CS/CE 102/11 - Data Structures and Algorithms Project Report
Parallel Programming

Abdullah Amir (08453), Musab Kasbati (07811)

Idea

Using parallel programming to implement and optimise algorithms covered in the course.

Novelty

Our course on Data Structures and Algorithms covered sequential algorithms and their analysis for solving problems on a single processor. Our project idea extends these algorithms into a parallel programming context, which requires a deeper understanding of data structures, algorithms, programming concepts such as threads, synchronization, and communication, and implementation of algorithms that are beyond the scope of the course.

Data Structures

1. Lists
2. Nested Lists
3. Graphs

Algorithms

Sequential and Parallel implementations of:

1. Image Convolution
2. Merge Sort
3. Closest Pair
4. Breadth-First Search

Application

We implemented the aforementioned algorithms in Python making use of the Multiprocessing library for the parallel variants. We ran most of these algorithms on data sets of different sizes and compared the difference in performance between the sequential and parallel variants as the size of the data changed.

Process

We started with an implementation of Parallel BFS with our implementation based on the theoretical version. Through testing, we identified that this may not be the best approach, especially for Python and on a machine with a limited number of cores as the overhead of allocating resources for new processes was significant. This informed our later implementations.

For the other algorithms i.e. Merge Sort, Closest Pair and Image Convolution, we worked on our own methods of parallelisation. Through an iterative system of trial and error, we landed on a rather simple approach of dividing the data into chunks (equal to the number of cores in our machine - changing number of processes usually decreased the performance) only at the initial stage of execution. These chunks could then be processed in parallel. Previously we were dividing the problem at every recursive call, incurring a significant overhead cost. The greatest intellectual challenge of the implementation then remained the efficient merging of data.

At this point, it is important for us to note that we had initially designed our algorithm on Google Colab, but, Google Colab did not offer us significant support for carrying out multiprocessing (at least in the way we

implemented it), thus we initially received less fruitful results. However, after testing on our local machine, we were able to achieve the desired results.

Results

BFS

In parallel algorithms, there is a level of randomness. We noticed this when we ran our algorithm for Parallel BFS repeatedly, occasionally receiving slightly different BFS trees. This brought our attention to the fact that processes occasionally finish in a different order.
For example, we used the following graph and using the parallel BFS algorithm we got two different BFS trees:

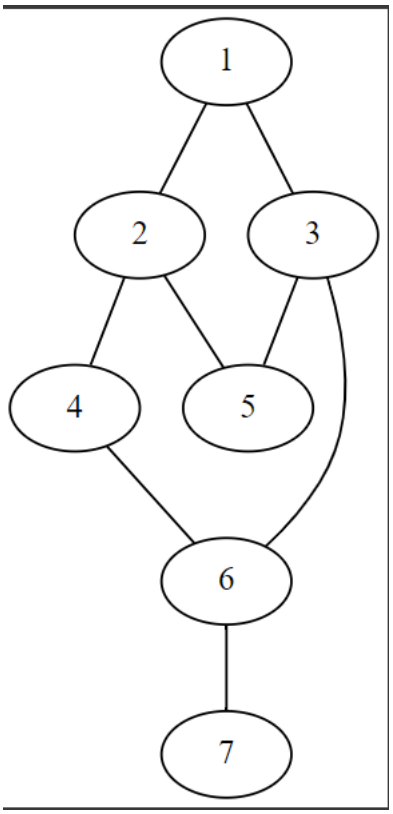

Figure 1: Graph

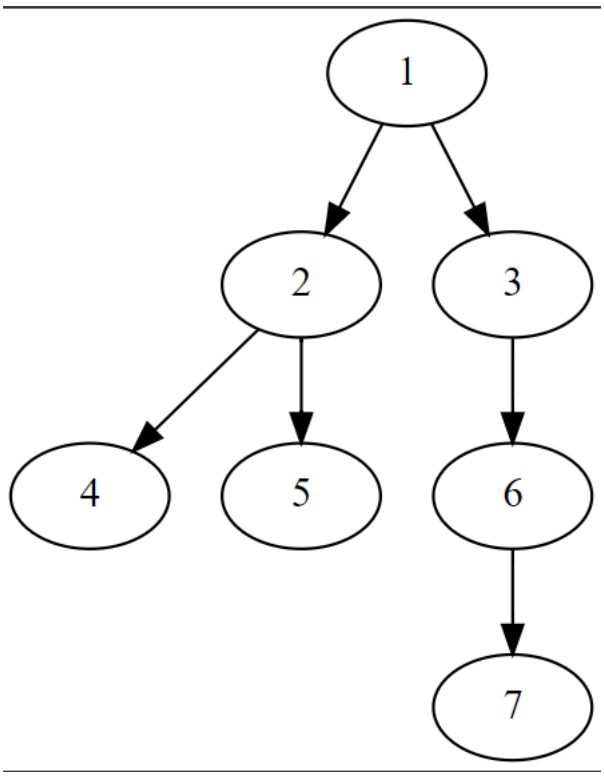

Figure 2: BFS Tree 1

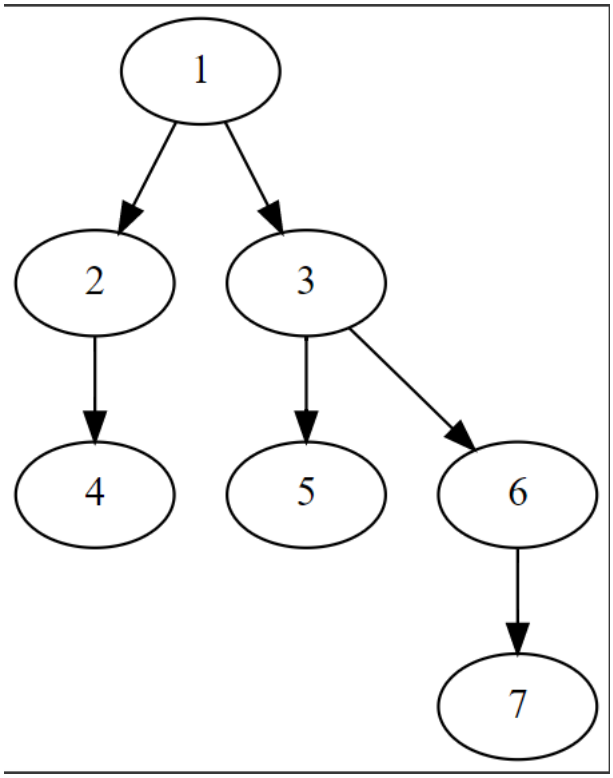

Figure 3: BFS Tree 2

Merge Sort, Closest Pair and Image Convolution

For Merge Sort, Closest Pair and Image convolution, after dividing the problem into chunks equal to the number of cores on our machine (in cases where it was appropriate), we were able to take advantage of the additional cores to the maximum while preventing overhead costs from continuously increasing. This led to the desired result where the parallel implementation of the algorithms performed better than the sequential ones for large

data sets (usually with more than 500,000 elements to process).
It was no surprise that the graphs we obtained from the time analyses of Merge Sort, Closest Pair and Image convolution were all kind of similar and had similar slopes because all of them were using Divide-and-Conquer strategies and were breaking the data into chunks, processing them and merging them back again.

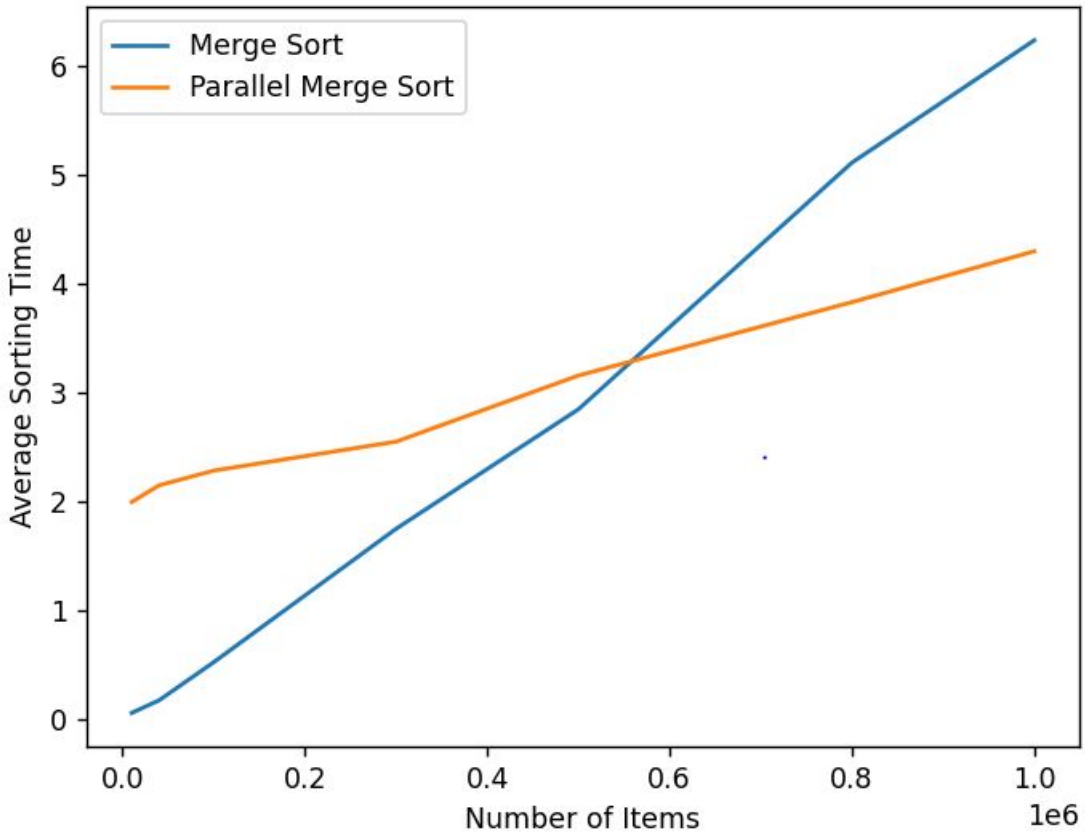

Figure 4: Merge Sort

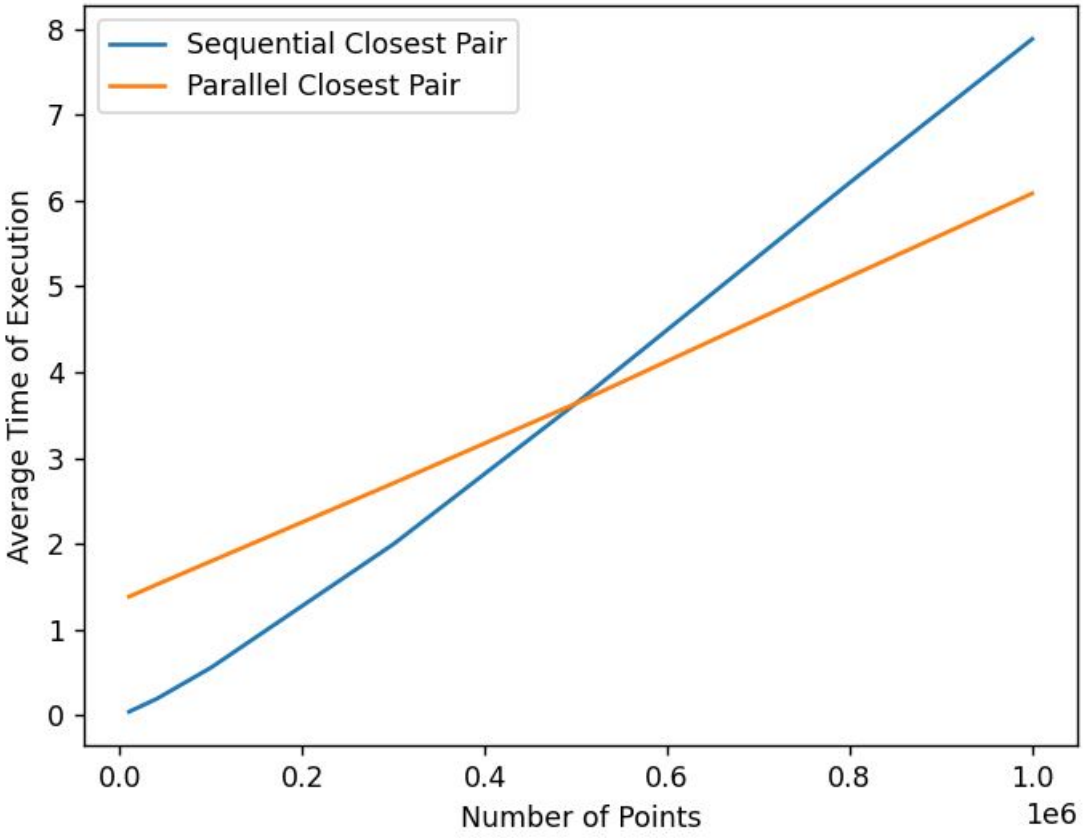

Figure 5: Closest Pair

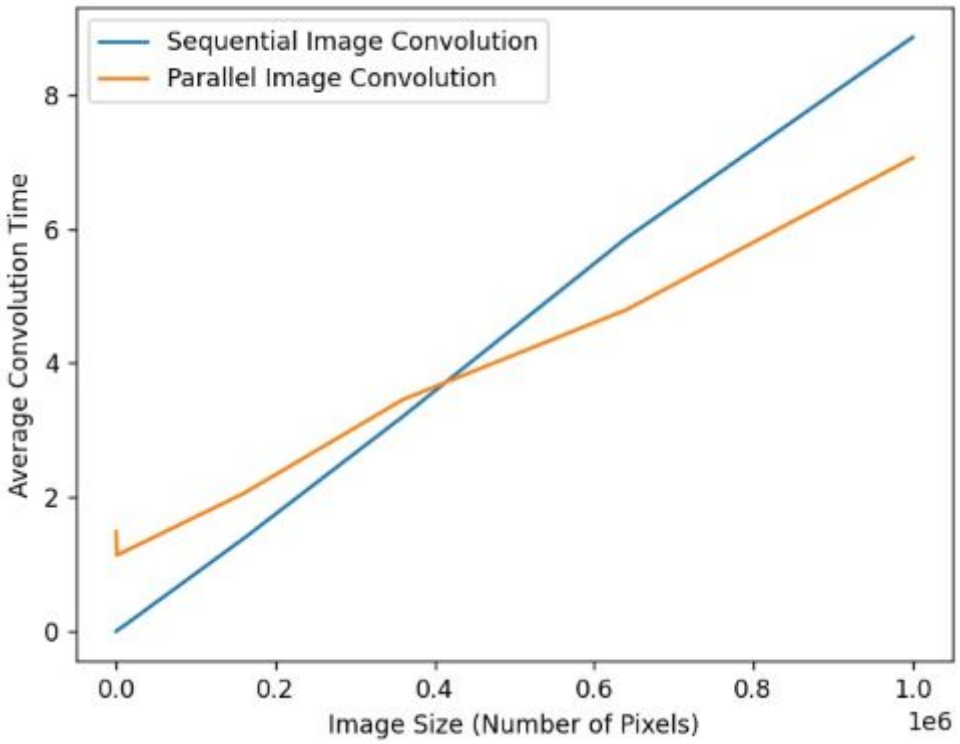

Figure 6: Image Convolution

Conclusion

Multiprocessing is an efficient way of optimising Divide and Conquer algorithms on large data sets. However, it is to be noted that the current design of computers, processors and programming languages may not be adept for supporting theoretical implementations of algorithms. We were still able to achieve fruitful results in terms of improving efficiency for Merge Sort, Closest Pair and Image Convolution algorithms through the adaptation of previously known algorithms.

References

[1] Guy E. Blelloch, Bruce M. Maggs (2010) *Parallel Algorithms*, School of Computer Science, Carnegie Mellon University.

[2] Python Software Foundation, *multiprocessing - Process-based parallelism*, Python Documentation, Version 3.X,
`https://docs.python.org/3/library/multiprocessing.html`.

[3] Corey Schafer. (2019, September 20). *Python Multiprocessing Tutorial: Run Code in Parallel Using the Multiprocessing Module* [Video], YouTube. `https://www.youtube.com/watch?v=fKl2JW_qrso`

[4] syphh. (2021, July 24). *closest_pair.py [Gist]*, GitHub.
`http://gist.github.com/syphh/b6668694edacf8cc987f89bf1270125c`.