Brandon Horner
Michael Gowanlock
CS499: Parallel Programming
26 November 2019

Assignment 6: Improving Application Performance Using OpenMP and Algorithmic Transformations

Question 1

1) Without compiler optimization, report the speedup you achieved, and the number of cores. (See Figure 1 & 2).
T1 = 176.339013 with 1 core
T2 = 44.865112 with 4 cores
Speedup = T1/T2 = 176.339013/44.865112

Speedup = 3.930426 with 4 cores
This parallel speedup is almost 4 times faster than the previous sequential implementation which is very good.

2) Without compiler optimization, report the parallel efficiency you achieved, and the number of cores.
Parallel Efficiency = (T1/T2)/n-cores = 3.90426 / 4

Parallel Efficiency = .976065 with 4 cores
This is a very good efficiency, almost 100% of the 4 cores are being utilized.

3) With compiler optimization (-O3), report the speedup you achieved, and the number of cores. (See Figure 3 & 4).
T1 = 17.180589 with 1 core
T2 = 4.348458 with 4 cores
Speedup = 17.180589 / 4.348458

Speedup = 3.950961 with 4 cores
The parallel speedup is again, almost 4 times faster on 4 cores, a great speed up.

4) With compiler optimization (-O3), report the parallel efficiency you achieved, and the number of cores.

Parallel Efficiency = (T1/T2)/n-cores = 3.950961 / 4

Parallel Efficiency = .987740 with 4 cores
This is a slightly higher efficiency than without the optimizations; nearly 100% of the 4 cores are being utilized.

5) For all items above, reason about your performance gains. Is your speedup or parallel efficiency good or bad? Why?
See answers above for initial interpretations. Overall, the parallel speedup was about 4 times faster. This is a great speedup considering 4 cores were used. This also translates into a very good parallel efficiency. Since there was almost a 4 times speedup on 4 cores, there was almost a 100% parallel efficiency which would have been *perfect* parallel efficiency.

Question 2

1) A description of how your new algorithm works. That is, how does it reduce the computational load in comparison to the O(n^2) algorithm? What overheads does it have? Overheads here refer to the "extra steps" that are needed to: (a) make the program utilize threads; and (b) reduce the amount of computation. Provide illustrative examples of how your new algorithm works, as appropriate.
My new algorithm works in a similar way, but first I added some overhead. Of course there is the overhead of utilizing threads, but I also do some preprocessing by sorting the points by their x values. This preprocessing did not seem to add much time to the total execution time.

I combine the fact that the points are sorted with smaller chunk sizes equal to epsilon * 2 (see Figure 0). Then, I check to see if points are within epsilon of the point in the x direction, if they are not, then they are not considered in the euclidean calculations.

I also reduce computation loads by not calculating a euclidean distance for a point and itself. These two computation reductions come at the cost of comparing two x values, but it is faster than doing a euclidean distance calculation which consists of 8 memory accesses and 7 floating point operations.

Last but not least, I removed the square root operation in the euclidean distance calculation. Instead, I check the distance squared with epsilon squared to see if it is smaller. My brute force

implementation would have gone faster if this optimization was realized earlier.

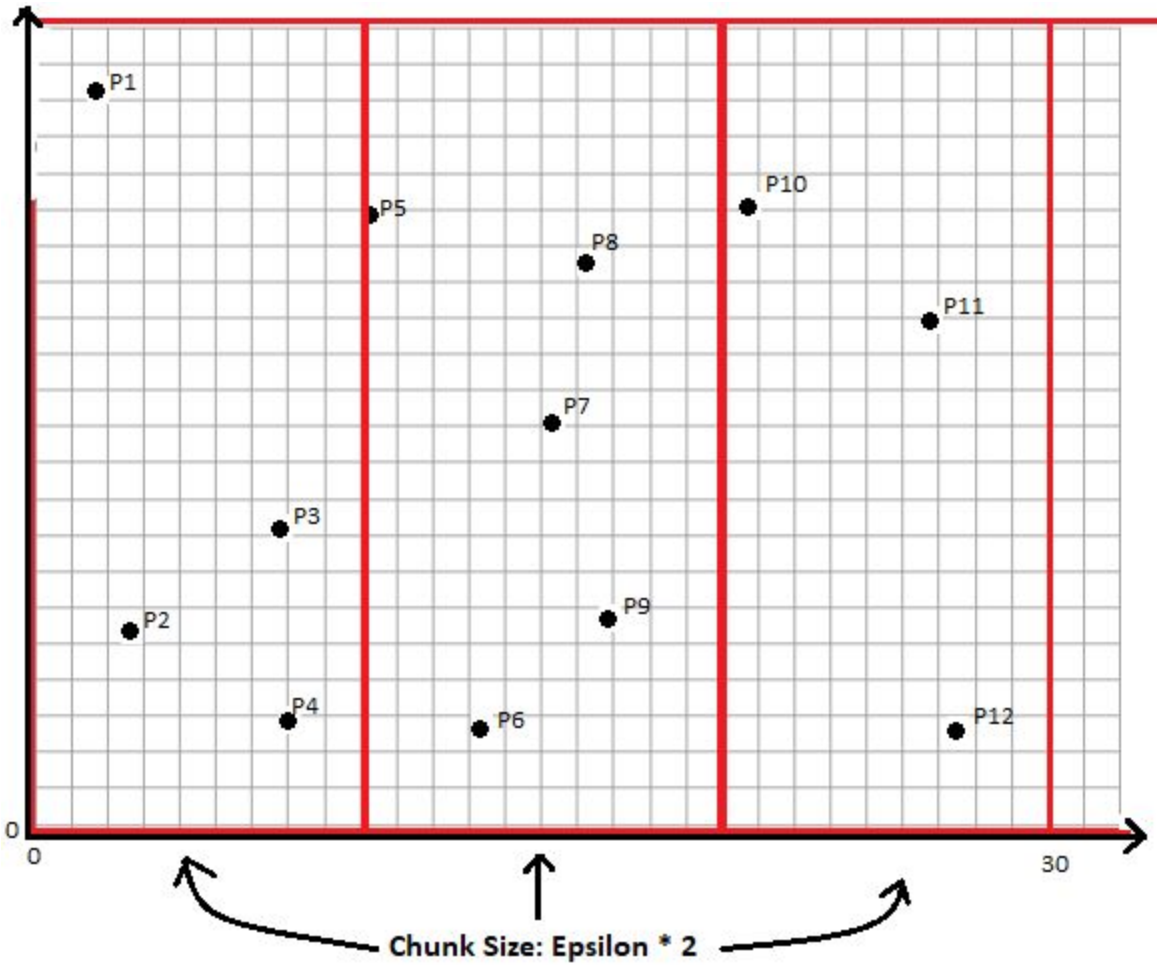

Figure 0.

2) Without compiler optimization, report the number of seconds it took to run the program (averaged over 3 trials), the speedup you achieved, and the number of cores.
(See Figure 5).
T1 = 176.339013 with 1 core
T2 = 12.019906 with 4 cores
Speedup = T1/T2 = 176.339013/12.019906

Speedup = 14.670582 with 4 cores
This speedup is 14.6 times faster than the brute force sequential implementation which is excellent.

3) With compiler optimization (-O3), report the number of seconds it took to run the program (averaged over 3 trials), the speedup you achieved, and the number of cores.

(See Figure 3 & 6).
T1 = 17.180589 with 1 core
T2 = 2.887547 with 4 cores
Speedup = 17.180589 / 2.887547

Speedup = 5.949891 with 4 cores
The parallel speedup is again, almost 6 times faster than the single core brute force implementation with compiler optimizations a great speed up.

4) Compare performance by reporting the ratio of the response times over the brute force implementation in Question 1. For example, you execute both algorithms in parallel, and obtain: T1- brute force response time, and T2- your new algorithm response time. Report the ratio T1/T2. (See Figure 2, 4, 5, 6).
Parallel brute force response time w/o compiler optimizations = T1 = 44.865112
My new algorithms response time w/o compiler optimizations = T2 = 12.019906
44.865112 / 12.019906 = 3.732568

Parallel brute force response time w/ compiler optimizations = T1 = 4.348458
My new algorithms response time w/ compiler optimizations = T2 = 2.887547
4.348458 / 2.887547 = 1.505935

5) Answer these questions: Is your new algorithm faster? Why or why not? What optimizations worked well? What ideas did you try that did not perform well?
I tried to create a section-based solution. In my original plan, I naively separated the quadrants of the coordinate grid into positives and negatives. After running one time I saw that all of the coordinates were actually positive numbers (in the first quadrant). Once I realized that all the coordinates were positive; I cut the problem area in quarters and assigned each quarter to a section.
In the end, I got a slight slowdown compared to the brute force parallel solution of the problem. This is probably because of a slight load imbalance, certain quadrants had more points. I also did not really cut down the amount of calculations and increased the comparisons which likely contributed to my slowdown.

Next, I tried sorting the data struct with the merge sort algorithm supplied in class. I believe sorting the points by x will allow for less memory cache misses and will help when trying to reduce point to point distance calculations. Cutting down euclidean distance calculations by seeing if points are close to each other, or if they are about to be compared to themselves, improved the execution time a lot. Something that was strange to me but helped my execution time was limiting how many *#pragma omp* calls I had. I took the parallelization out of mergesort and quick sort and my execution time decreased with minor sorting overhead. I imagine there is more overhead than I think when utilizing threads. I also changed the schedule chunk size from

n/num_threads to epsilon*2 which reduced my execution time (likely due to better core utilization). Another optimization I tried was taking the square root operation out of the distance calculation. Instead, I compare the squared distance with the squared epsilon value. This led to a significant decrease in computation time. This new algorithm is quite a bit faster because of these optimizations.

6) Give a more detailed comparison of the performance of your algorithm in comparison to the brute force algorithm. Use other metrics to convey why your program is now faster. Examples include: reduction in floating point operations, reduction in the number of points compared, counting cache misses, and others.

My new algorithm is still about *O(n^2)* worst case. However, there are *n**8 (8 memory accesses occur in the euclidean distance*)* memory accesses that are skipped by checking first if the algorithm is comparing a point to itself.

When epsilon is 5.0 and n is 100,000, there are 655,108,988 euclidean calculations that are skipped because those points are farther than epsilon away in the x direction from the current point. This number will go down with a higher epsilon, and will go up with lower epsilon values. This in theory would reduce the memory accesses by 6 * 655 million (since there are 8 memory accesses happening in the euclidean distance, and only 2 memory accesses for the comparisons to skip this calculation). It would also reduce the amount of floating point operations required per execution by 7 * 655 million, since there are 7 floating point operations in the euclidean calculation.

My algorithm is also sorted based on x values. When comparing points in close proximity in the array, there should be less cache misses.

7) Do these metrics translate linearly to the observed reduction in response time as compared to the brute force algorithm? Why or why not?

I saw a significant decrease in the response time when applying each optimization discussed. While the reduction of floating point operations seemed to only reduce the execution time by a small factor, reducing memory accesses had a greater effect. This is because memory accesses are slow when compared to processing computations. Sorting the program actually helped significantly. I saw a speedup in execution time (without compiler optimizations) by around 60% versus the time it took to execute on the unsorted list. This suggests that there has been a reduction in cache misses.

Bonus [5 Points]

I would like to be considered for the bonus points as I personally saw a lot of speedups.

Reference screenshots

Brute-force sequential executions (Figure 1):

```
tuco@tuco-VirtualBox:~/Documents$ ./sequential 5.0

Size of dataset (MiB): 1.525879

Total time of generating data (s): 0.004172

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 176.230685

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 176.492710

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 176.175375

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 176.293681

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 176.260660

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 177.110290
```

Parallel executions (Figure 2):

```
tuco@tuco-VirtualBox:~/Documents$ ./parallel 5.0

Size of dataset (MiB): 1.525879

Total time of generating data (s): 0.004627

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 44.623022

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 44.845302

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 45.127012

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 44.986580

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 44.988218

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 44.669433
```

Brute force sequential with -O3 optimization (Figure 3):

```
tuco@tuco-VirtualBox:~/Documents$ ./sequential 5.0

Size of dataset (MiB): 1.525879

Total time of generating data (s): 0.004577

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 17.154805

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 17.183788

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 17.203176

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 17.145862

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 17.198127

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 17.089049
```

Parallel with -O3 optimization (Figure 4):

```
tuco@tuco-VirtualBox:~/Documents$ ./parallel 5.0

Size of dataset (MiB): 1.525879

Total time of generating data (s): 0.004796

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 4.346749

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 4.344700

epsilon = 5.0
Total count of points within epsilon of each other: 879688
Total time (s): 4.353925

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 4.379912

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 4.389949

epsilon = 10.0
Total count of points within epsilon of each other: 3211140
Total time (s): 4.426644
```

New algorithm without compiler optimizations (Figure 5):

```
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.044120

Total count of points within epsilon of each other: 879688
Total time (s): 12.182914
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.043759

Total count of points within epsilon of each other: 879688
Total time (s): 11.954201
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.041353

Total count of points within epsilon of each other: 879688
Total time (s): 11.922604
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.047983

Total count of points within epsilon of each other: 3211140
Total time (s): 11.993058
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.040878

Total count of points within epsilon of each other: 3211140
Total time (s): 11.996144
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.035725

Total count of points within epsilon of each other: 3211140
Total time (s): 12.012698
```

New algorithm with compiler optimizations (Figure 6):

```
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.031385

Total count of points within epsilon of each other: 879688
Total time (s): 2.891401
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.035456

Total count of points within epsilon of each other: 879688
Total time (s): 2.884780
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 5.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.039488

Total count of points within epsilon of each other: 879688
Total time (s): 2.886459
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.032790

Total count of points within epsilon of each other: 3211140
Total time (s): 2.913286
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.038840

Total count of points within epsilon of each other: 3211140
Total time (s): 2.900995
tuco@tuco-VirtualBox:~/Documents/Assignment_6$ ./question2_bkh76 10.0

Size of dataset (MiB): 1.525879
Total time generating and sorting (s): 0.034095

Total count of points within epsilon of each other: 3211140
Total time (s): 2.899152
```