CSE411 hw2 analysis
Joss Duff
jod323@lehigh.edu

Overview

I first attempt to parallelize a prime checking algorithm using different techniques, then I parallelize k-means using the best technique. In the prime section, I am just exploring different parallelization techniques, not optimizing speedup. In the kmeans section, I use parallelization and other techniques to optimize speedup.

Prime

The prime algorithm takes an input N and returns the number of primes in 0..N. It is O(n^2) because it loops over each number twice: for each n in N, it checks all numbers 2..n for factors. The function `isPrime(n)` returns true if the number is prime. This function is easily optimized with techniques like checking only up to n/2, or skipping even numbers, but we leave optimization exercises to the K-means section and only investigate parallelism in the prime section. Parallelization isn't trivial because the load is unbalanced. `isPrime(100)` is 100x faster than `isPrime(10,000)`. I attempt to parallelize using TBB `parallel_for`, manual `std::thread` static load balancing, and manual `std::thread` dynamic load balancing. All implementations use the same exact `isPrime()` check.

Hypothesis

The entire workload is known before execution time so in a vacuum, static load balancing approach would perform the best. However, this experiment is run on Sublab, a public resource, so at any time another student could execute programs on the machine and slow down execution for some threads. Additionally, some cores might execute slower than others due to operating system execution, heating issues, or other small background tasks. Because of these factors, a dynamic load balancing approach like TBB parallel_for will perform the best.

Methodology

All tests were run on Sunlab. Tests that don't mention threadcount were run with 16 threads. Most tests were run on different machines, as sshing into sunlab assigns a random machine. For this reason, conclusions shouldn't be drawn across different tests. Occasionally I switched machines because other users were running CPU intensive tasks. For longer running tests, other users might have used resources and affected the results.

Values of N lower than 10k produced drastically varying results. The execution time was extremely low and not an accurate measure of a parallelization technique. Across all implementations, I test 10k (10^4), 100k (10^5), and 1M (10^6). In some cases I also had enough time to test 10M (10^7). The prime checking algorithm is O(n^2), so execution time expectantly grows quickly.

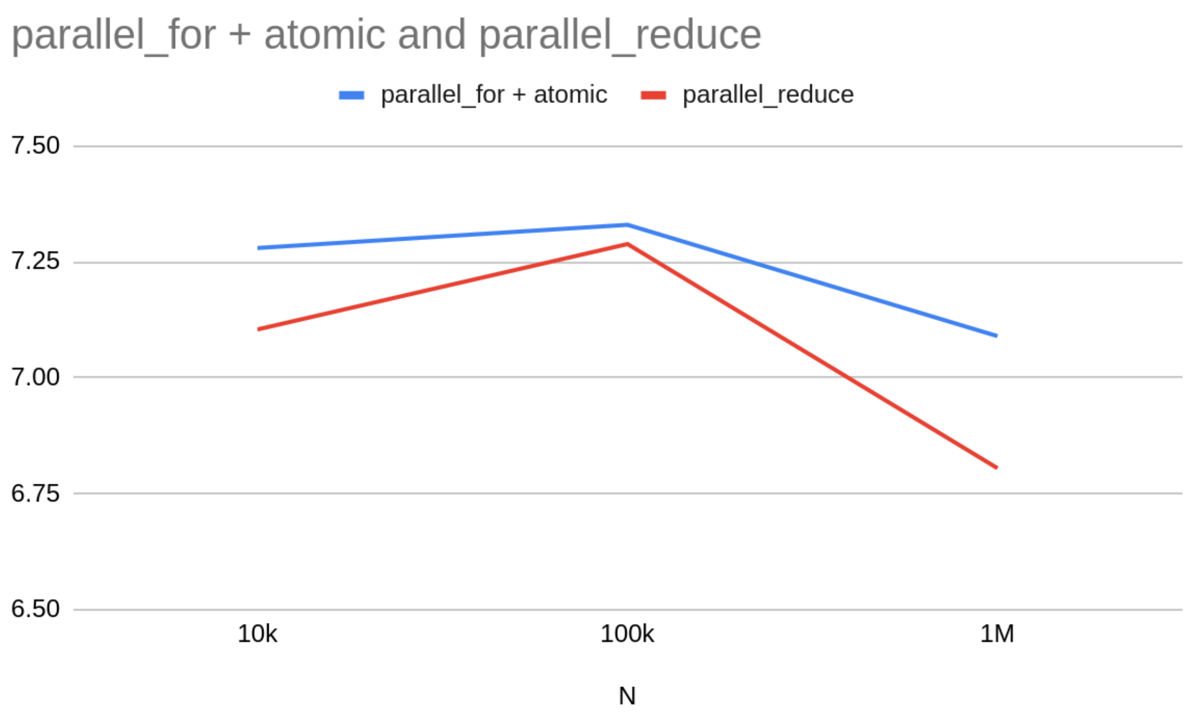
Execution of each algorithm was run 6 times. 1 warmup run and 5 timed benchmark runs. When the benchmark was complete, it reported the average, median, min, and max values. Speedup is <average time serial> / <average time this impl>. Graphs that simply note execution time use the average execution time.

In every execution except sequential, an atomic counter is used to return the final count of prime numbers. This is necessary to test for correctness after each execution. The serial implementation is treated as the "correct" prime count, and other implementations are compared against it. None of these implementations return an incorrect prime count.

TBB Implementation

As the goal of this assignment is to gain appreciation for TBB, I only explored using `parallel_for` and `parallel_reduce`. I explore more advanced techniques in part 2. The `parallel_for` approach uses an atomic<int> to count the number of primes across threads. `parallel_reduce` doesn't need an atomic variable as the writing is handled by TBB.

The below graph shows that `parallel_for` + an atomic performs better than `parallel_reduce` in all tested workloads. For benchmarking in other tests, I utilized the `parallel_for` implementation. This could be due to the overhead in TBB cross thread communication being greater than simply incrementing a shared counter.



Static Load Balancing Implementation

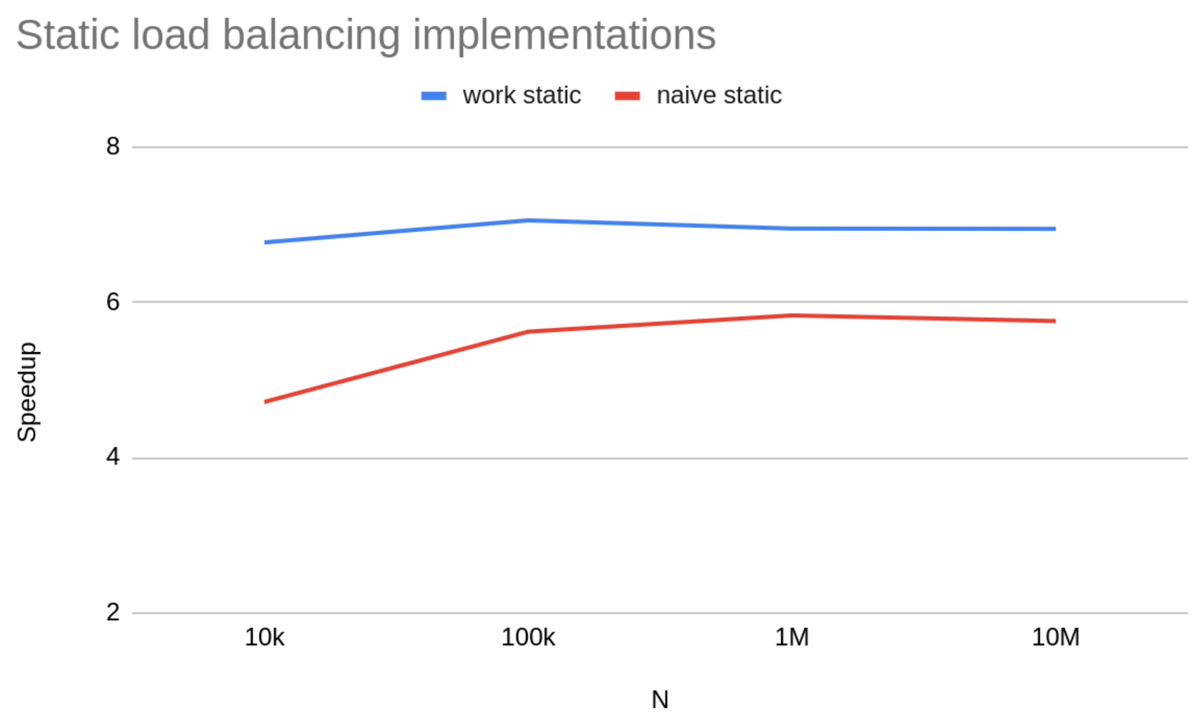
The naive approach of assigning each thread 16/N values performs terribly as the workload is heavily unbalanced. The `isPrime` function tests all factors 2-N-1, so larger numbers take much longer to execute. There will always be a thread that is largely bottlenecked in this approach. My solution was to sum up the total iterations in `isPrime` required for N and then assign numbers [0..N] to threads to evenly distribute `isPrime` iterations. Each thread should get <total_work>/<num_threads> work. I allocated numbers to threads by iterating backwards over values in N and greedily assigning values, removing them from a list of numbers to assign. If a number would put a thread over it's allocation, it was skipped and the next number was checked. The last thread received all remaining numbers. This could result in the last thread having an unbalanced workload, but since we were iterating backwards it was minimal. In practice this algorithm is very fair. Here is an example of the work distribution:

```
N=10M
Thread 0 work:  3124999062500
Thread 1 work:  3124999062500
Thread 2 work:  3124999062500
Thread 3 work:  3124999062500
Thread 4 work:  3124999062500
Thread 5 work:  3124999062500
Thread 6 work:  3124999062500
Thread 7 work:  3124999062500
Thread 8 work:  3124999062500
Thread 9 work:  3124999062500
Thread 10 work: 3124999062500
Thread 11 work: 3124999062500
Thread 12 work: 3124999062500
Thread 13 work: 3124999062500
Thread 14 work: 3124999062500
Thread 15 work: 3124999062501 (has 1 more unit of work)
```

The issue with this static approach is that it only counts iterations of `isPrime` as work, and doesn't account for the overhead of iterating over the vector of assigned numbers. Each thread will likely have a different amount of numbers to check, with the last thread likely having the most because it is assigned many low numbers, which have low `work` values. I tried to optimize for this by adding a constant to each numbers `work` value. It isn't clear if this constant accurately reflects the overhead of adding another number to a thread's array.

```
N=10M
Thread 0 number count:  317542
Thread 1 number count:  328316
Thread 2 number count:  340266
Thread 3 number count:  353625
Thread 4 number count:  368693
Thread 5 number count:  385869
Thread 6 number count:  405695
Thread 7 number count:  428933
Thread 8 number count:  456690
Thread 9 number count:  490655
Thread 10 number count: 533555
Thread 11 number count: 590171
Thread 12 number count: 669874
Thread 13 number count: 794594
Thread 14 number count: 1035535
Thread 15 number count: 2499987 (2x as many numbers as next highest,
maybe it's okay)
```

As expected, the static load balancing based on work iterations performs better than the naive static load balancing that assigns a fixed amount of work to threads:



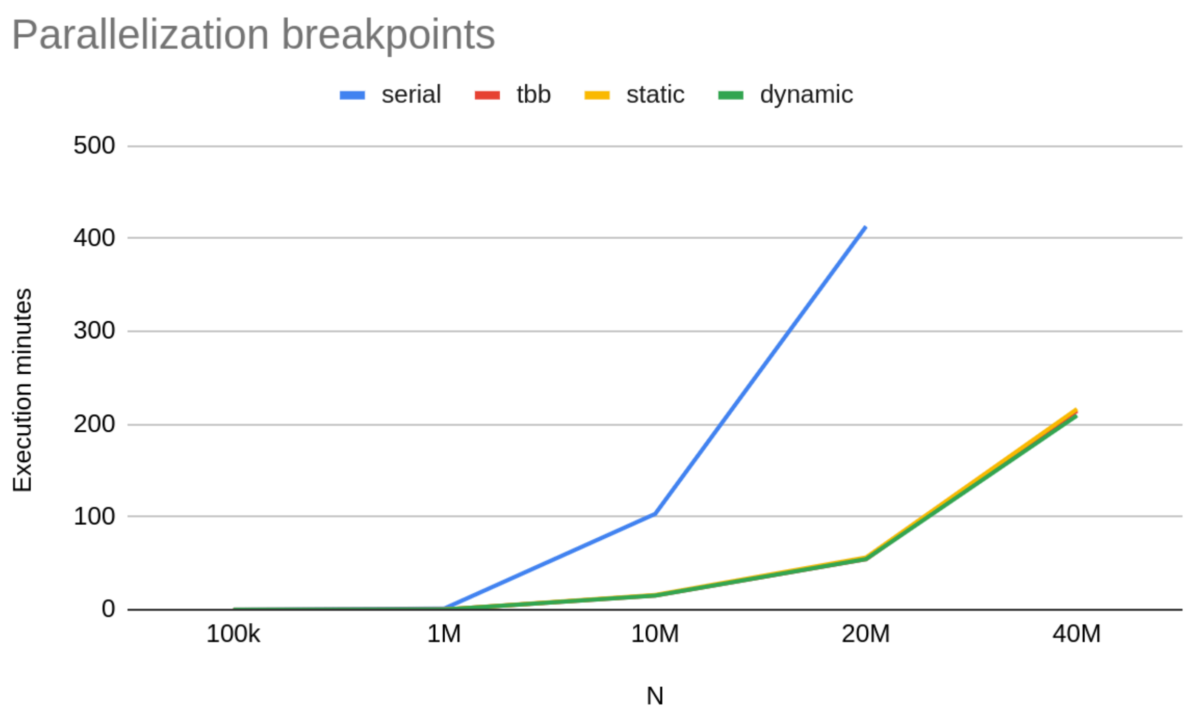
Dynamic Load Balancing Implementation

All threads share an atomic int which is the next number to check for primes. When a thread is done checking `isPrime` for it's current number, it does an atomic `fetch_add(1)` to get the next number it should check `isPrime` for. This approach works very nicely and natively balances workload across threads. It is also resistant to some threads being slower than others. The best performance was attained by iterating through N backwards, because starting with the largest `isPrime` checks will allow for better `packing` of each thread.

Breakpoints

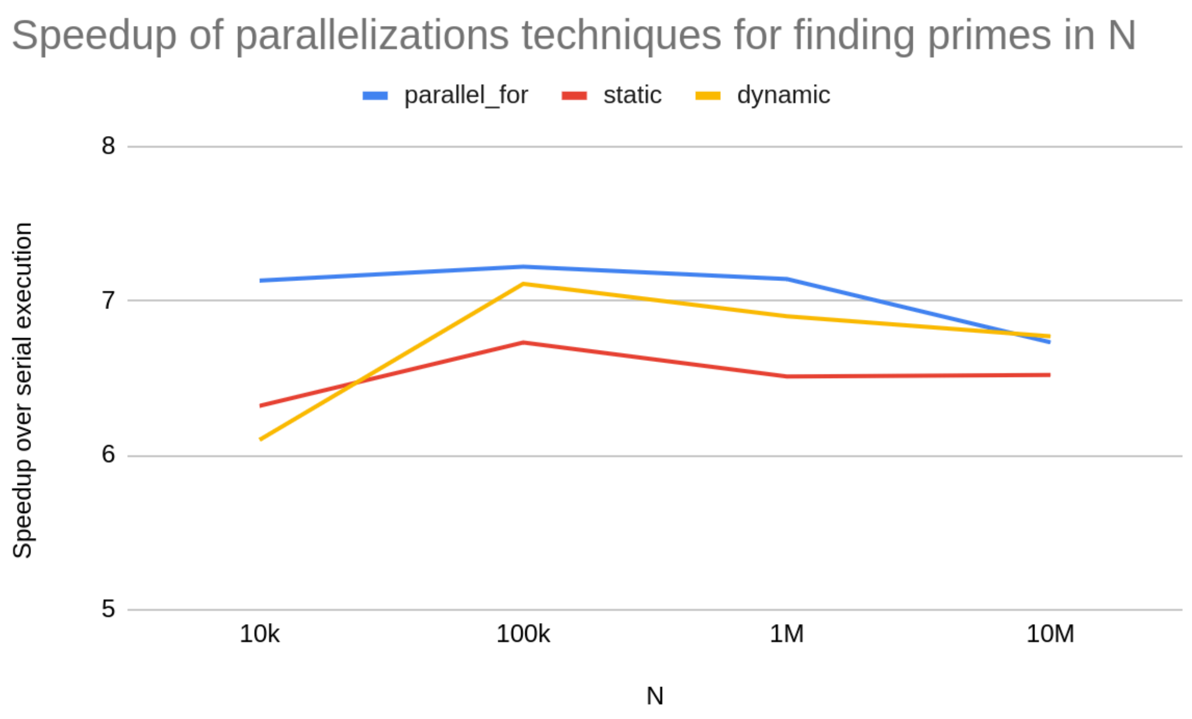
The static approach is naturally expected to have the earlier breakpoint than the parallelized breakpoints. Unfortunately, my time on earth is limited and so is this assignment duration so I didn't have enough time to explore breakpoints for all parallelization techniques. 300 minutes is 5 hours. I managed to finish some tests overnight, but I didn't have enough nights to terrorize Sunlab and collect data for all tests I wanted to run for this assignment.

We see the parallelized implementations have a much later breakpoint than the serial implementation. The static load balancing approach has a slightly higher execution time. If more tests were run, there might be a degradation in performance due to memory limits in the static and dynamic implementations, as they both involve turning N into a vector of size N, intuitively.



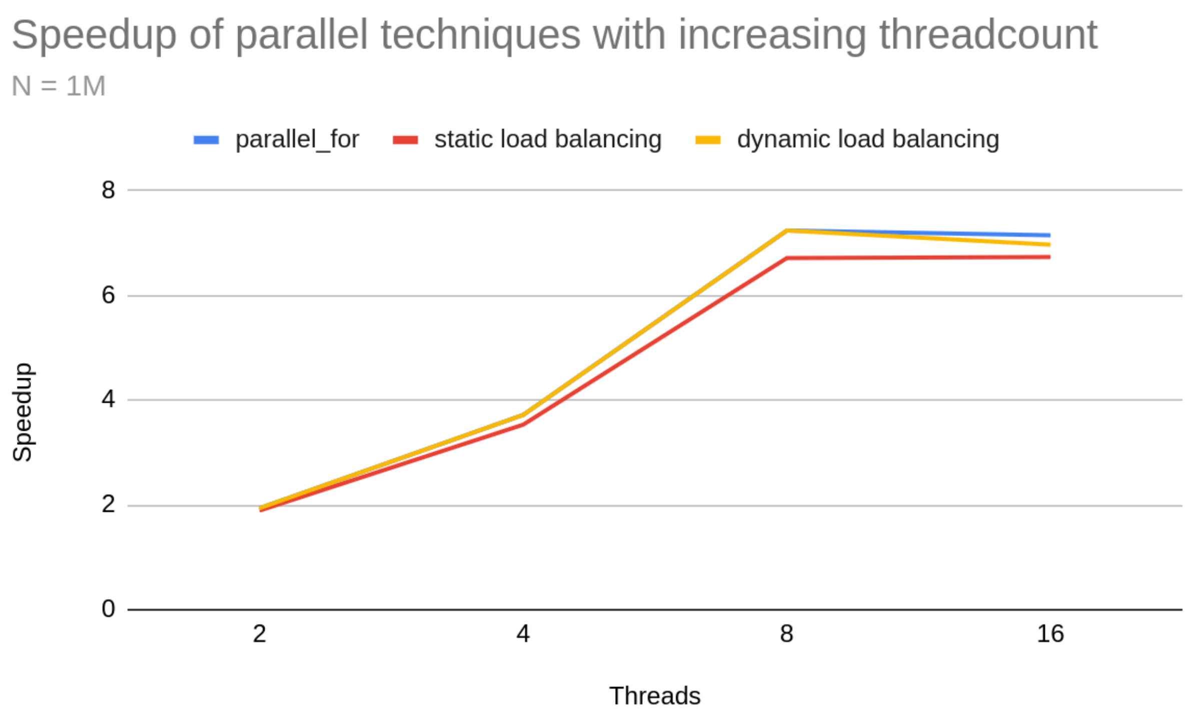
Overall speedup

TBB `parallel_for` performed the best across different workloads of N. At 10M however, the dynamic implementation performed better with 6.77x speedup vs TBBs 6.73x speedup. The dynamic approach was elegant. It involved iterating through all numbers in N backwards with a shared atomic index counter. When a thread was done calculating factors for its current number, it checked the index for the next number to calculate and incremented the index counter by 1. This is a very low overhead approach and tailored to this problem set. The TBB library has a larger overhead, and it is extremely impressive that a simple TBB `parallel_for` performed so well in all cases.



Threadcounts

Speedup was similar for all threadcounts until 8 threads were reached. At 8 threads, we see a divergence between the static load balancing approach vs TBB and dynamic load balancing. This is due to factors discussed in the `Static Load Balancing` section. As this algorithm has no dependencies, we expect and see speedup to be close to the number of threads being used. At 16 threads there is no meaningful increase in performance. This could be due to Sunlab having 8 physical cores, and 16 with hyperthreading.



K-means

K-means clustering is a clustering algorithm. It takes an input of data entries, each with multiple dimensions, and iterates over them to attempt to logically cluster them based on nearby entries.

Explanation

1. First, k-means randomly generates locations for K clusters.
2. Calculate the closest cluster to each point using the center of each cluster.
3. Calculate the new center of each cluster based on its new point membership.
4. Restart from step 2 until cluster membership doesn't change.

The problem is NP-hard, and most algorithms only find a local optimum clustering. The running time of these algorithms is O(N K D I) where

N = number of data entries
D = number of dimensions (I refer to them as "features" in this writeup)
K = number of clusters
I = number of iterations (step 4)

There is a hard dependency between step 2 and 3: the centers of each cluster are based on their point membership. There is also a hard dependency between iterations, as each iteration checks if points have changed since the previous iteration. The greatest opportunity in parallelism comes from parallelizing operations in step 2 and 3. I refer to step 2 as `calculate clusters` and step 3 as `recalculate center`.

Dataset

I found a large dataset of household power consumption. The full dataset has 2.5 million entries and 7 features. Unfortunately, the full dataset is too large to fit in my Sunlab partition so I had to reduce it to 250K entries.

Methodology

The same random seed was used across all benchmarks. If threadcount isn't mentioned, 16 threads were used. If feature count isn't mentioned, 7 features were used. Almost all runs completed after the first clustering iteration, indicating that there are strong natural clusters in this data.

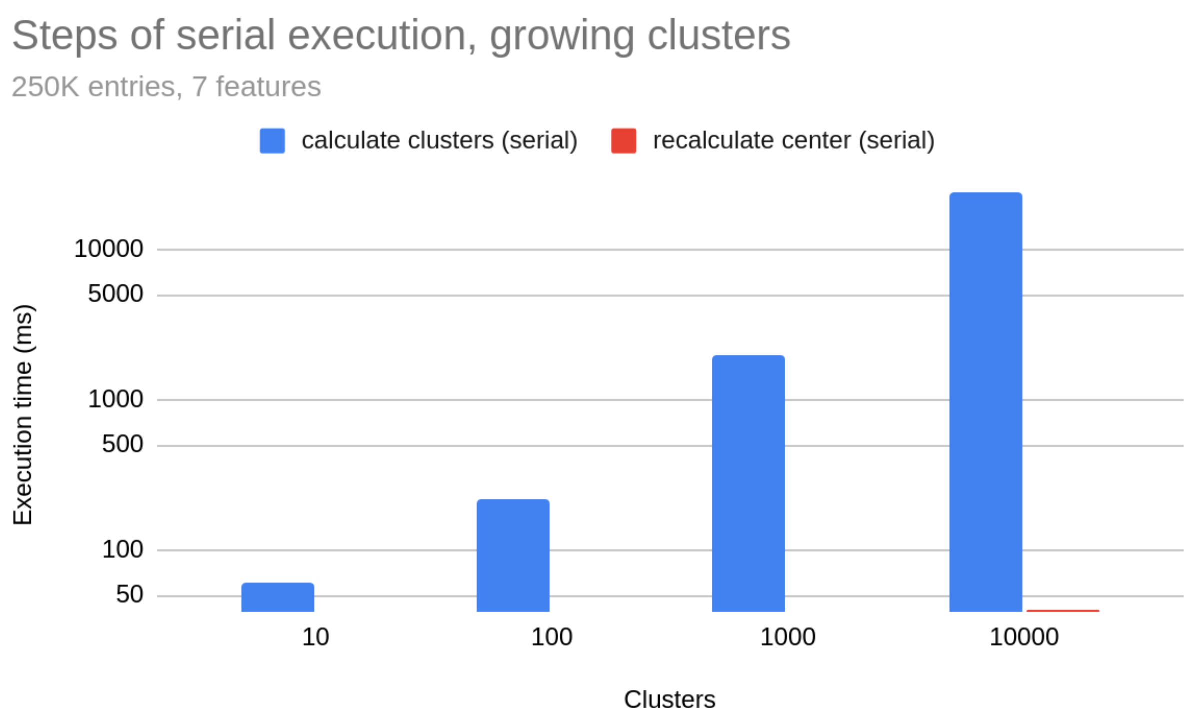
All tests were run on Sunlab. Most tests were run on different machines, as sshing into sunlab assigns a random machine. For this reason, conclusions shouldn't be drawn across different tests. Occasionally I switched machines because other users were running CPU intensive tasks. For longer running tests, other users might have used resources and affected the results.

Tests were done with a separate testing binary that compares clusters and makes sure the parallel version has the same clusters and the sequential version.

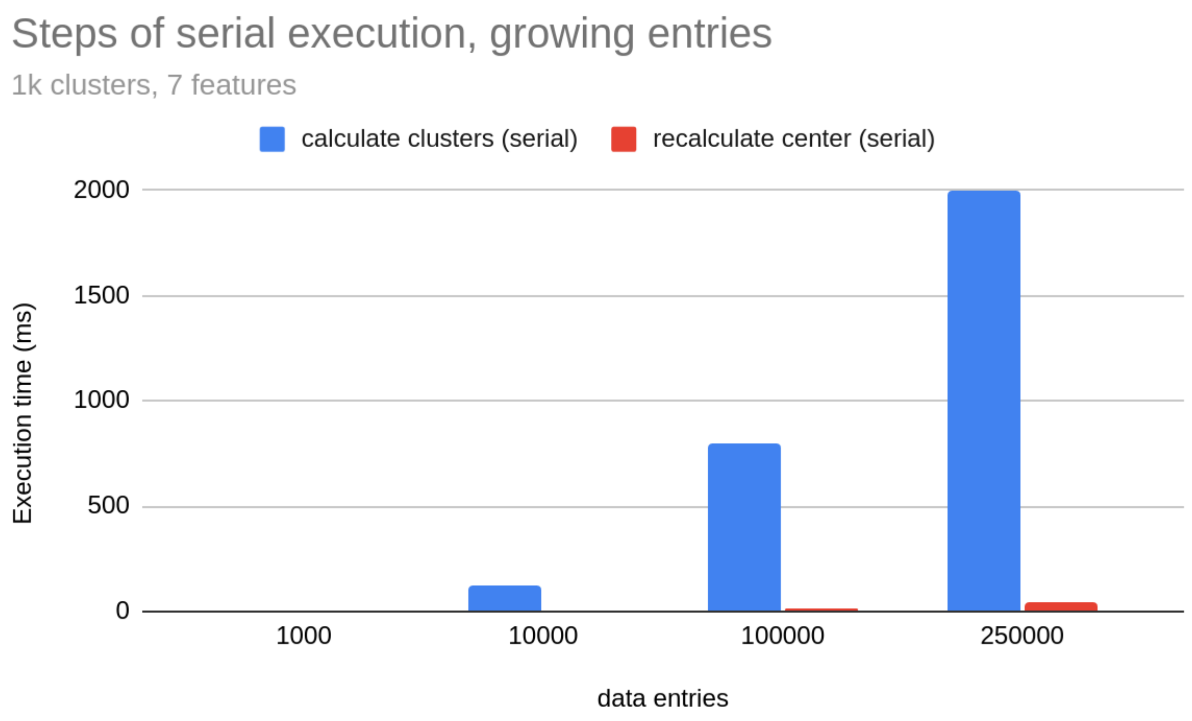
Profiling

I profiled the execution time of the serial algorithm to get an idea of what portions of the code took the longest to execute. These are the two major steps of the algorithm.

- `calculate clusters` calculates the closest cluster to each point and sets them
- `recalculate center` calculates the cluster's center based on its newly adjusted point list



We see that as we increase the number of clusters, the `calculate clusters` time is the limiting factor. Note the scale.



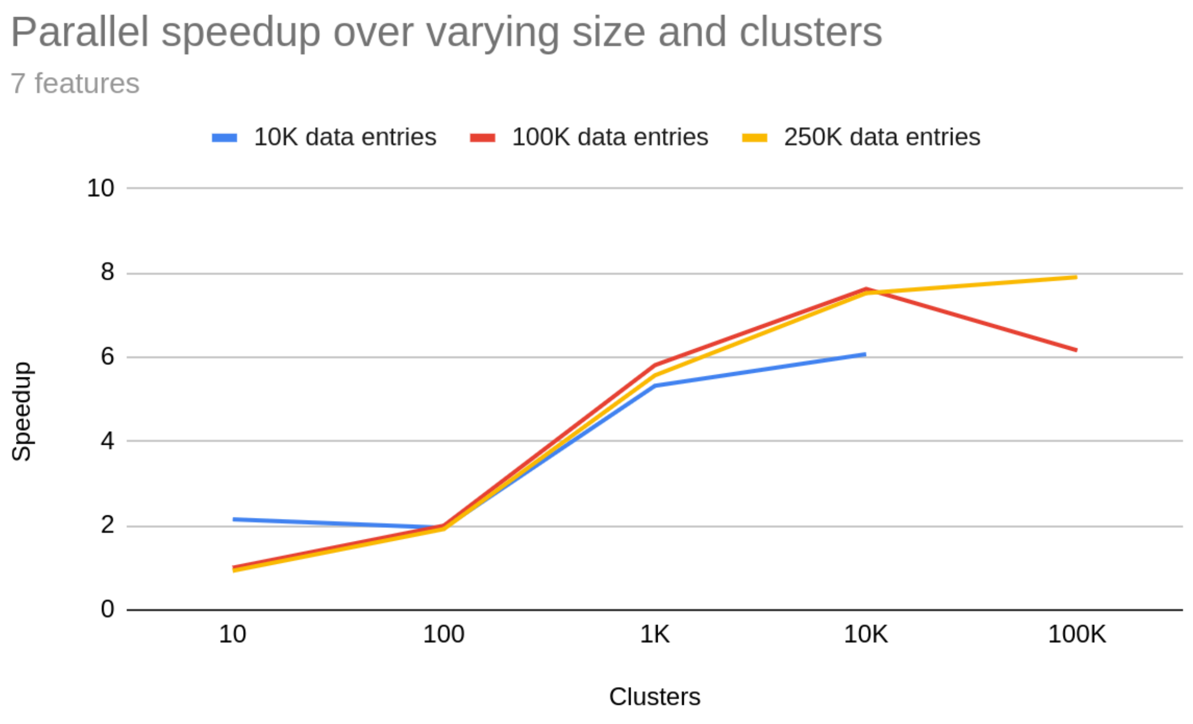
Again when comparing growing entry size, `calculate clusters` is still the limiting factor. It is clear that the best performance will be attained by optimizing this step.

Initial parallelism

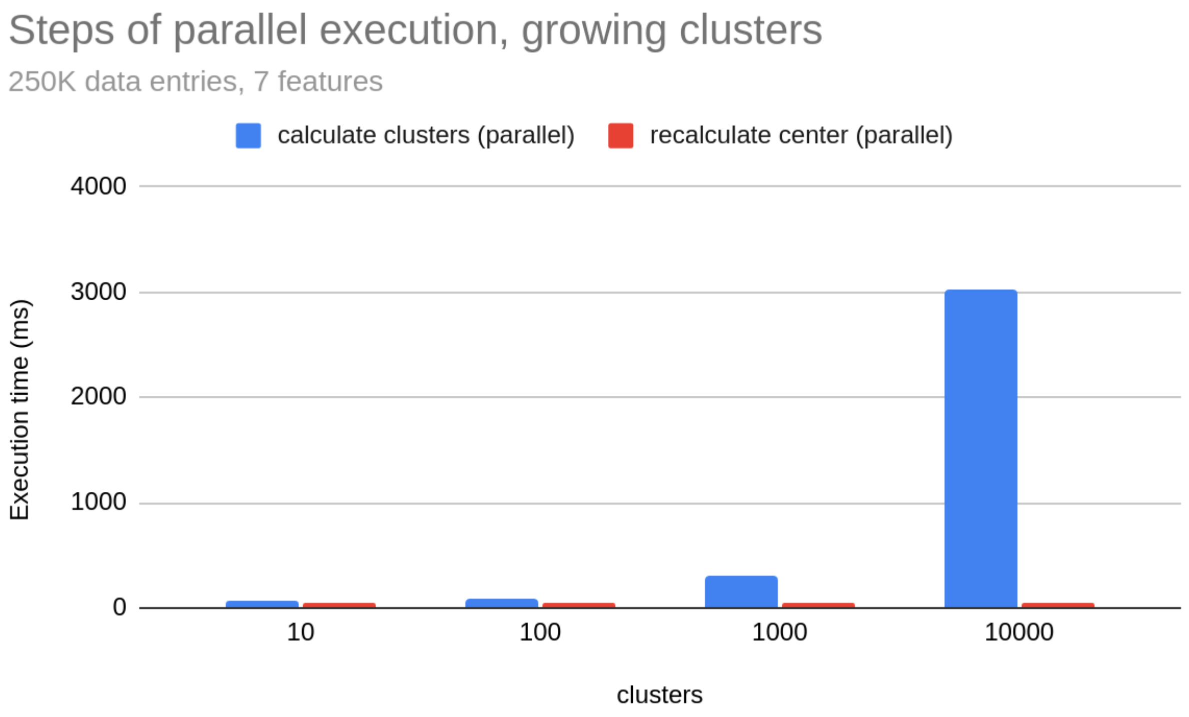
The step `calculate clusters` is easily parallelizable by iterating over all points with a TBB `parallel_for`. Modifications to clusters happen after this step, and I collected the points that had to be added/removed to vectors by pushing to a `concurrent_vector` during the `parallel_for`. Next, I iterated over clusters in parallel and applied the add/remove point vectors constructed in the first step.

I then parallelized `recalculate center` by iterating over clusters in parallel again with `parallel_for`. This step has much less impact than the first on the overall runtime of the algorithm. See the above section on profiling.

This initial approach showed good results for performance over sequential with high cluster counts. There is no data point for 100k clusters and 10k data entries because the cluster count cannot be larger than the number of data entries.

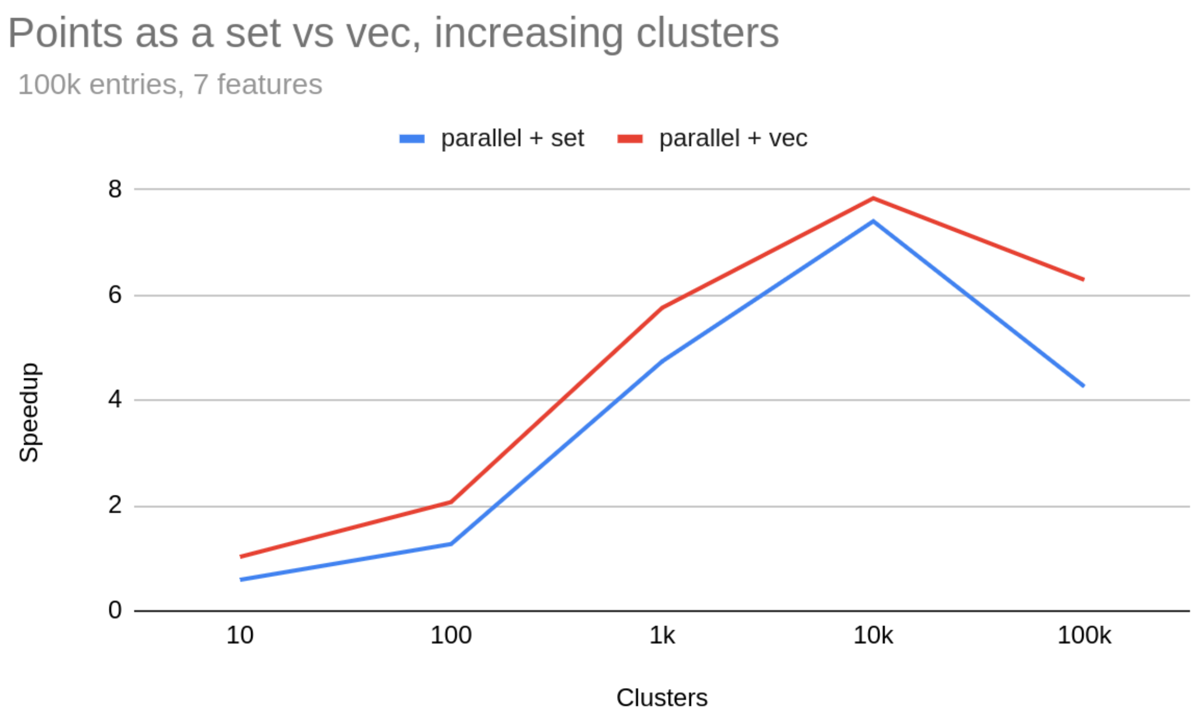


However, the `calculate clusters` step is still the bottleneck. I attempt to optimize this below.

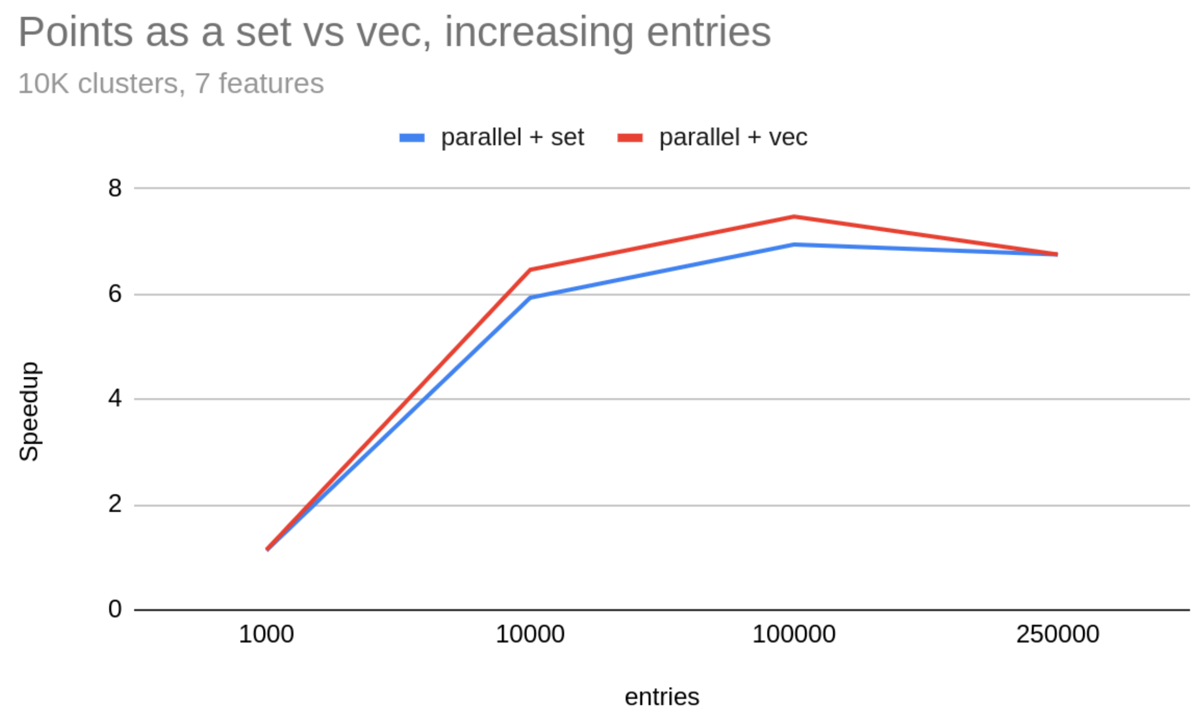


Points as a Vector vs Set

`calculate clusters` can involve a lot of add and remove operations from a cluster's Point vector. To attempt to optimize this, I changed the cluster's list of points from a vector to a set for O(1) addition and removal of points. Initially, this showed a great speedup. But I realized I was benchmarking incorrectly and the results weren't nearly as good as I thought. With correct benchmarking, storing Points in a vector outperforms storing Points in a set. This could be due to iterating over points more often than directly referencing points or TBB better handling parallelization with vectors. I also optimized the remove operation on the vector by reconstructing the vector in parallel (without certain values) instead of finding and removing specific elements.

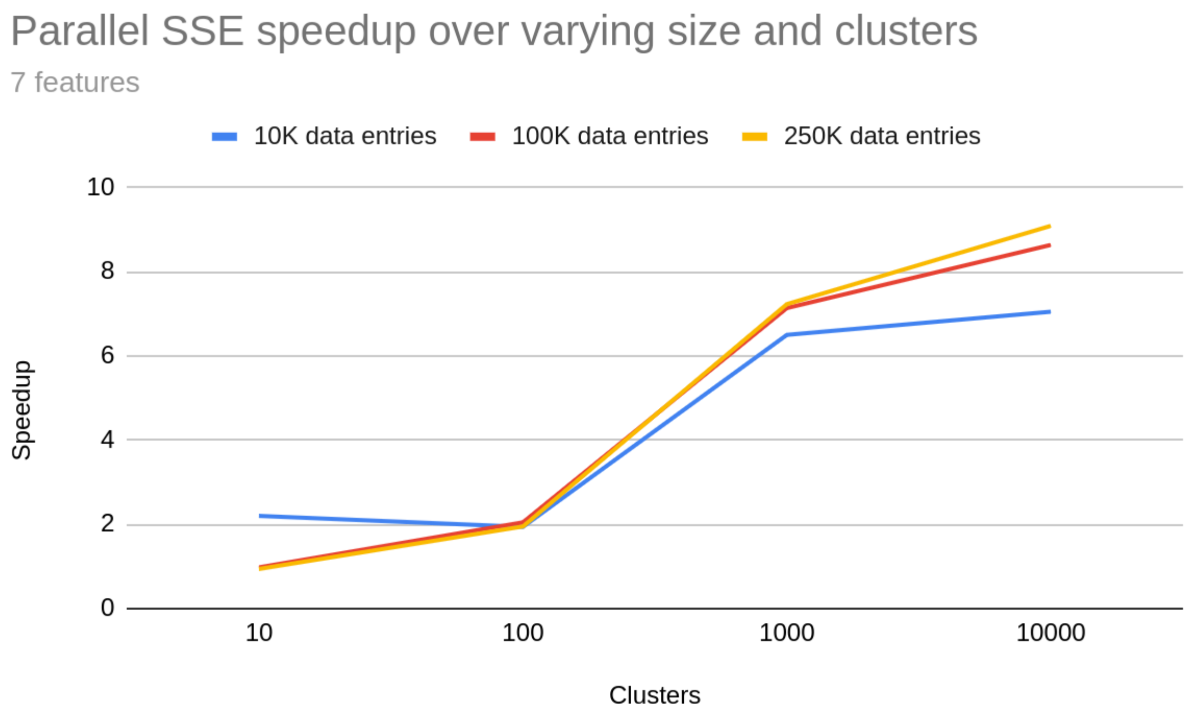


I test the comparison on another axis, increasing the data entries. Also in this result we see storing data entries in a vector is better.



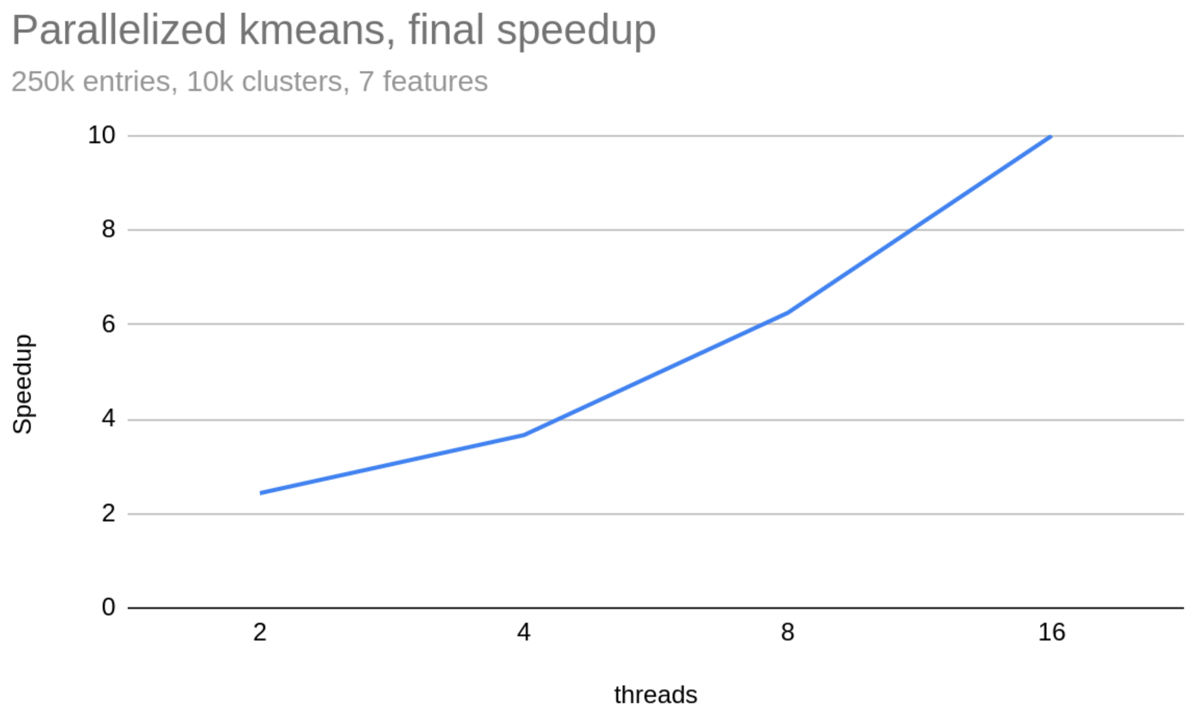
Streaming SIMD Extensions

The function `getIDNearestCenter` takes a lot of computations and can be optimized using SIMD extensions. A simple optimization that should show improvement is using SSE2. I use this to batch operations over features 2 at a time. This results in great performance improvements at larger K values. Even breaking 9x speedup over sequential execution for some workloads!



Compile time constants & caching

I added caching for looking up point values in the `getIDNearestCenter`, because in the current iteration I was looking up the same exact point.getValue(i) K times. My first attempt I allocated the cache at a constant size 7, the number of features I have in all my data. It would be better to be dynamic so I instead allocated it to be size `total_values`. I saw a huge performance loss when I did this. I realized many constants were being declared at runtime when they're read in from the file instead of compile time (an artifact of the sequential execution), preventing many compiler optimizations! With caching and compile time constants, I get even better performance, up to 10x speedup at 16 threads. As this is the final state of my program, I'll provide the graph over threadcounts.


Parallelized kmeans, final speedup
250k entries, 10k clusters, 7 features
10
8
6
4
2
0
Speedup
2
4
8
16
threads