Genome Searching in Parallel

Jake Masters

September 14, 2018

1 Implementation

When thinking about how to implement genome searching in parallel, I first developed the following algorithmic strategy:

```
main()
  partition()*
  for thread in threads:
    pthread_create(match)
      match()
        // compute number of local matches*
        // lock mutex
        global_matches += local_matches
        // unlock mutex
  for thread in threads:
    // join threads
  // done!
```

The two steps with the asterisks next to them were the most difficult parts of my implementation. I thought about solving the local match computing through interleaving or load-balancing, but I ended up choosing what I believe to be the simplest route of solving this problem in parallel: dividing the entire problem into equivalent, continuous chunks.

For my specific solution starts off with partitioning the data into chunks. I decided to give each processor a single chunk as determined using the following equation:

$$chunk = \frac{N_n}{N_p}$$

Where N_n is the number of nucleotides being searched and N_p being the number of processors being employed in the current computation.

To implement the idea of giving each thread a single chunk, I gave each thread a starting pointer, which I will call p. For the first thread, I gave it `fasta->sequence.`Every thread after that was given a pointer that start *chunk* bytes after the preceding thread's pointer.

To implement searching each chunk, I simply imitated the sequential search found in `sg.c.`The only logical changes I made had to do with computing the boundaries and ensuring that the entire search space is covered exactly once across all processors.

2 Experiments

For my experimental parameters, I decided to vary my nucleotide counts and number of threads while holding my pattern string constant. All of my experiments were conducted with the pattern string `GATTACA.`To determine which time to record for each experiment, I ran each experiment three times and took the median time as my recorded result. I included the S and E results for the first table (single chromosome) and the last table (full genome).

2.39e+08 Nucleotides	1 thread	2 threads	4 threads	8 threads	16 threads
Computation Time	1.616 s	0.832 s	0.449 s	0.386 s	0.360 s
S	1	1.942	3.599	4.187	4.489
E	1	0.971	0.899	0.523	0.281

4.84e+08 Nucleotides	1 thread	2 threads	4 threads	8 threads	16 threads
Computation Time	1.626 s	0.825 s	0.441 s	0.429 s	0.435 s
S	1	1.971	3.687	3.790	3.738
E	1	0.986	0.899	0.474	0.234

Figure 1: Speedup graph for 2.39e+08 Nucleotides

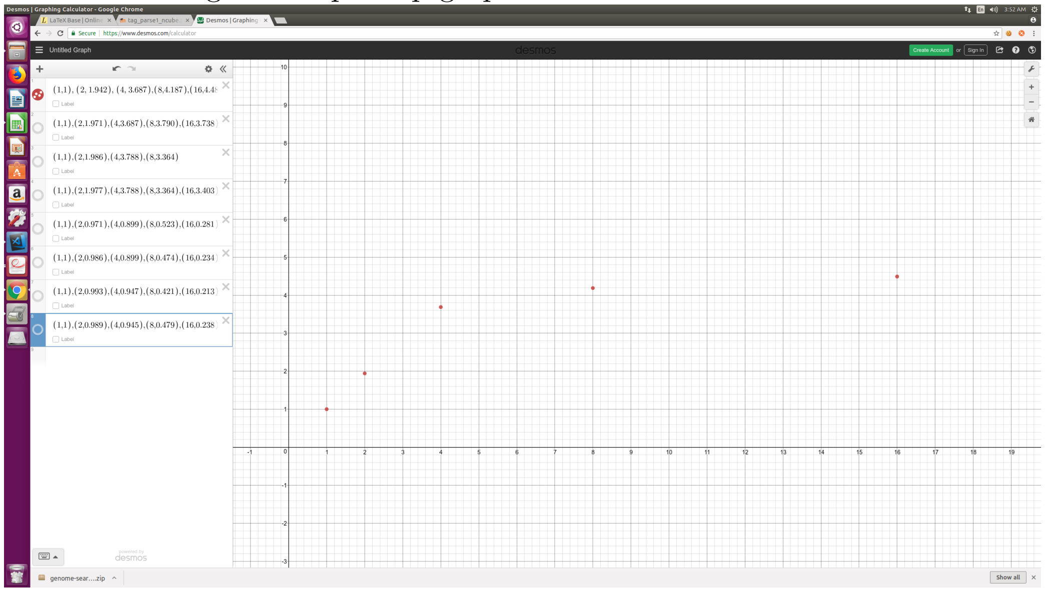

```
(1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
(1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738)
(1,1),(2,1.986),(4,3.788),(8,3.364)
(1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.403)
(1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281)
(1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234)
(1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213)
(1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238)
```

Figure 2: Efficiency graph for 2.39e+08 Nucleotides

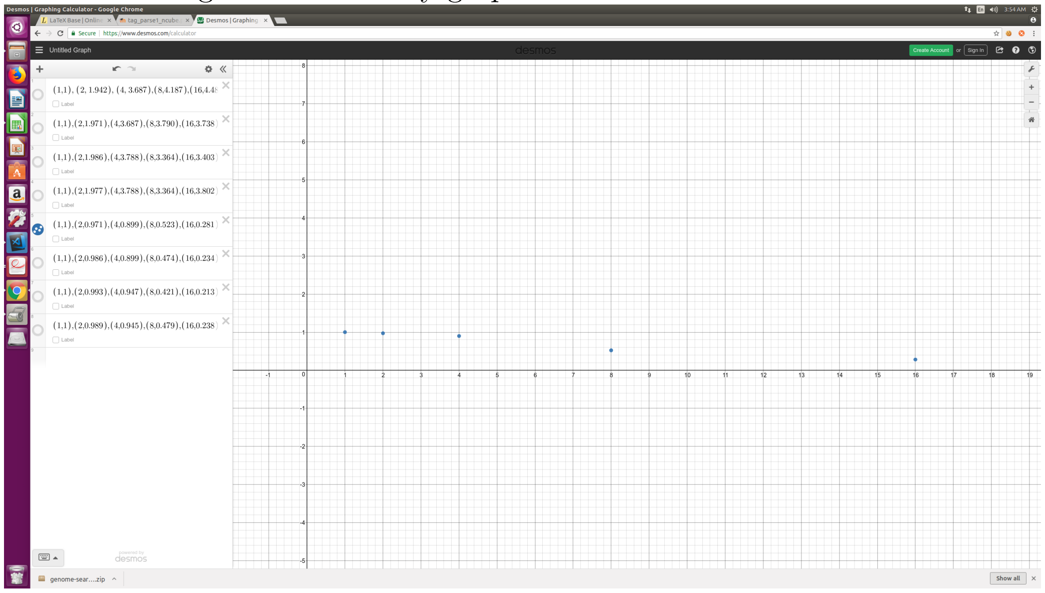
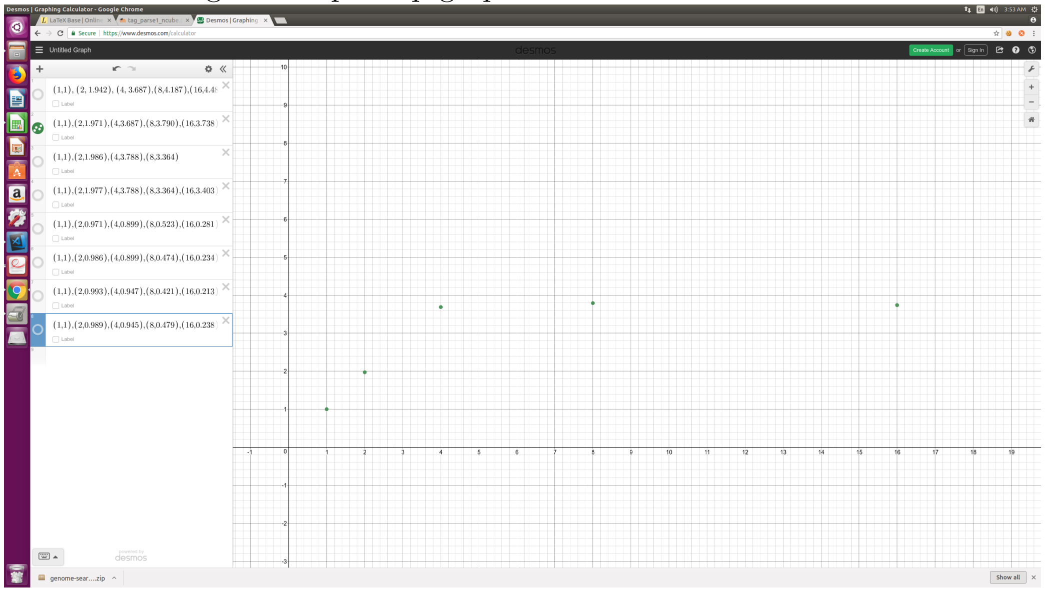
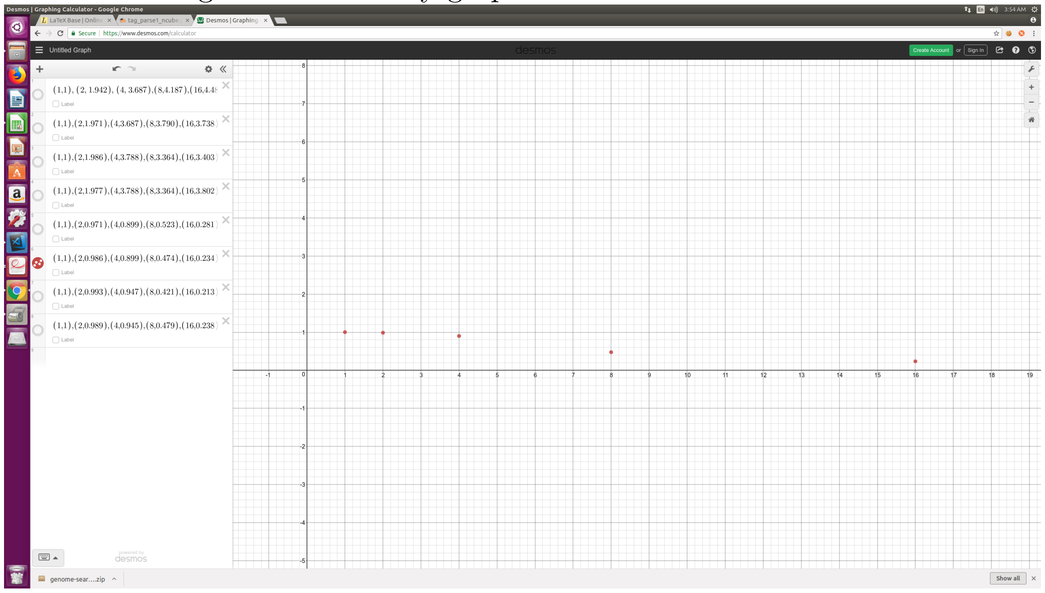

```
(1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
(1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738)
(1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403)
(1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802)
(1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281)
(1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234)
(1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213)
(1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238)
```

Figure 3: Speedup graph for 4.84e+08 Nucleotides



```
1 (1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
2 (1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738)
3 (1,1),(2,1.986),(4,3.788),(8,3.364)
4 (1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.403)
5 (1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281)
6 (1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234)
7 (1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213)
8 (1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238)
```

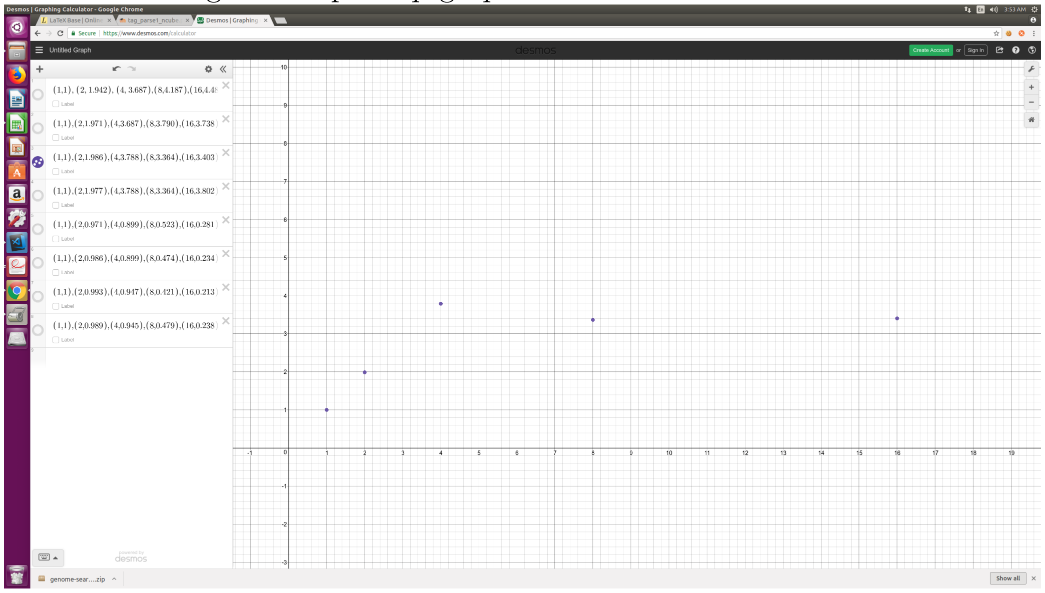
Figure 4: Efficiency graph for 4.84e+08 Nucleotides



```
1 (1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
2 (1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738)
3 (1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403)
4 (1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802)
5 (1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281)
6 (1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234)
7 (1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213)
8 (1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238)
```

2.19e+09 Nucleotides	1 thread	2 threads	4 threads	8 threads	16 threads
Computation Time	9.793 s	4.932 s	2.585 s	2.911 s	2.878 s
S	1	1.986	3.788	3.364	3.403
E	1	0.993	0.947	0.421	0.213

Figure 5: Speedup graph for 2.19e+09 Nucleotides

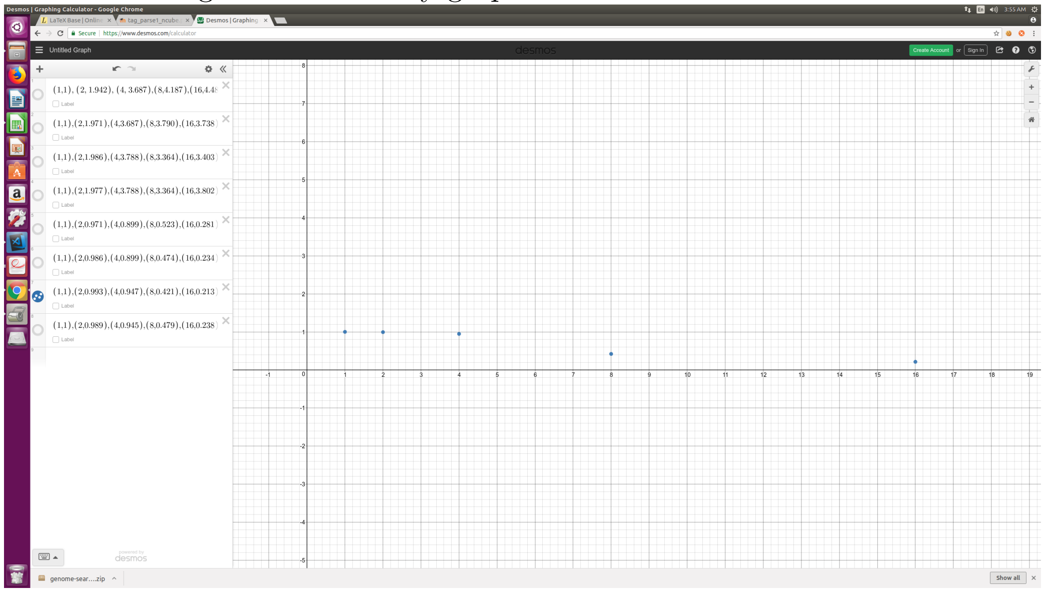


1. (1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
2. (1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738)
3. (1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403)
4. (1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802)
5. (1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281)
6. (1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234)
7. (1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213)
8. (1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238)

Desmos | Graphing Calculator - Google Chrome
https://www.desmos.com/calculator
Untitled Graph
Create Account
Sign In
genome-sear....zip
Show all

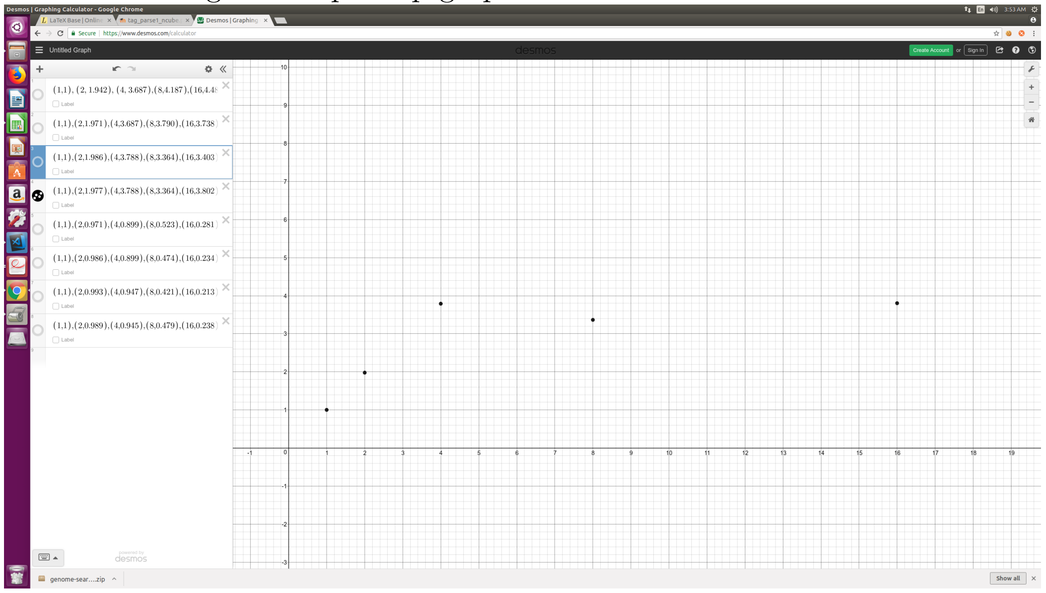
3.11e+09 Nucleotides	1 thread	2 threads	4 threads	8 threads	16 threads
Computation Time	10.221 s	5.170 s	2.704 s	2.663 s	2.688 s
S	1	1.977	3.779	3.838	3.802
E	1	0.989	0.945	0.479	0.238

Figure 6: Efficiency graph for 2.19e+09 Nucleotides



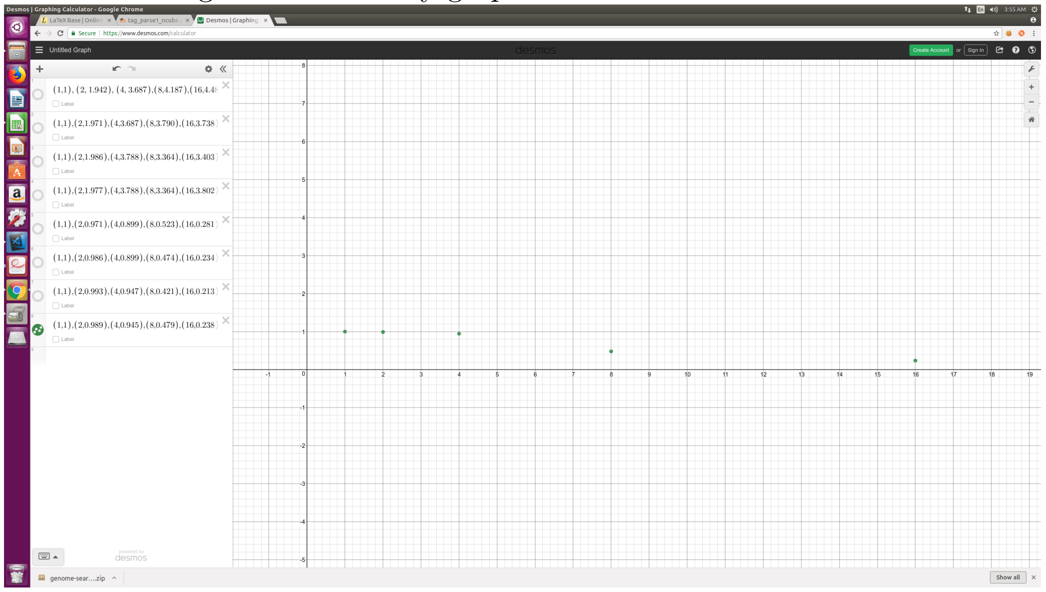
```
(1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
(1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738
(1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403
(1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802
(1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281
(1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234
(1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213
(1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238
```

Figure 7: Speedup graph for 3.11e+09 Nucleotides



```
(1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
(1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738
(1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403
(1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802
(1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281
(1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234
(1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213
(1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238
```

Figure 8: Efficiency graph for 3.11e+09 Nucleotides



```
(1,1), (2, 1.942), (4, 3.687),(8,4.187),(16,4.4
(1,1),(2,1.971),(4,3.687),(8,3.790),(16,3.738
(1,1),(2,1.986),(4,3.788),(8,3.364),(16,3.403
(1,1),(2,1.977),(4,3.788),(8,3.364),(16,3.802
(1,1),(2,0.971),(4,0.899),(8,0.523),(16,0.281
(1,1),(2,0.986),(4,0.899),(8,0.474),(16,0.234
(1,1),(2,0.993),(4,0.947),(8,0.421),(16,0.213
(1,1),(2,0.989),(4,0.945),(8,0.479),(16,0.238
```

3 Analysis

My implementation of genome search in parallel performed very well through 4 threads of execution, which is equivalent to the number of physical cores on the machine I tested my implementation on. However, speedup and efficiency drop off drastically starting with 8 threads. This makes sense because due to the nature of the problem and my implementation, all physical cores should be in use for the entire duration of the program. Since there are only 4 physical cores, then it naturally follows that for the majority of the program's runtime, any thread outside of the 4 threads currently executing on the physical cores would be blocked until one of the currently executing threads either finishes or is waiting on another system call to finish.