Dominic DiCarlo
Parallel Computing
Assignment 1
Final Write-up

Final Implementation Details:

The final implementation was as described in the original design write-up, with the exception that the number of semaphores used was reduced by a factor of No. of threads. The adjacency matrix is divided up into sequences of rows, which are assigned to threads which compute that part of the for (i = 0; i < vertex_count; i++) loop. The i loop for each thread was thus adjusted to be the following:

```
for (i = this_threads_start_row; i <= this_threads_end_row; i++)
```

Rows are divided up as equally as possible, by dividing the total number of rows by the number of threads and assigning each thread the quotient. If the number doesn't divide with remainder == 0, then the remainder is divided to the first *remainder* threads such that each of those threads has one more row to process than the others. Threads are assigned only neighboring rows, to optimize spatial locality for caching.

The algorithm requires that parallel threads can access an up-to-date version of each matrix value they use in computation. This is fine for the i,j and i,k terms, since these values are always computed by the same thread and so must be up-to-date. For the k,j term, this represents a concurrency issue since there may be many *k* values for which that thread must rely on other threads' updating abilities.

To address this, semaphores were employed. Specifically, an array of size No. of vertices + 1 was used, assigning a semaphore to each row. This is because only the *kth* row needs to be accessed on iteration *k*, so we only need to lock access to each row for only one iteration. All semaphores were initialized at 0, except for the k = 0 semaphore (since for the first iteration all threads work from the original input matrix). When a thread finishes its calculation on a row *i* for iteration *k*, where i == k, it increments the semaphore for that row No. of threads times. Whenever a thread starts a new iteration in the k loop, it calls sem_wait on the semaphore for row k. This is so a thread on iteration *k* ensures it is accessing at least the *k* - 1 iteration value of a matrix. Note that it doesn't matter if the matrix value is even ahead of this at *k* or *k* + *n* .[1] A total of No. of threads will decrement each semaphore (when each thread

[1] To see why this is the case, first note that the computations are such that a value is updated by taking the minimum of i,j and i,k + k,j. In other words, the minimum is between the current value for the minimum distance between i and j and another path i and j, namely through k. If this algorithm found a distance smaller than the serial algorithm, it would only show that the serial algorithm is incorrect, since the parallel version is still choosing valid paths.

reaches the particular k value corresponding to each semaphore), which is why it must be incremented No. of threads times.

After a thread finishes its allotted rows, it simply calls pthread_exit, with the exception of one thread which joins all the threads in the end.

Data:

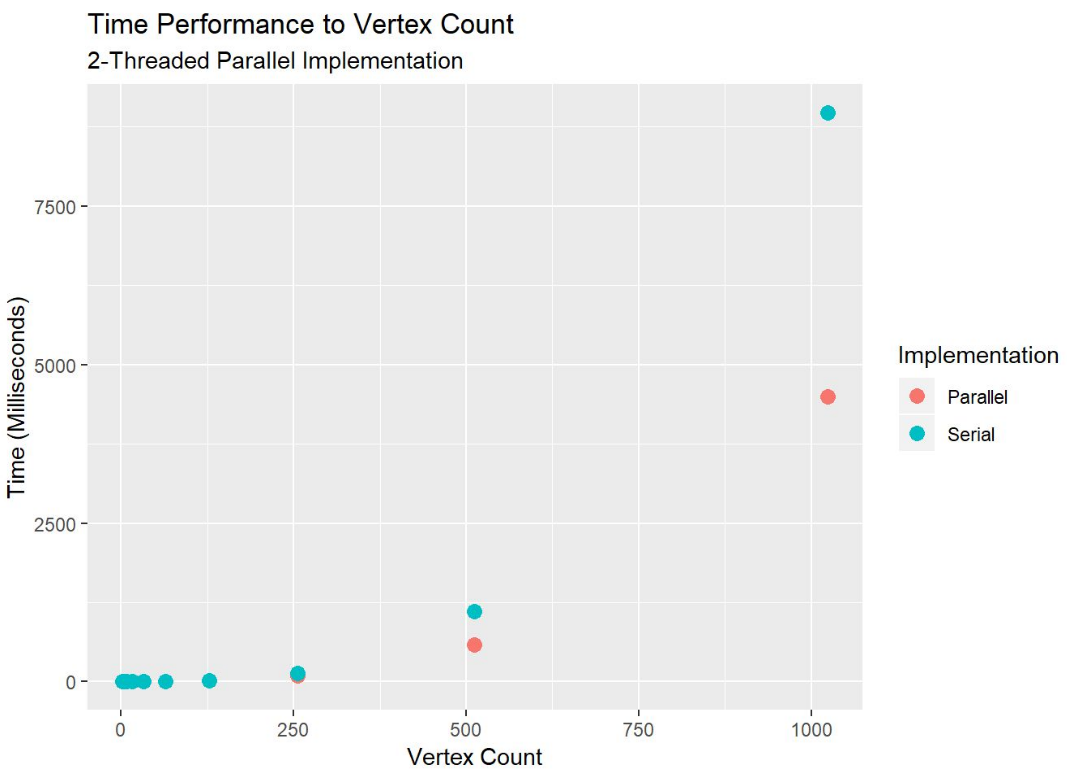
All values are time in milliseconds.

Serial Data for No. of Vertices ϵ [2,4,6,8,16,32,64,128,256,512,1024]

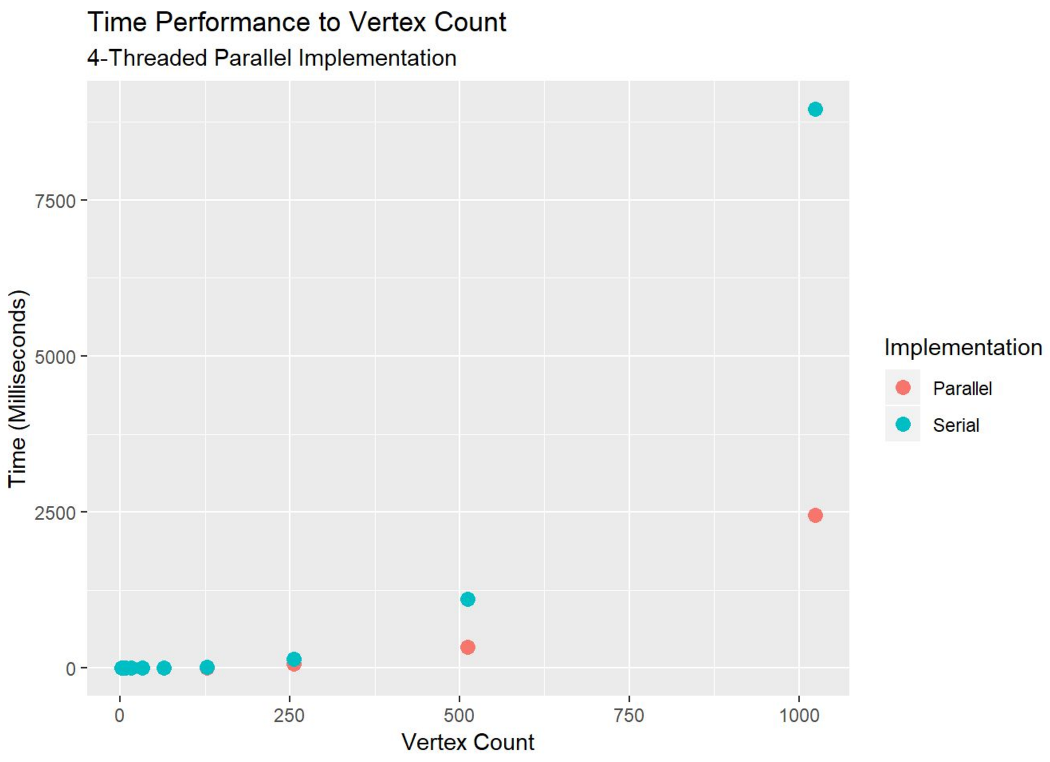
2	4	8	16	32	64	128	256	512	1024
0.001	0.004	0.017	0.061	0.925	6.239	22.785	140.6280	1121.021	8958.8964

Parallel Data for No. of Vertices ϵ [2,4,6,8,16,32,64,128,256,512,1024]
And No. of Threads ϵ [2,4,6,8,16,32,64]

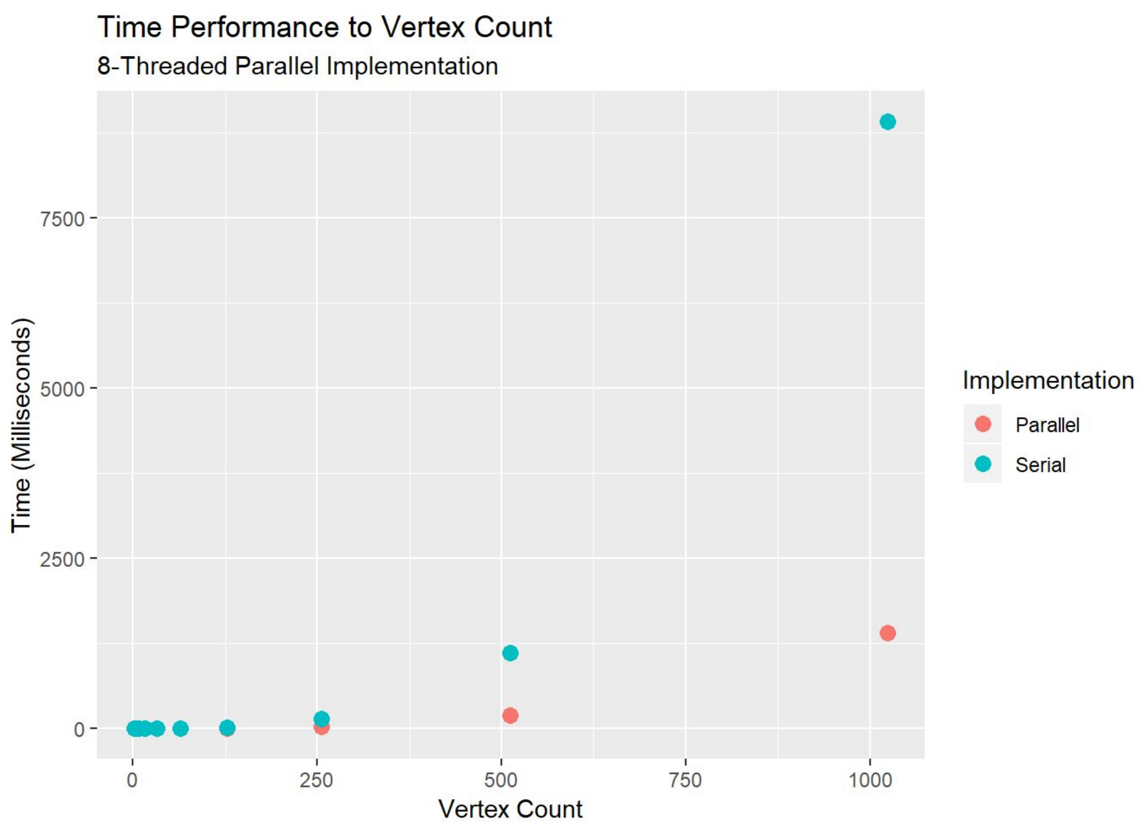
	Thread Count →					
Vertex Count	2	4	8	16	32	64
2	0.341	0.398	0.288	0.43	0.136	0.229
4	0.279	0.59	0.521	0.411	0.4	0.352
8	0.255	0.412	0.881	0.783	0.642	0.866
16	0.479	0.435	0.931	1.509	1.106	1.789
32	1.719	1.184	1.279	2.375	3.866	3.868
64	9.107	5.2	4.016	5.272	8.912	10.845
128	41.598	25.521999	16.905001	13.271	17.839001	26.525
256	217.869003	130.516998	85.058998	51.153999	74.732002	88.668999
512	1615.682983	874.294006	549.008972	286.306	315.701996	378.813995
1024	12860.20215	6561.037109	3788.462891	2179.039062	2205.406006	2381.789062

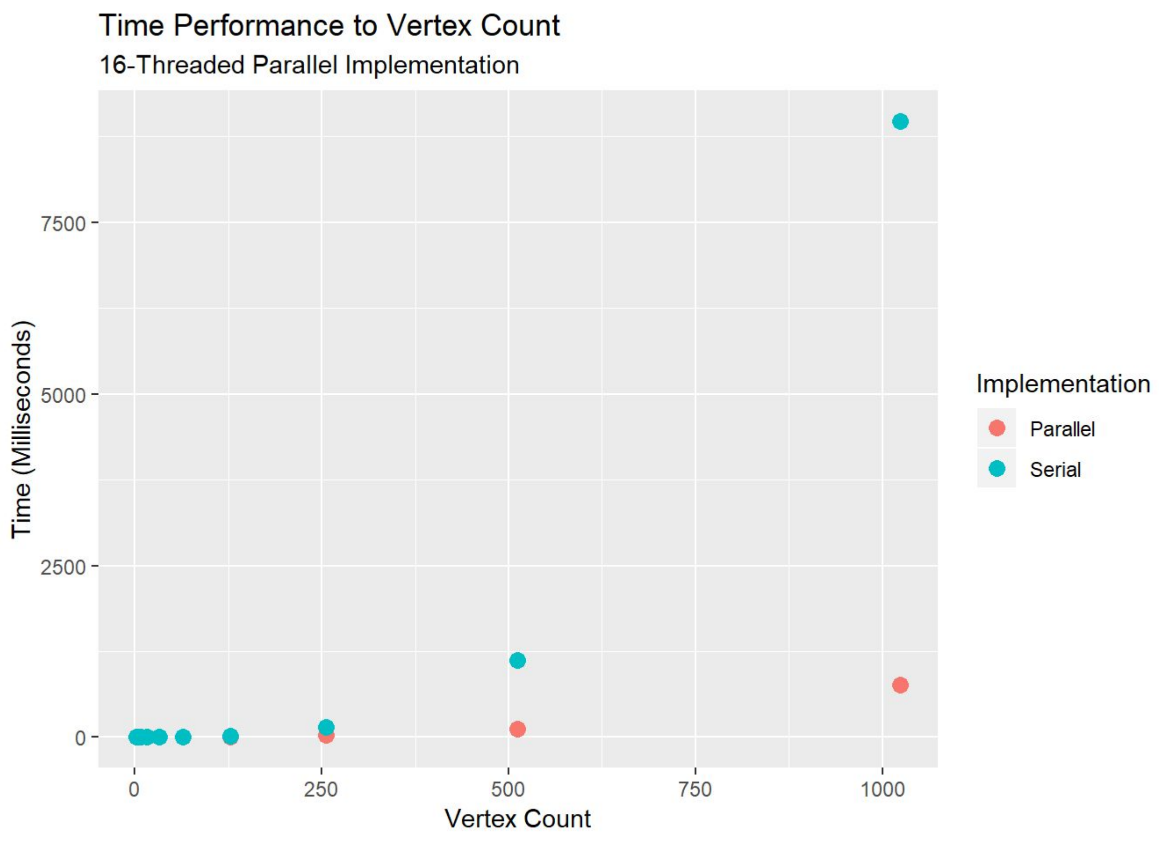


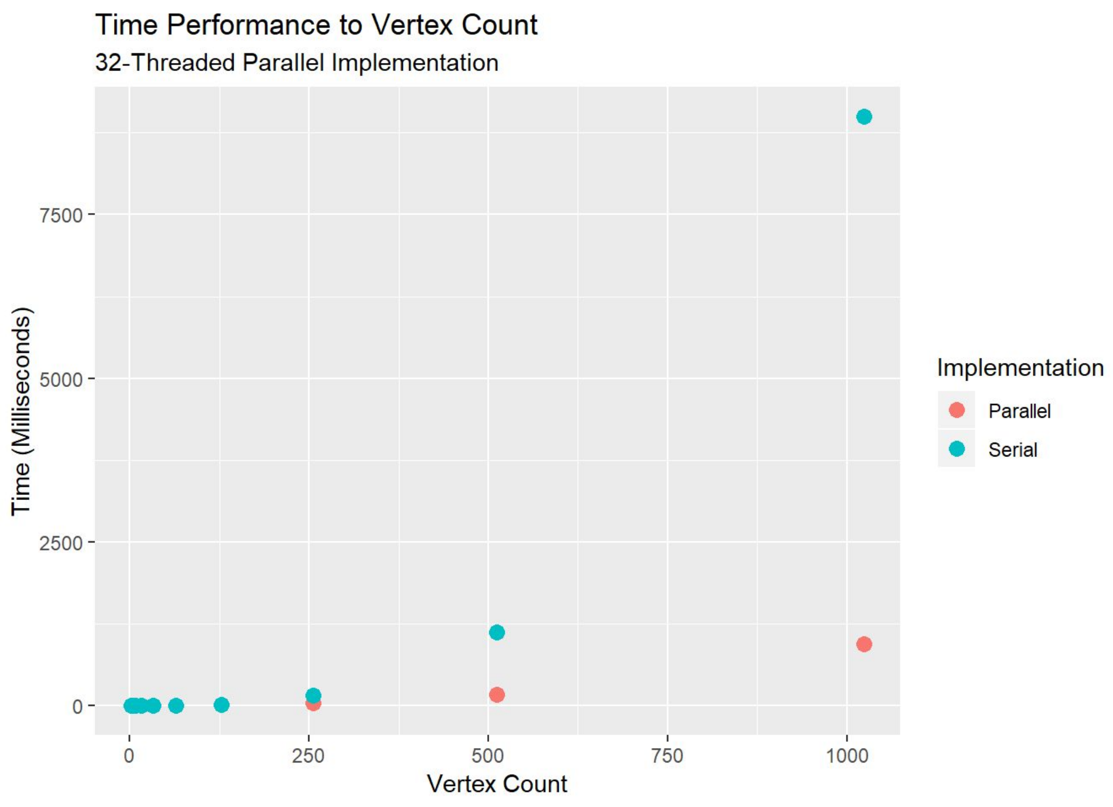
Time Performance to Vertex Count
2-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial

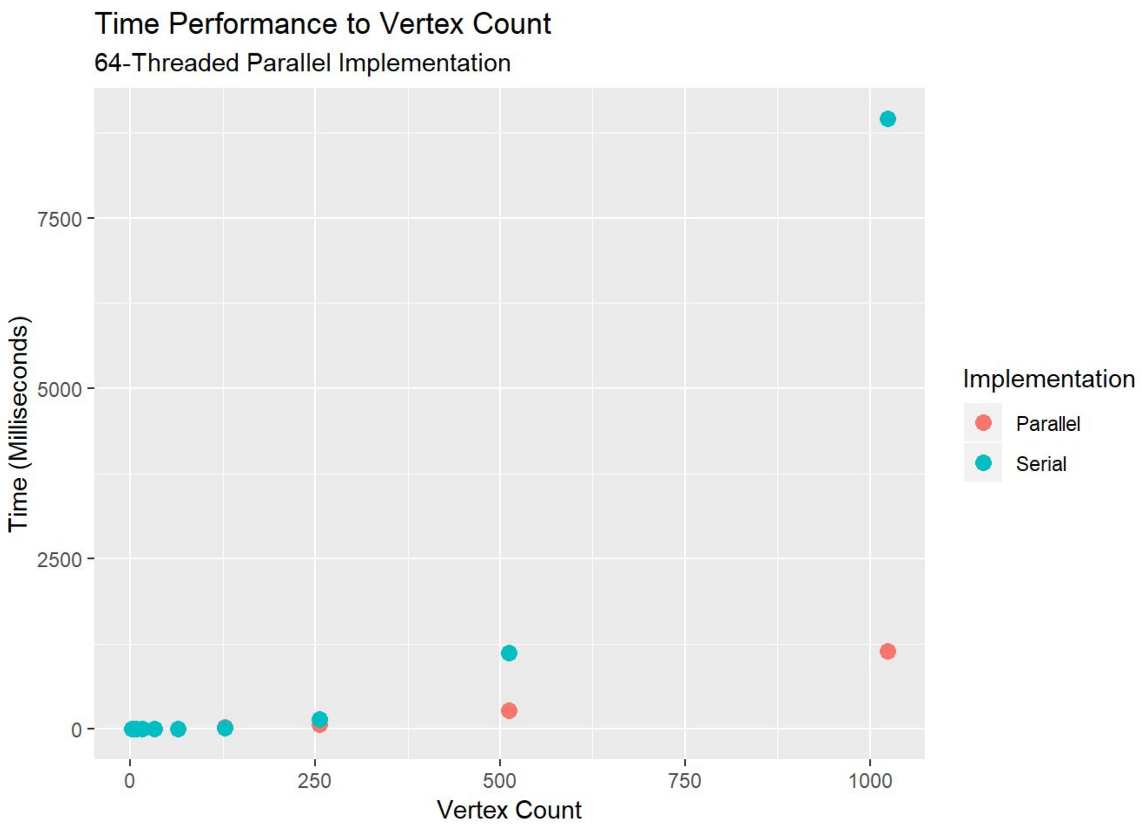


Time Performance to Vertex Count
4-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial


Time Performance to Vertex Count
8-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial

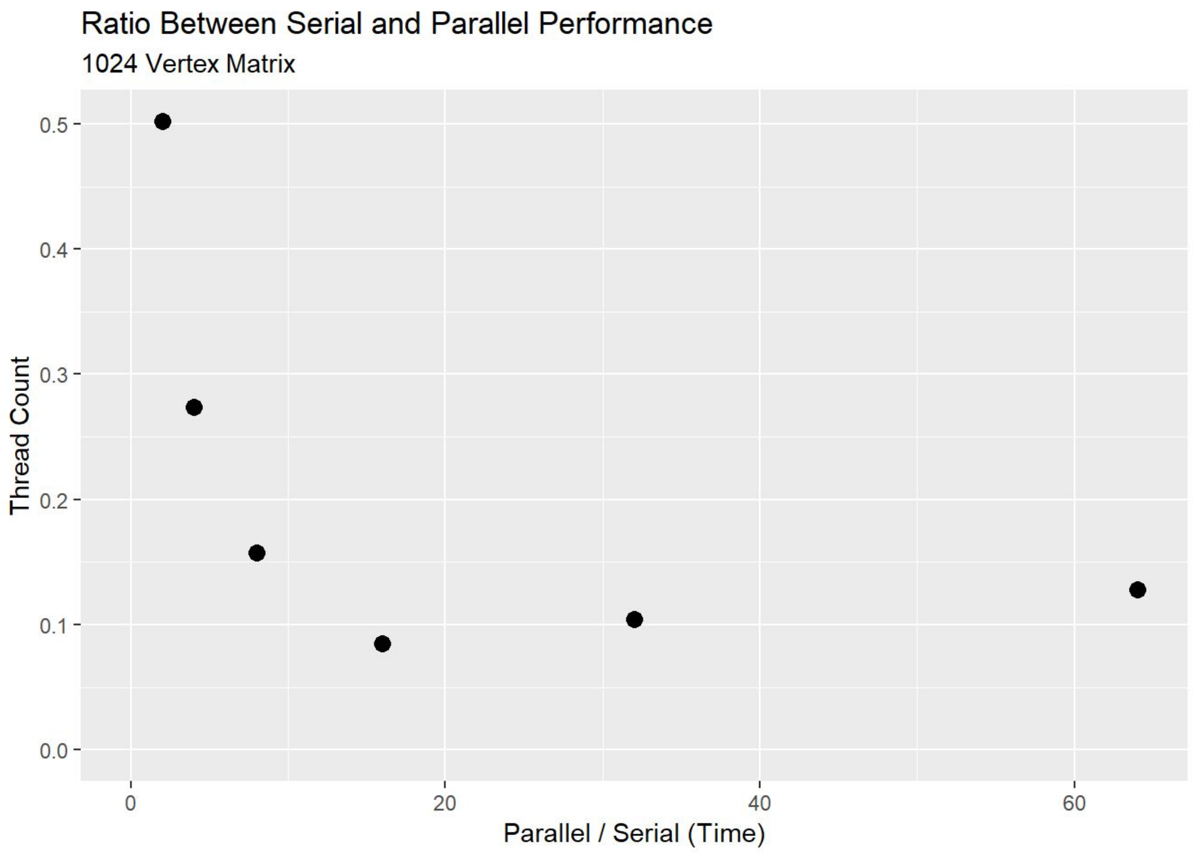

Time Performance to Vertex Count
16-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial


Time Performance to Vertex Count
32-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial


Time Performance to Vertex Count
64-Threaded Parallel Implementation
Time (Milliseconds)
7500
5000
2500
0
0
250
500
750
1000
Vertex Count
Implementation
Parallel
Serial

Analysis:

At first glance, the graphs illustrate a clear victory of parallel over serial for thread counts of 2 or greater. The gap grows wider from 2 to 4 to 8 to 16 only to stagnate or even rise a bit across 16, 32, and 64. This trend is most visible in the following graph for testing on 1024 x 1024 matrix:



No. of Threads	Parallel	Serial	Ratio
2	4499.728027	8966.560547	0.5018343437
4	2453.428955	8951.294922	0.2740864843
8	1404.870972	8913.519531	0.1576112519
16	764.099976	8970.419922	0.08517995619
32	941.504028	8992.023438	0.1047043565
64	1149.130981	8952.446289	0.1283594387

Originally, it was hypothesized that we would see a performance optimization of:

$$(\text{“Time of Serial Implementation” / Number of Threads}) + T_S$$

since in theory 100% of the work would be parallelized so we wouldn’t need to take Amdahl’s law into account. The “ T_S ” stood for synchronization time, where threads would have to wait up for other threads due to the nature of the semaphores used, plus the time it took to increment and decrement these semaphores. The graph above shows an interesting trend, where we start off with a ratio that matches the original prediction, but slowly lose that trend until we stop decreasing time at 16 threads. A naive explanation may be that at higher thread counts we are using more semaphores and have to make more calls to sem_wait and sem_post which could be taking a lot of time. However, the algorithm is designed such that we only use No. of vertices semaphores, so while more *threads* are making calls to sem functions, the same number of total calls should be made.

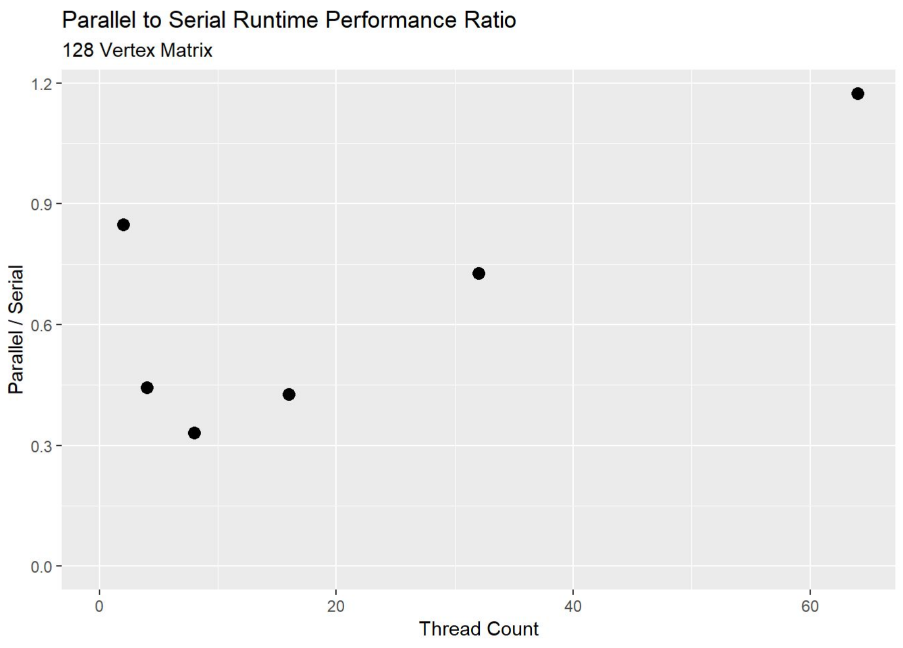
Instead, a possible explanation for this behavior is that threads are getting out of sync, and at different times in the runtime. It may be that many threads advance to a point where they need to access a certain row that has not been updated yet, and the thread they are waiting on is lagging behind. When the lagging thread finally catches up, another thread might start lagging behind and then the rest of the threads are waiting on that thread now. The more threads at play, the more chances for bottlenecks such as these to show up if not all threads are rolling at the same pace throughout runtime. This would help explain why we are actually increasing time compared to the serial algorithm at 32 threads and 64 threads. For this new conjecture, the runtime would better be captured by the following:

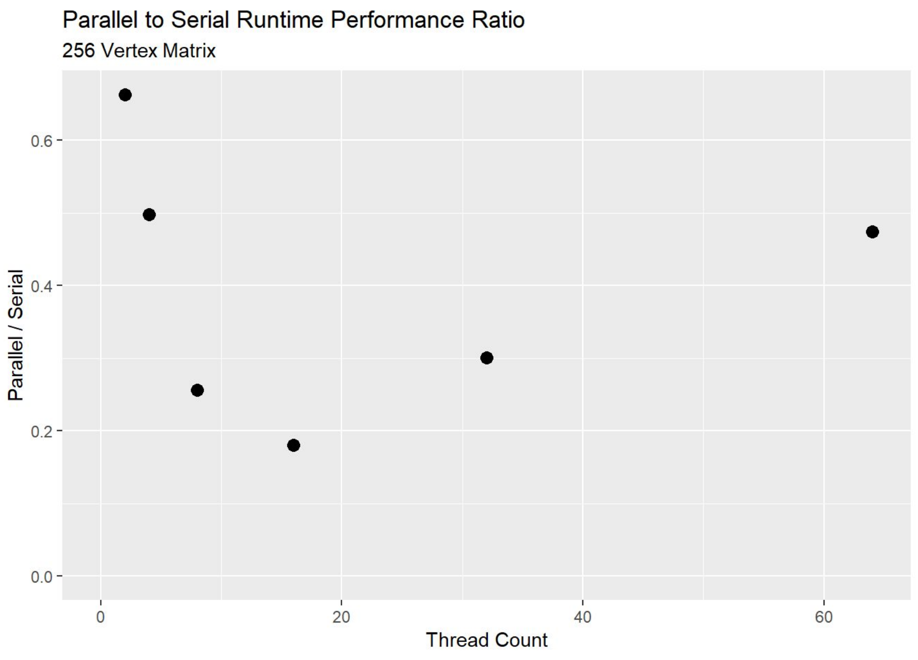
$$(\text{“Time of Serial Implementation” / Number of Threads}) + T_S(\text{Number of Threads})$$
$$=$$
$$O(n^3)\,/\,P \;+\; T_S(P,\,n)$$

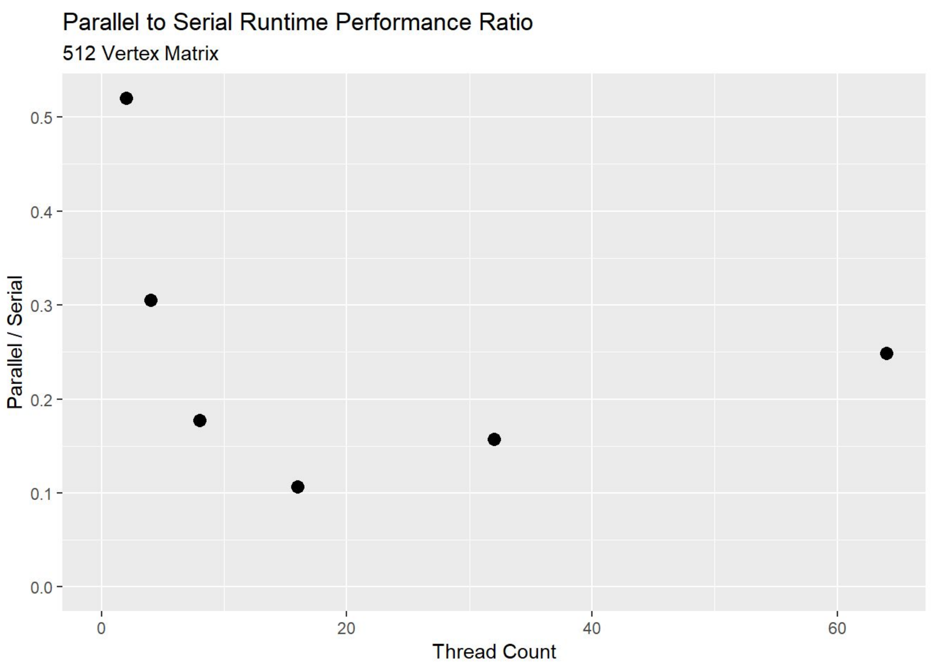
Where n = number of vertices, P = number of threads / CPU cores

Here T_S (P, n) would actually be a function on the number of threads and the number of vertices, since the time to synchronize many threads is greater than fewer threads. The input matrix would also help determine that synchronization time, since it determines how long the threads will be running in general and thus the possible number of times they may stall at. What that function is, is not exactly clear yet and would require more testing.

The 128, 256, and 512 vertex matrix tests show a similar curve but with the added effect of steeper climbs at the 16 thread mark correlating with lower vertex counts:


Parallel to Serial Runtime Performance Ratio
128 Vertex Matrix
1.2
0.9
0.6
0.3
0.0
Parallel / Serial
0
20
40
60
Thread Count

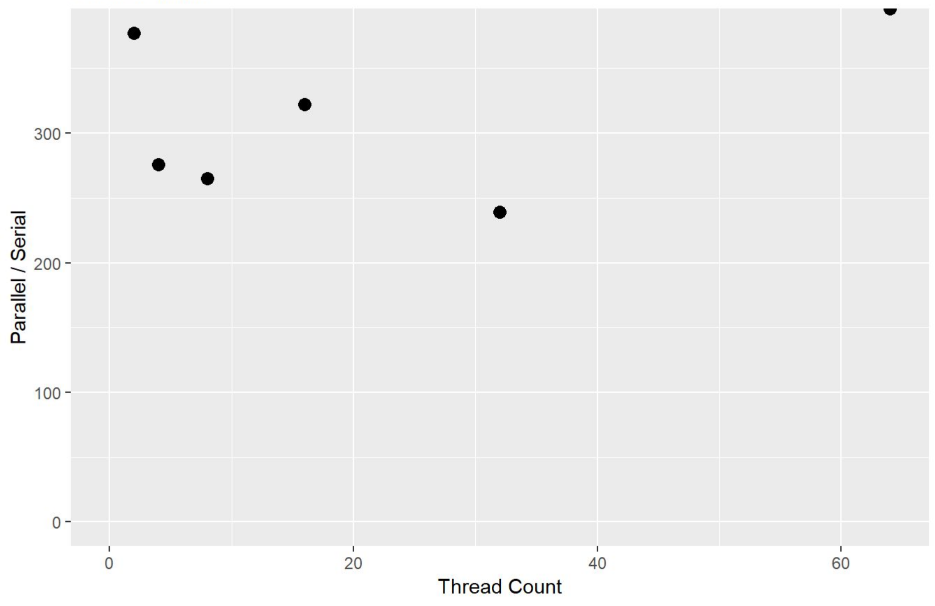

Parallel to Serial Runtime Performance Ratio
256 Vertex Matrix
0.6
0.4
0.2
0.0
Parallel / Serial
0
20
40
60
Thread Count



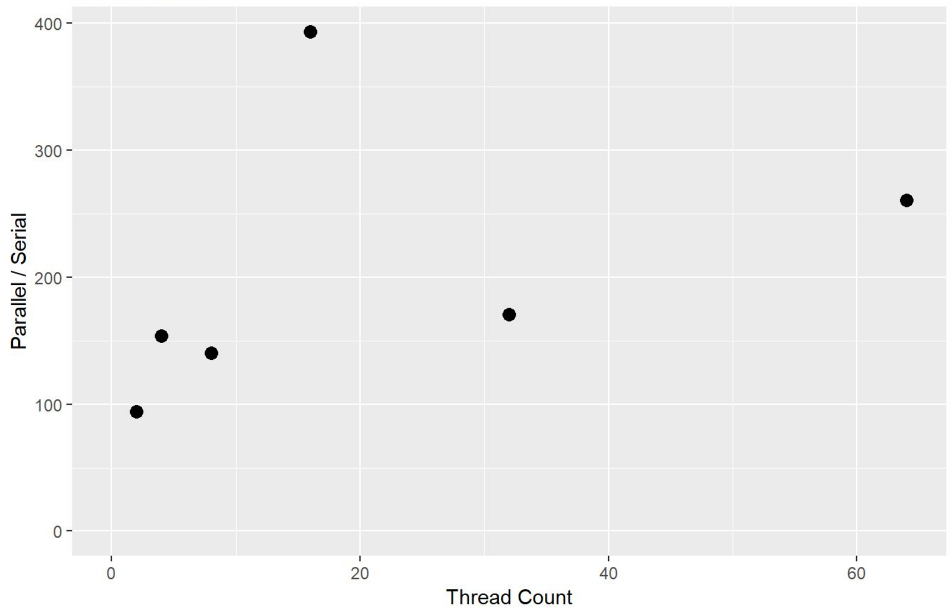
These graphs support the above equation, and more specifically that $T_S(P, n)$ may be more dominated by *P* than *n*, since the offset is greater for the smaller graphs. The serial running time is a lot smaller for the smaller input graphs, so that's why we adding in similar values for T_S appear more drastic in the above charts.

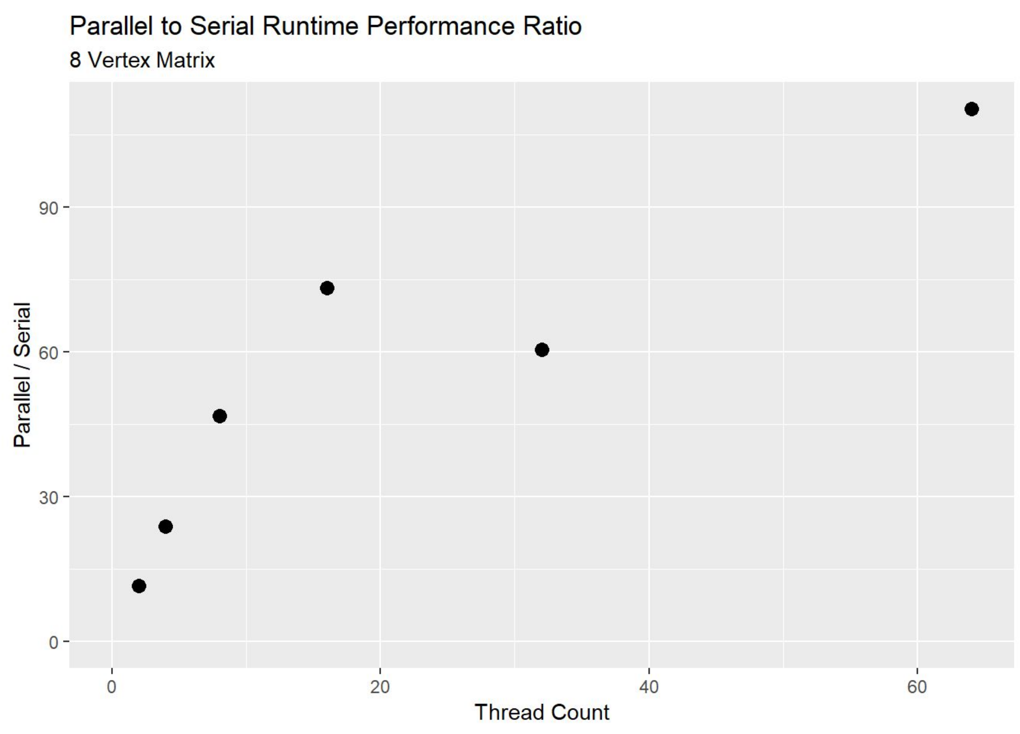
Lastly, the initial hypothesis didn't take into account the initial overheard involved, which includes initializing the semaphores as well as initializing separate threads. This initial overhead would explain why we see a performance <u>decrease</u> in the parallel implementation in comparison to the serial implementation for graphs smaller than 32 vertices (and more chaotic behavior in general):

Parallel to Serial Runtime Performance Ratio
2 Vertex Matrix

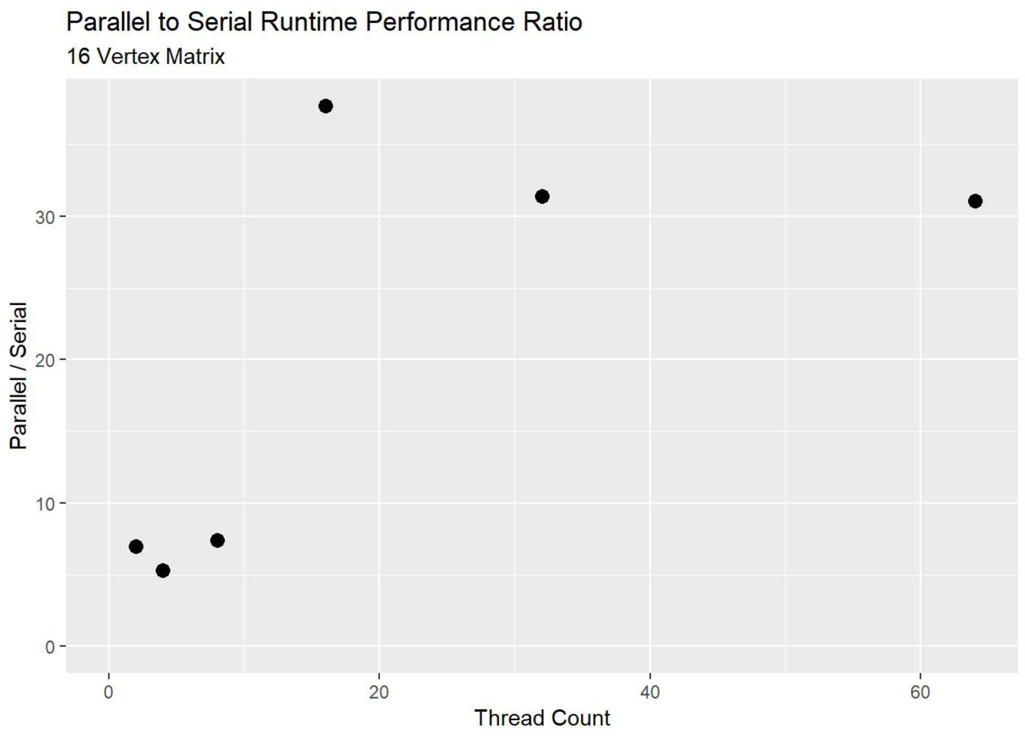


Parallel to Serial Runtime Performance Ratio
4 Vertex Matrix





Parallel to Serial Runtime Performance Ratio
8 Vertex Matrix
Parallel / Serial
0
30
60
90
0
20
40
60
Thread Count



Parallel to Serial Runtime Performance Ratio
16 Vertex Matrix
Parallel / Serial
0
10
20
30
0
20
40
60
Thread Count

While this initial overhead is not trivial, especially if we were particularly interested in working only with smaller graphs, we don’t want to factor it into our big O notation equation, since there we only care about the growth rate of the run time (what it tends toward).