CS450: Introduction to Parallel Programming

Assignment 7: Improving Application Performance Using OpenMP and Algorithmic Transformations

Due: August 7th, 2022, 11:59 PM, MST

Preliminaries

You are expected to do your own work on all homework assignments. You may (and are encouraged to) engage in discussions with your classmates regarding the assignments, but specific details of a solution, including the solution itself, must always be your own work. See the academic dishonesty policy in the course syllabus.

Submission Instructions

You should turn in an electronic archive (.zip, .tar., .tgz, etc.). The archive must contain a single top-level directory called CS450_aX_NAME, where "NAME" is your NAU username and "X" is the assignment number (e.g., CS450_a7_ab1234). Inside that directory you should have all your code (no binaries and other compiled code) and requested files, named exactly as specified in the questions below. In the event that I cannot compile your code, you may (or may not) receive an e-mail from me shortly after the assignment deadline. This depends on the nature of the compilation errors. If you do not promptly reply to the e-mail then you may receive a 0 on some of the programming components of the assignment. Because I want to avoid compilation problems, it is crucial that you use the software described in Assignment 0. Assignments need to be turned in via BBLearn.

Turn in a single pdf document that outlines the results of each question. For instance, screenshots that show you achieved the desired program output and a brief text explanation. If you were not able to solve a problem, please provide a brief write up (and screenshots as appropriate) that describes what you tried and why you think it does not work (or why you think it should work). You must provide this brief write up for each programming question in the assignment.

This pdf should be independent of the source code archive, but feel free to include a copy in the top level of that archive as well.

Overview

A simple operation that is used in many applications is calculating distances between points. In this assignment, we will use the Euclidean distance. For example, if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, then the distance is as follows: $distance(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. In this assignment, you will calculate the distance between points, and report the total number of distances between points that are within some distance threshold, ϵ. Thus, you will count the total number of occurrences $distance(p_1, p_2) \leq \epsilon$. As ϵ increases, the total number of points within ϵ should increase. For validation instructions, see the end of the assignment.

Illustration

Consider Figure 1 with $p_1 = (2, 2), p_2 = (3, 4), p_3 = (5, 2), p_4 = (-4, 4), p_5 = (-2, -3), p_6 = (2, -1)$. An example calculation of the distance between a few pairs of points (to a few decimal places) are as follows:

- p_1 and p_2: $\sqrt{(2-3)^2 + (2-4)^2} = \sqrt{5} = 2.236$
- p_2 and p_3: $\sqrt{(3-5)^2 + (4-2)^2} = \sqrt{8} = 2.828$

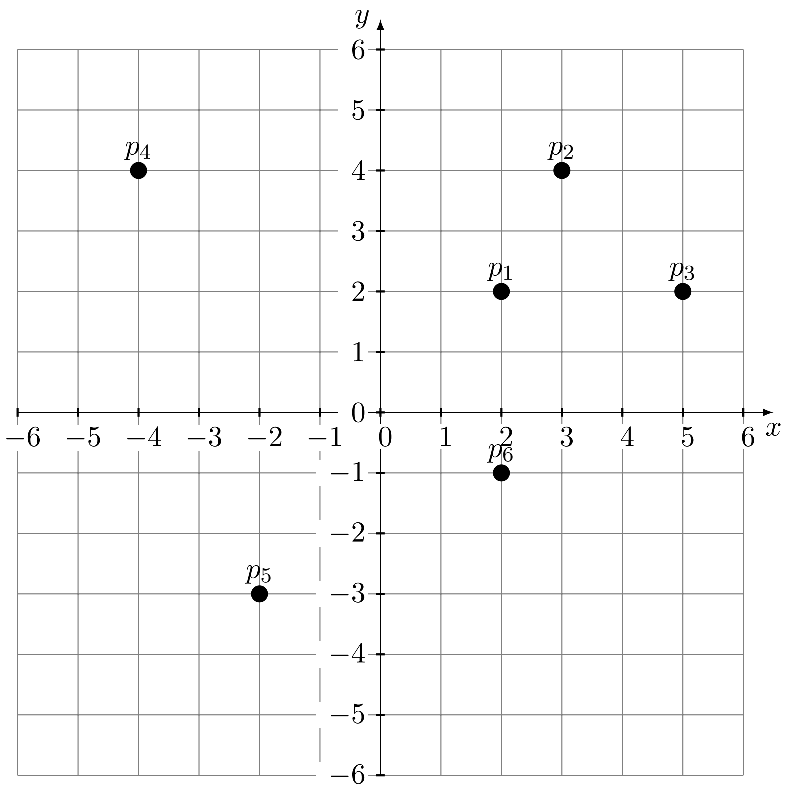

Figure 1: Example. See text for details.

- p_6 and p_5: $\sqrt{(2-(-2))^2+(-1-(-3))^2} = \sqrt{20} = 4.472$

Consider all the points in Figure 1.

- If $\epsilon = 1$ or $\epsilon = 2$, then no pairs of points are within ϵ.
- If $\epsilon = 3$: $d(p_1, p_2) \leq \epsilon$, $d(p_2, p_3) \leq \epsilon$, $d(p_1, p_6) \leq \epsilon$, $d(p_1, p_3) \leq \epsilon$.
- If $\epsilon = 100$: all points are within ϵ of each other.

Details

The Euclidean distance is symmetric, i.e., $distance(p_1, p_2) = d(p_2, p_1)$. You must "double count" these pairs. That is, if p_1 and p_2 are within ϵ then p_2 and p_1 are also within ϵ, and both must be counted towards the total number of points within ϵ of each other. You must also add to the total count the distance between a point and itself. E.g., $distance(p_1, p_1) = 0$ will always be within ϵ. These requirements make the total count easier to compute, as these are considered corner cases.

If $\epsilon = 100$ in Figure 1, then all points are within ϵ of each other. Including the "double counting" above, and a point counting itself, this means that the total number of counts you should compute are: $6^2 = 36$.

If $\epsilon = 3$ (see example above), then the total count of points within ϵ of each other should be 14. All points are within ϵ of themselves (6), and the 4 pairs of points within ϵ above yield 4×2=8.

Program Guidelines

Your program will take as input on the command line a value for ϵ (declared as a double precision float). Your program will output the total number of point comparisons that are within ϵ. I will provide you with starter code that passes ϵ in on the command line.

Table 1: Measurements and metrics for the subquestions in Question 1. All response times have been averaged over 3 time trials, each using separate executions of the program. The column "No Opt." refers to executing the program without compiler optimization. Blank lines indicate values that you need to add to the table.

Subquestion (ϵ =_____)	#Cores	Time (s) No Opt.	Time (s) w/ -O3	Speedup	Parallel Efficiency
Sequential	1	_____		1	1
(a)	_____	_____		_____	_____
Sequential	1		_____	1	1
(b)	_____		_____	_____	_____

Question 1: Baseline with OpenMP

•Write the brute-force sequential algorithm to compute the total number of points that are within ϵ according to the guidelines above. This algorithm compares all points to each other, calculating the distance and determining if they are within ϵ. Hint: the algorithm is $O(N^2)$, where N is the input size (total number of data points).

•Parallelize the sequential code with OpenMP. The number of threads you use should match with the number of threads or your computer/virtual machine.

•Compare the execution time of the sequential vs. parallel algorithm.

•For all items below, include the number of seconds it took to run the program. Report an average of 3 time measurements.

(a) Without compiler optimization, report the number of cores, speedup, and parallel efficiency achieved.

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

(c) For the items above, reason about your performance gains. How is your speedup or parallel efficiency? Explain.

Example compilation using optimization: `g++ main.cpp -o main -O3`
Example of running the program with the $\epsilon = 10$ as a parameter: `./main 10.0`
Submission:
I provide you with `point_epsilon_starter.cpp`. This code generates the data, gets ϵ from the command line, and shows you where to time your code. Download it, rename it `question1_NAME.cpp`, complete the question, and turn in an archive with this file. Make sure that you include responses to Questions (a)–(c) above in your pdf writeup. In addition, fill out a table that looks like that shown in Table 1.

Question 2: Algorithmic Transformation

The above algorithm is $O(N^2)$, which is not very efficient. For instance, in Figure 1, if $\epsilon = 3$, then we know visually that there is no reason to compare p_3 and p_4, because they are clearly far away from each other. Modify your brute force $O(N^2)$ algorithm to be more efficient (i.e., fewer total distance calculations, fewer floating point operations, or other ways of reducing the computational load of comparing points). Parallelizing a program to get better performance is good. But in the end, not computing something remains more efficient! This will require an *algorithmic transformation*, meaning that you cannot use the brute force algorithm in Question 1. You will need to be *creative* to reduce the total computational load. Use OpenMP to parallelize the code. Remember that the algorithm should still give the correct result for a given value of ϵ.

Report the following information:

(a) 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

Table 2: Measurements and metrics for the subquestions in Question 2. All response times have been averaged over 3 time trials, each using separate executions of the program. "No Opt." refers to executing the program without compiler optimization. Blank lines indicate values that you need to add to the table.

Subquestion ($\epsilon =$_____)	#Cores	Time (s) No Opt.	Time (s) w/ -O3	Speedup	Parallel Efficiency	T1/T2 No Opt.	T1/T2 w/ -O3
Sequential	1	_____		1	1		
(b)	_____	_____		_____	_____		
Sequential	1		_____	1	1		
(c)	_____		_____	_____	_____		
(d)						_____	_____

"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.

(b) Without compiler optimization, report the number of seconds it took to run the program (averaged over 3 trials), the speedup you achieved, the parallel efficiency, and the number of cores.

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

Compare the performance between your new algorithm and the parallel algorithm in Question 1 (executed using no compiler optimization and with -O3). Report the following information:

(d) 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 T1/T2 ratios.

(e) 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?

You must provide a detailed performance comparison. Parallel speedup gives a very coarse-grained overview of performance. But it does not actually tell us what makes the parallel program faster. Hint: you should collect these metrics separately from timing your program because collecting performance metric data may slow down your optimized program!

(f) 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.

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

Submission:
Use the same starter code as above as a starting point. Download it, rename it `question2_NAME.cpp`, complete the question, and turn in an archive with this file. Make sure that you include responses to Questions (a)–(g) above in your pdf writeup. In addition, fill out a table that looks like that shown in Table 2.

Validation of all Questions

The starter program randomly generates the data to be analyzed. Do not change the seed for the random number generator. I have selected the number of points (N in the program) to be 131,072 (2^{17}). The sequential brute force algorithm on my computer without compiler optimization takes roughly 80 s to execute. This gives you a lot of room to optimize Question 2. It is enough work such that the effects of *efficient* algorithmic changes should be noticeable. **Parallelization is not the unique answer to this question. In addition to parallelization, you need to reduce the computational complexity of your program.** Since $N = 131072$ means that you will need to wait roughly 100 s for each execution, you may want to change N for testing purposes. Below are some values of N and ϵ and the corresponding total number of points within ϵ. Note that depending on your machine and the math library it uses, you may have slightly different results.

You can check that your program is correct using the following results:

- $N = 131072$, $\epsilon = 10.0$, total count: 5476288.
- $N = 131072$, $\epsilon = 5.0$, total count: 1472216.
- $N = 1024$, $\epsilon = 10.0$, total count: 1402.
- $N = 1024$, $\epsilon = 5.0$, total count: 1136.

Upon submitting your programs, they should be configured to use $N = 131072$. Supply screenshots of your program outputting the correct values for the total number of points within ϵ as shown above for $N = 131072$. Note that I might test your programs with large values of ϵ, where the result can be N^2. With $N = 131072$, this value is high. Your program should thus be robust and work nonetheless (i.e., the result variable should not overflow).

Grading

Because you are implementing an algorithmic transformation which requires thinking about different ways to implement the problem, there are a wide range of possible solutions. This assignment will be graded based on the level of effort you put into your implementation, and the amount of thought you have given to effectively computing the distance between points.

The brute force algorithm is $O(N^2)$. However, there are solutions that can compute the distances between points within $O(N\log N)$. If we assume that the brute force algorithm takes 100 seconds to compute the solution sequentially for $N = 131072$, then the time to perform a single distance calculation between two points is as follows: 100 s$/(100000^2) = 10^{-8}$ s. If the $O(N\log N)$ algorithm were to have no overhead, then the lower bound response time would be $100000\log_2(100000) \cdot 10^{-8}$ s $= 0.0166$ s. This is a large reduction in the response time; however, the $O(N\log N)$ algorithm is expected to have overhead, and you will not be able to compute the distance between all points in only 0.0166 s. Since it is possible to achieve a large reduction in the response time, there are many optimizations that can be used to reduce the time needed to compute the distance between the points. Keep in mind that the above "back-of-the-envelope" calculation is for the sequential algorithm, and parallelizing the algorithm can achieve further performance gains. Also, note that I do not expect you to come up with an $O(N\log N)$ solution to the problem!

Here are a few general grading guidelines/scenarios for Question 2 regarding your transformation of the brute force algorithm.

- To get all the points: In your sequential implementation, you should have a substantial reduction in the amount of work as compared to the brute force approach. This could be a reduction in floating point operations, or number of distance comparisons (as described in Question 2). You must parallelize the algorithm as appropriate. For instance, if you have 4 cores and 4 threads, you shouldn't have 3 threads idle while one thread is performing the computation.
- Fewer points: In your sequential implementation, you should have a moderate reduction in the amount of work as compared to the brute force approach. Like the criterion above, you must parallelize the algorithm as appropriate to your algorithm.

- Even fewer points: In your sequential implementation, your solution to reducing the number of distance calculations yields a modest reduction in the amount of work as compared to the brute force approach. The parallelization of the algorithm may not be efficient (e.g., may have too much overhead).

Bonus 1: Ratio my Algorithm [2 points]

If you do not want your assignment to be considered for these bonus points, do not respond to this section in your write-up.

My brute-force implementation, as executed on my machine with 1 thread/core (using -O3) executes in 33.26 s for both $\epsilon = 5.0$ and $\epsilon = 10.0$. The optimized version executes in 0.34 s ($\epsilon = 5.0$) and 0.67 s ($\epsilon = 10.0$). The ratio of the brute force to optimized algorithm response time is thus 97.8 and 49.7, for $\epsilon = 5.0$ and $\epsilon = 10.0$, respectively.

The sequential time to execute the programs with compiler optimization (-O3) has been obtained for Questions 1 and 2. **Use a sequential version of your algorithm**. Respond to this question with your response times and ratios for $\epsilon \in \{5, 10\}$. I will run your program on the same machine to ensure consistency. Furthermore, I will use my brute force algorithm to make the calculation. If you beat or match these ratios, you will get two bonus points.

Table 3: Response time ratios for the bonus question. All response times have been averaged over 3 time trials, each using separate executions of the program. The column "No Opt." refers to executing the program without compiler optimization. Blank lines indicate values that you need to add to the table.

	T1/T2 (-O3) $\epsilon = 5.0$	T1/T2 (-O3) $\epsilon = 10.0$
Sequential:	_____	_____

Bonus 2: Ratio Others [Up to 3 points]

If you do not want your assignment to be considered for these bonus points, do not respond to this section in your write-up.

Bonus points will be awarded for the top three fastest algorithms, as reported in this section of the assignment as the ratio of the brute force to optimized algorithm response time using -O3 and **executed sequentially**. The algorithms are executed sequentially as you will have varying numbers of cores in your computers.

I reserve the right to retract these bonus points if the algorithms do not substantially reduce the computational complexity (i.e., if they do not satisfy the criteria to get all the points, as explained in the grading section above). I will execute all programs on the same platform to ensure a fair comparison between approaches. Be sure to complete Table 3.

You cannot get the bonus points for beating my algorithm, if you are awarded bonus points for this question.