sort Algorithm

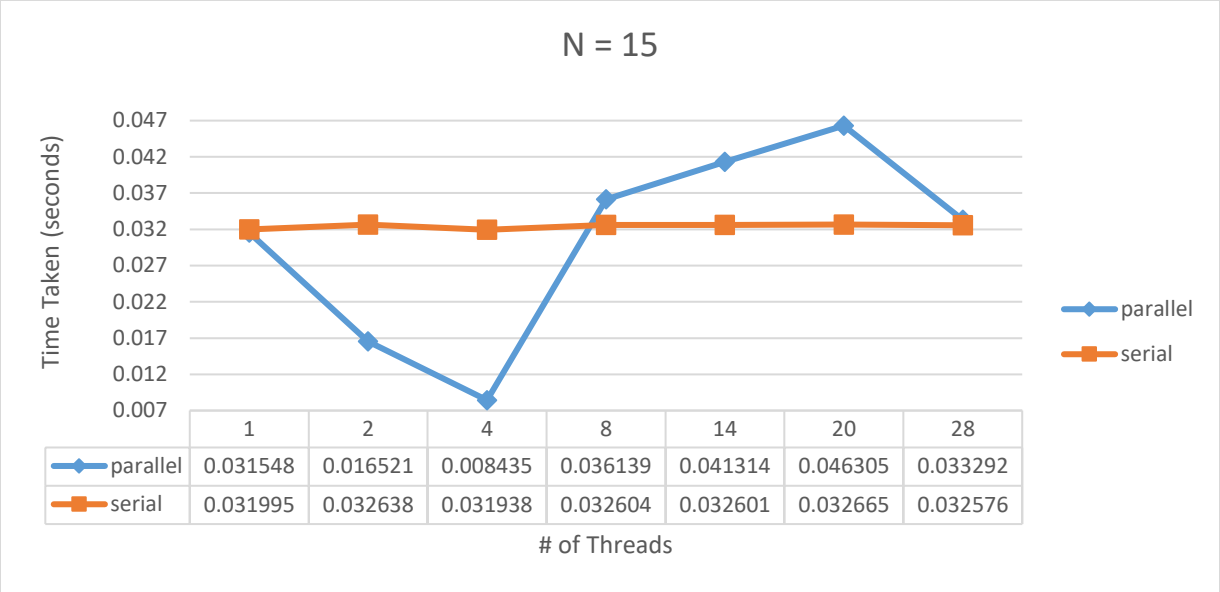


	1	2	4	8	14	20	28
J Iteration	2.2791	3.7417	4.372	4.816	3.642	3.97	2.1112
I Iteration	33.3399	16.9903	10.0135	4.2044	2.41713	2.28236	2.22346
Quick Sort	0.0103577	0.0102	0.01035	0.009923	0.0103577	0.01034577	0.01044423

Figure 1

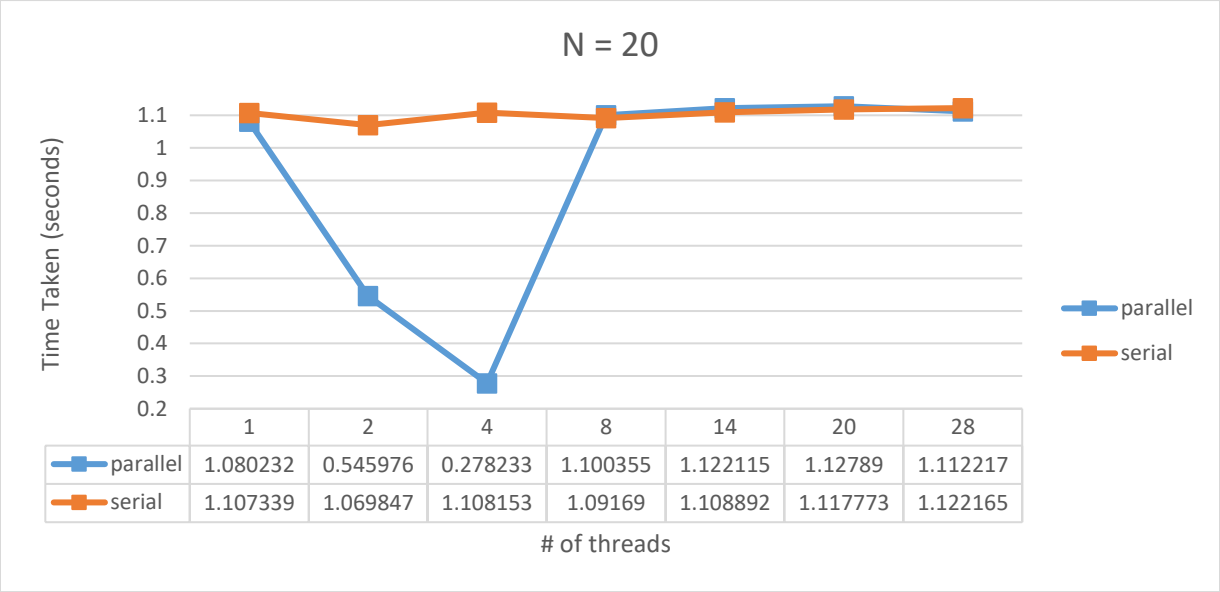
iii.) Both I and J parallelized versions run slower overall in comparison to the sequential quick sort. I believe this makes sense because even if you parallelize an n^2 algorithm, only the constants will change but the overarching asymptotic time complexity will stay the same. After a certain cross over point, as N increases the nlogn algorithm will continue to run quicker than the n^2 algorithm. Here at 75,000 elements, the speed up is nonexistent in the parallelized code and the j loop runs slower than the i loop linearly until we reach 28 threads, where the i loop speed down is less significant. I'm not exactly sure why. As for the parallelized memcpy, I just created a for loop, and set the contents of an assigned to the contents of temp.

Bonus Problem: Weight Balancing



	1	2	4	8	14	20	28
parallel	0.031548	0.016521	0.008435	0.036139	0.041314	0.046305	0.033292
serial	0.031995	0.032638	0.031938	0.032604	0.032601	0.032665	0.032576

Figure 1



	1	2	4	8	14	20	28
parallel	1.080232	0.545976	0.278233	1.100355	1.122115	1.12789	1.112217
serial	1.107339	1.069847	1.108153	1.09169	1.108892	1.117773	1.122165

Figure 2

For this problem, I originally believed that the way of not generating too many tasks that created the parallelization code slower in all cases. Something that I decided to do is once my global min has moved to a small weight, I called the serial code, because the overhead of paralleling was no longer needed.

With a consistent N value, thread's with around 2 and 4 have noticeably large speedups. One thing I decided to do, is to limit the number of tasks that can run in parallel. I set that to an arbitrary value of 5. That speed up around thread's close to 5 are greatly significant, with the if clause in the task directive. I

also used the mergeable directive to try to merge any tasks to relieve work for the individual executions. All other thread number arguments, tend to have a small speed down, or practically same speed.

I'm still kind of confused of how that determines such optimal results. In both figures, 4 threads were the number that created such optimal results. I think it should do with how I implemented the if clause. I used a qsub script to grab results, so I believed it wasn't noisy data, however I can't explain what caused figure 1's larger time for 8 thread's spike, and what made figure 2's threads to follow consistently with the serial code.