CPE745-PARALLEL COMPUTING

Customized Topological Sort

Final Report

Group 1:

Nada Aladdin Abdel-fatah – 158613

Batool Mohammad Kayyam – 158612

With:

Dr. Fady Ghanim

Contents

1. Introduction

In this report we will explore how we implement the customized topological sort algorithm using openMP. We will discuss all details about using programming models and techniques to transform the serial code into parallel code. The following sections will explore the key distinction between topological sort and customized topological sort. We will provide an example to illustrate how the algorithm functions and discuss the process of converting serial code into parallel code. Finally, we will analyze and discuss the outcomes.

1.1 OpenMP

OpenMP is a widely adopted application programming interface (API) that facilitates parallel programming in shared-memory computing environments. By allowing developers to write parallel programs using a combination of compiler directives and runtime library routines, OpenMP provides a simple and portable approach for exploiting parallelism in programs, which allows for efficient utilization of multicore processors and shared-memory systems. It also provides support for offloading computations to accelerators like GPUs using target directives.

With openMP you can parallelize sections of your code by specifying parallel regions where multiple threads execute the code concurrently. The main advantage of openMP is its ease-of-use, developers can incrementally parallelize their code by selectively applying pragmas without having to rewrite the entire program.

By leveraging openMP you can take advantage of multi core processors and accelerate the execution of computationally intensive tasks or exploit parallelism in your algorithms ultimately improving performance and scalability.

1.2 Topological sorting algorithm

Topological sorting is a fundamental algorithmic concept that plays a critical role in numerous real-world applications ranging from task scheduling and job sequencing to project management and network analysis. Also, is a powerful algorithmic technique used across various disciplines to organize and prioritize tasks based on their dependencies. In a directed acyclic graph, it is used to determine the overall ordering of vertices so that each vertex comes before each of its children.

1.3 Customized Topological Sort

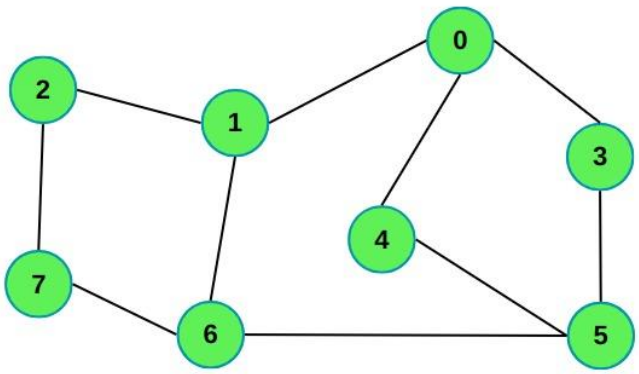
Extending to what topological sort's introduction focused about. Now we want to perform the customized form of topological sort. Customized topological sort will have the same purpose as we mentioned previously. It aims to perform sorting based on dependencies of nodes given. On the other hand, the customized version will be different in the sorting process. The most significant difference is about the structure of the graph, the topological sort applied on. It must be an undirected graph, which will change the way of sorting. The sorting process on customized topological sort starts from initializing the frontier with the node –named source node-. Then we start getting a group ID for each node in the frontier. After that for all the edges of the nodes at the current frontier, we add the destination to be visited in the next frontier list. When done we will swap the fronter with the next fronter contents, make the fronter empty and the group ID is incremented by 1. We do that until we iterate over all the nodes in the graph given.

To handle undirected graphs regarding to the group ID assigning, if a node has already been visited in a previous iteration (i.e., it was in the "frontier" array in a previous iteration), the algorithm skips it to prioritize the first checking of each node. The checking process is done on the group ID, if it has not been assigned a group ID yet it will be visited in the current iteration. In this way we deal with the undirected graph in topological sort.

We will clarify how the algorithm works in the following example.

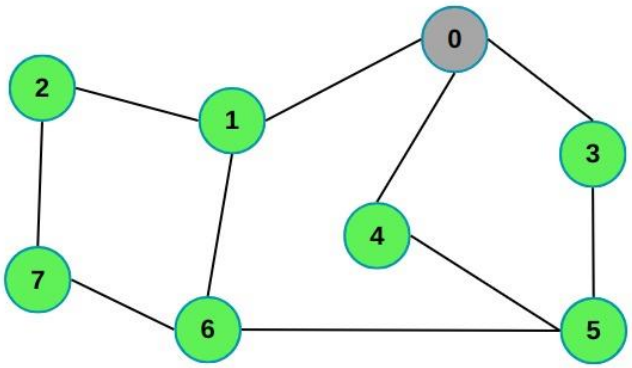
1.3.1 Customized Topological Sort's Example:

This graph will firstly be converted on written way contains the input data to our code.

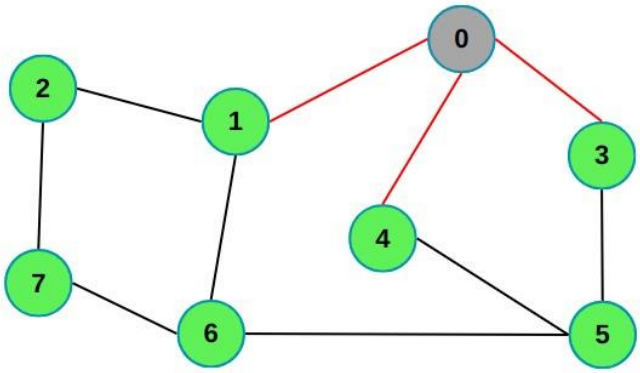


- Number of nodes in the graph: 8
- Graph Nodes will contain each node information, the starting index in Edge list and the number of edges it has.
  = {Starting index, Number of edges}
  = {{0,3}, {3,3}, {6,2}, {8,2}, {10,2}, {12,3}, {15,3}, {18,2}}
- Starting node index= 0
- Size of the Edge List which contains the destination Node for this edge = 20
  Edge List = {1, 3, 4, 0, 6, 2, 1, 7, 0, 5, 5, 0, 3, 4, 6, 1, 5, 7, 2, 6}

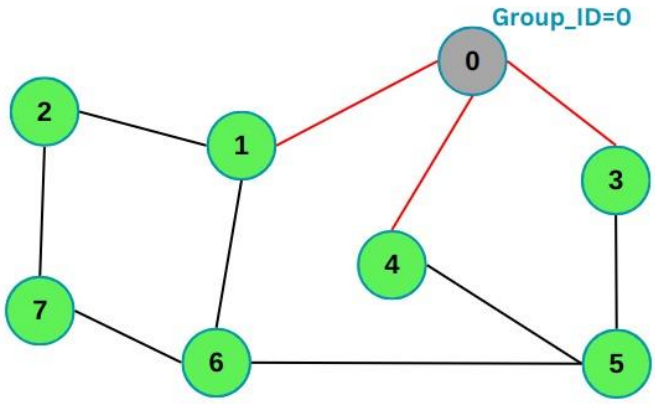
1) Starting with the source node, whose index 0, based on the input data.



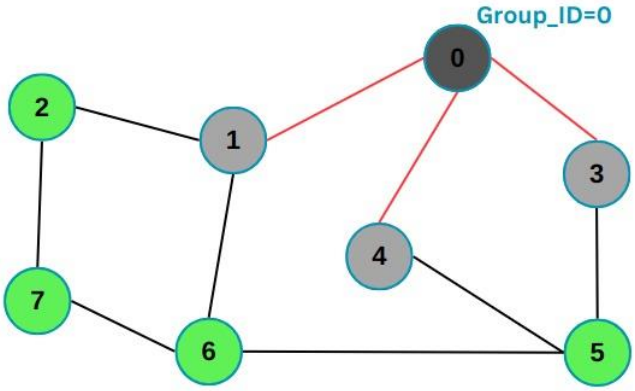
2) Index 0 in graph Nodes array contains {0,3}, which indicates that this node has 3 edges with start position 0 in edge list.



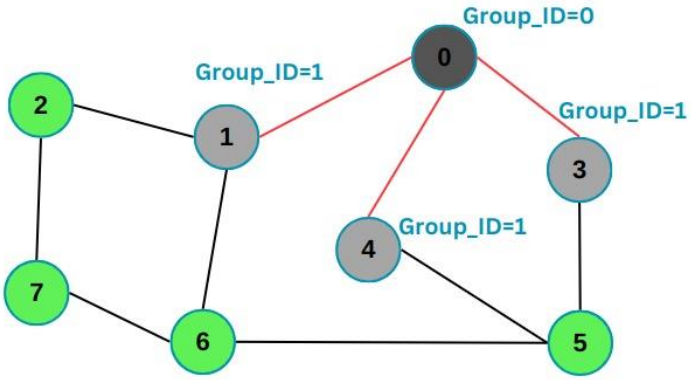
3) Now, we add node 0 to the frontier list, then give it a group ID.



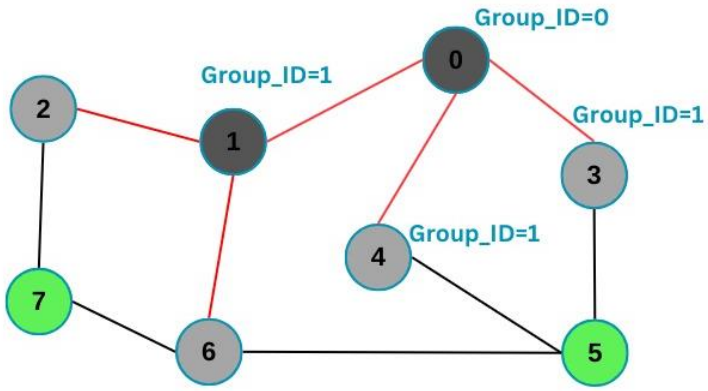
4) Add the destination nodes to be visited in next frontier (1,4,3)



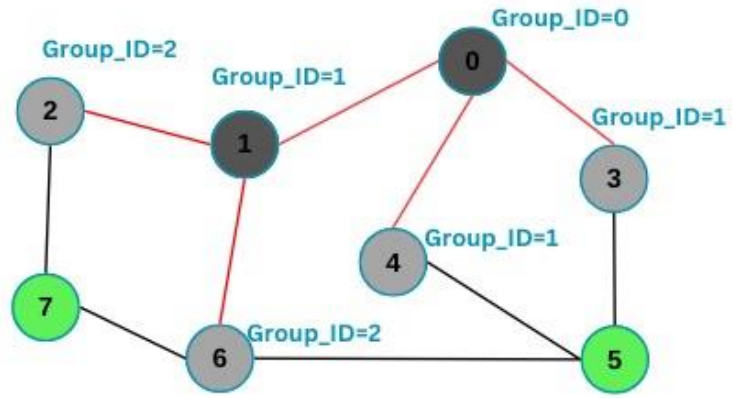
5) Swap the frontier with next fronter >> and give them group ID. (Group_ID=1)



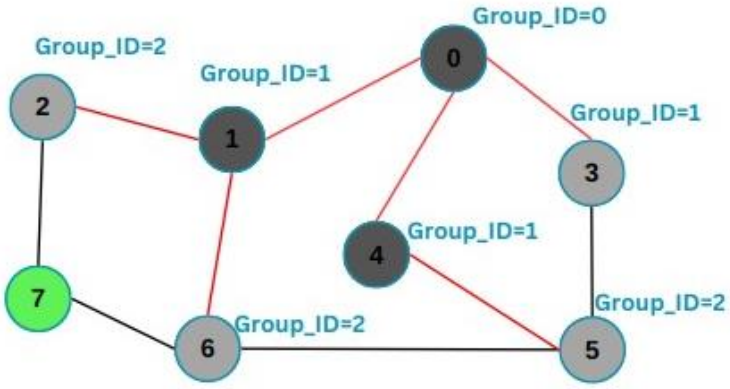
6) Start from the first node in the frontier (Node 1) and add its destination to be visited in the next frontier (2,6). (0 already visited we will skip it)



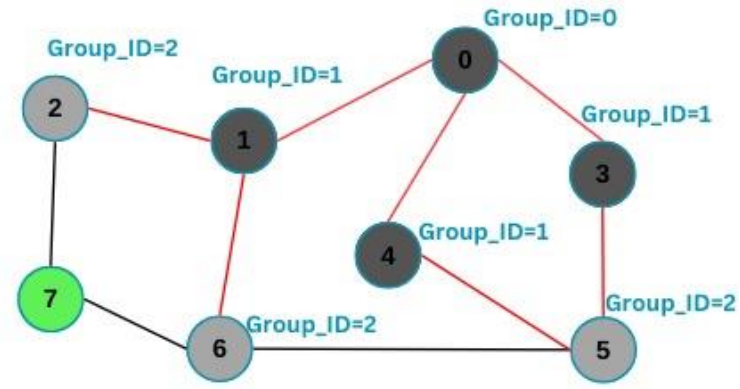
7) Give the nodes group ID=2.



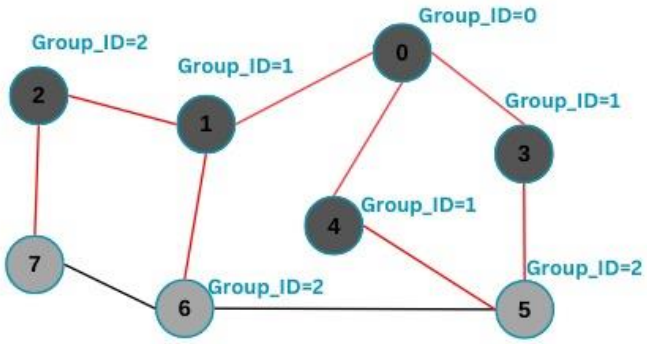
8) Start with the second node in the frontier (Node 4) and add its destination to be visited in the next frontier (Node 5). (0 already visited skip it)



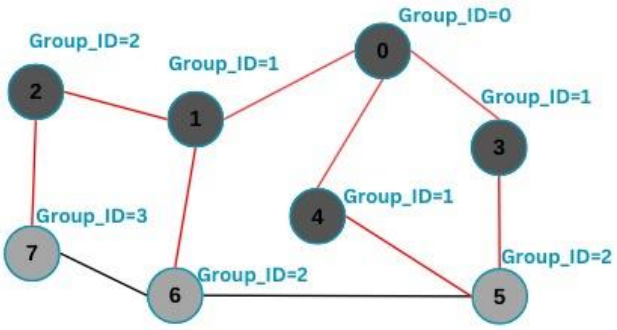
9) Now, we start with node 3, and note that nodes 5 and 0 have group ID so they ignored.



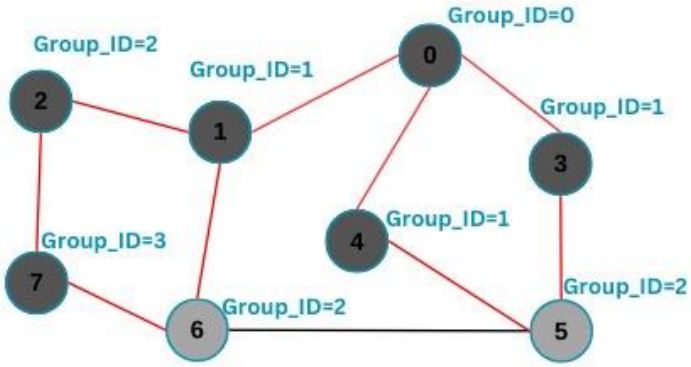
10) Swap the frontier with next frontier and start from the first one (node 2) And add its destination to be visited in the next frontier (only node 7). (1 is already visited skip it)



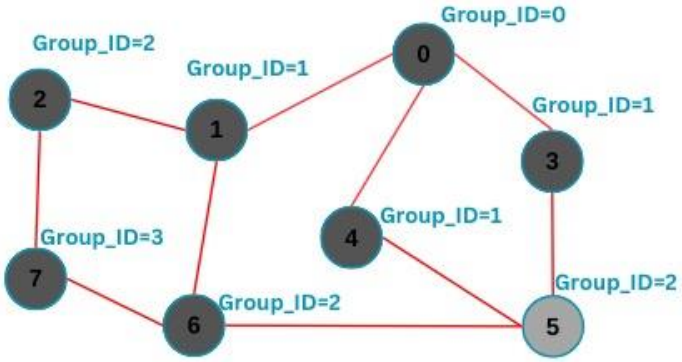
11) Give the node (7) group ID.



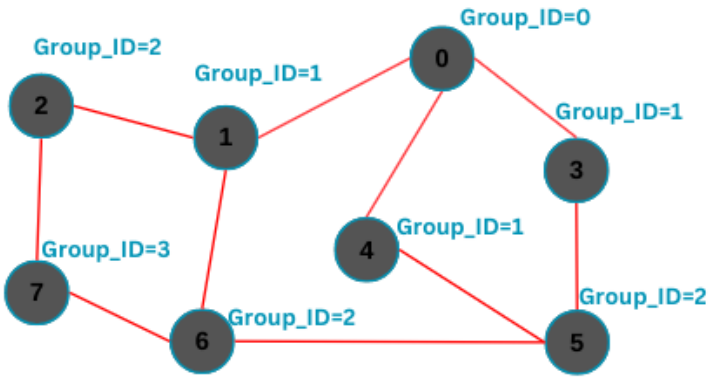
12) Swap, then start from node 7. No destination to add to the next frontier. (Node 6 and 2 already visited skip them)



13) Now, node 6, no destination to add to the next frontier. (1,7 and 5 are already visited skip them).



14) Node 5, no destination to add to the next fronter. (3,4 and 6 are already visited skip them)



1.4 System Configuration

In our project we used a dual-boot system to run serial and parallel codes. This was providing an environment that has direct access to system resources and hardware capabilities and enable a parallel programming technique like openMP which we used.

2. Implementation

In this section we will talk about how we implement the customized topological sort using openMP in 3 versions. The function of all versions will start from a given node (source node) and assigns group IDs to each node based on their connectivity.
The parameters for the function in each version include:

- graph_nodes: An array containing the nodes in the graph.
- graph_edges: An array representing the edges between nodes.
- n: The total number of nodes in the graph.
- source: The starting node.

These parameters are used to initialize and define the graph structure.
By using OpenMP directives, we parallelize the process, distributing the workload across multiple threads. This allows for concurrent execution of tasks and can significantly improve the performance of the algorithm. At the beginning of each version, we set up the necessary variables. The two vectors frontier and next frontier represent the current and next layers of nodes to be processed.

To compute the execution time, we measure the execution time multiple times and take the average. We do this to provide a more reliable estimate of the code's performance. We measures the execution time using **omp_get_wtime()** before and after the while loop to compute only the time of the parallel region.

2.1 Serial Code

After allocating memory for groups, we initialize it by -1 using "*memset*". This will indicate that each node does not have a group ID yet.

```
85      while (changed) {
86        changed = 0;
87        for (const auto& node : frontier){
88            if (groups[node.id] == -1)
89                groups[node.id] = group_id;
90            for(int i=node.starting; i<(node.starting + node.no_of_edges); i++){
91                int x = graph_edges[i];
92                Node dest = graph_nodes[x];
93               // printf("node %d \n", dest.id);
94                if(groups[dest.id] == -1){
95                    next_frontier.push_back(graph_nodes[dest.id]);
96                    changed = 1;
97                }
98            }
99        } //end of 1st for
100
101       if(!next_frontier.empty()){
102           group_id+=1;
103           // Copy all elements using the assign member function
104           frontier.assign(next_frontier.begin(), next_frontier.end());
105           next_frontier.clear();
106       }
107   } //end of while
```

The main loop "while" used to ensure that no further changes will do. This will be using "**changed**" Boolean variable. After that nodes will enter a loop that iterates over "*frontier*" vector. It checks the group ID for each node, if it is -1 which indicates the node has not assigned by a group ID yet. If it has not yet been assigned, it will assign by a group ID then find all connected nodes to add to the next frontier vector. For each node in the next frontier, the code checks the assignment in of group ID. If it is -1 it will be added to the next frontier vector and set the Boolean variable to 1 (changes performed).

If the next frontier vector is not empty, that means that still nodes in it do discover in next iterations. Then increments the group ID. After that the value of the "*frontier*" vector is updated with the "*next frontier*" values using "assign" function line 104. Finally, clear the "next frontier" vector to be ready for the next iteration.

2.2 Parallel Code Version 1

(Using **pragma omp single** and **pragma omp task**)

```
83      while (changed) {
84          changed = 0;
85          next_frontier.clear();
86          #pragma omp parallel
87          #pragma omp single
88          {
89              for (int i = 0; i < frontier.size(); i++) {
90                  const auto& node = frontier[i];
91                  #pragma omp task
92                  {
93                      if (groups[node.id] == -1) {
94                          groups[node.id] = group_id;
95                          for (int j = node.starting; j < (node.starting + node.no_of_edges); j++) {
96                              int x = graph_edges[j];
97                              Node dest = graph_nodes[x];
98                              if (groups[dest.id] == -1) {
99                              #pragma omp critical
100                                 {
101                                     next_frontier.push_back(graph_nodes[dest.id]);
102                                     changed = 1;
103                                 }
104                             }
105                         }
106                     }
107                 }
108             }
109         }
110         if (!next_frontier.empty()) {
111             group_id += 1;
112             frontier = std::move(next_frontier);
113         }
114     }
```

In this code we utilize parallelization by using **“pragma omp parallel**” which creates a parallel region that allows the code to executed in parallel by more than one thread. About using “**pragma omp single**” we ensure that only one thread can execute this portion of code. In our code we used it while creating tasks to ensure that no redundant tasks were created.

Using **“omp task**” allows iterations to be executed concurrently by different threads, which produces more utilization on computational resources. We used this directive on a loop that iterates over frontier vector. Then each task will be for certain iteration in the loop.

To avoid data race the code uses **“pragma omp critical**” when adding nodes to the next frontier vector. This is important to prevent any incorrect data from appearing when multiple threads try to modify the same vector.

2.3 Parallel Code Version 2

Parallel code using “**pragma omp for**”

```
85      while (changed) {
86          changed = 0;
87          #pragma omp parallel shared(frontier, next_frontier, groups, changed)
88          {
89              #pragma omp for
90              for (int i = 0; i < frontier.size(); i++) {
91                  const auto& node = frontier[i];
92                  if (groups[node.id] == -1) {
93                      groups[node.id] = group_id;
94                  }
95                  #pragma omp parallel for
96                  for (int j = node.starting; j < (node.starting + node.no_of_edges); j++) {
97                      int x = graph_edges[j];
98                      Node dest = graph_nodes[x];
99                      if (groups[dest.id] == -1) {
100                         #pragma omp critical
101                         {
102                             next_frontier.push_back(graph_nodes[dest.id]);
103                             changed = 1;
104                         }
105                     }
106                 }
107             }
108         }
109         if (!next_frontier.empty()) {
110             group_id += 1;
111             frontier = std::move(next_frontier);
112             next_frontier.clear();
113         }
114     }
115
```

In this version we firstly used “**pragma omp parallel shared**” which defines the variables as shared among all threads.

Then about the most important changes which done with using “**pragma omp for**”. The use of this directive was due to parallelizing the loop that iterates over frontier vector. This will divide the loops between threads to execute in parallel.

After that each thread will execute the assigned iterations independently. Which causes more efficient execution time.

Finally to avoid data race the code uses “**pragma omp critical**” when adding nodes to the next frontier vector. This is important to prevent any incorrect data from appearing when multiple threads try to modify the same vector.

2.4 Parallel Code Version 3

In this version we try to increase the performance by utilizing the concept of private definition.

```
85    while (changed) {
86        changed = 0;
87        #pragma omp parallel shared(frontier, next_frontier, groups, changed)
88        {
89            std::vector<Node> private_next_frontier;
90            #pragma omp for
91            for (int i = 0; i < frontier.size(); i++) {
92                const auto& node = frontier[i];
93                if (groups[node.id] == -1) {
94                    groups[node.id] = group_id;
95                }
96                for (int j = node.starting; j < (node.starting + node.no_of_edges); j++) {
97                    int x = graph_edges[j];
98                    Node dest = graph_nodes[x];
99                    if (groups[dest.id] == -1) {
100                       private_next_frontier.push_back(graph_nodes[dest.id]);
101                       changed = 1;
102                   }
103               }
104           }
105           #pragma omp critical
106           {
107               next_frontier.insert(next_frontier.end(), private_next_frontier.begin(), private_next_frontier.end());
108           }
109       }
110
111       if (!next_frontier.empty()) {
112           group_id += 1;
113           frontier = std::move(next_frontier);
114           next_frontier.clear();
115       }
116   }
```

By defining a new variable “*private_next_frontier*” using a private directive, that will create a private copy for each thread in the parallel region. This will be alternative for using a critical section and put a shared “*next_frontier*” inside it, now each thread will have its own version.

The importance of this will appear clearly in the need of synchronization which will not appear in this way. Then each thread can modify and write on its own variable.

About “**pragma parallel for**” it used to parallelize the iterations over “*frontie*r” vector.

Finally, we will need a critical section. The critical section will be important in the process of merging the “*private_next_frontier*” for each thread. So that the “*next_frontier”* will have a correct and consistent value, with avoidance of any race condition.

3.Evaluations and Results

In this section, we present the evaluations and results for our work. We compared the results of the parallel codes with serial code to assess their effectiveness in improving efficiency and performance.

3.1 Checking the parallel codes

To compare the output obtained from different codes, we used 'diff' command which allows us to effectively compare two files and highlight the differences between them. By executing this command on the output files generated by the serial code and the three versions of the parallel code, we were able to discern any disparities in the results.
Upon executing the 'diff' command to compare the output files (serial code and parallel code), we found no differences between them. This outcome implies that the parallel code results are correct.

- Comparing version one with serial code:

```
~/just_pc2023_g1/Project$ cd Results/
~/just_pc2023_g1/Project/Results$ diff testingVersion1/graph_8nodes.txt testingSerialCode/graph_8nodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion1/graph_4096nodes.txt testingSerialCode/graph_4096nodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion1/graph_65Knodes.txt testingSerialCode/graph_65Knodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion1/graph_1Mnodes.txt testingSerialCode/graph_1Mnodes.txt
~/just_pc2023_g1/Project/Results$
```

- Comparing version two with serial code:

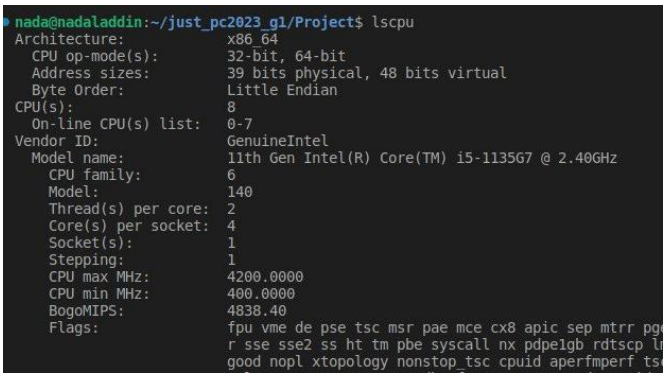
```
:~/just_pc2023_g1/Project/Results$ diff testingVersion2/graph_8nodes.txt testingSerialCode/graph_8nodes.txt
:~/just_pc2023_g1/Project/Results$ diff testingVersion2/graph_4096nodes.txt testingSerialCode/graph_4096nodes.txt
:~/just_pc2023_g1/Project/Results$ diff testingVersion2/graph_65Knodes.txt testingSerialCode/graph_65Knodes.txt
:~/just_pc2023_g1/Project/Results$ diff testingVersion2/graph_1Mnodes.txt testingSerialCode/graph_1Mnodes.txt
:~/just_pc2023_g1/Project/Results$
```

- Comparing version three with serial code:

```
~/just_pc2023_g1/Project/Results$ diff testingVersion3/graph_8nodes.txt testingSerialCode/graph_8nodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion3/graph_4096nodes.txt testingSerialCode/graph_4096nodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion3/graph_65Knodes.txt testingSerialCode/graph_65Knodes.txt
~/just_pc2023_g1/Project/Results$ diff testingVersion3/graph_1Mnodes.txt testingSerialCode/graph_1Mnodes.txt
~/just_pc2023_g1/Project/Results$
```

3.2 Choosing number of threads

This configuration section will aim to show an overview of our hardware specifications of the system which we used in our project. This is done using “**lscpu**” command line. Which has been clarified in the screenshot provided. Our system architecture was x86_64, with 4 cores each has 2 threads (in total 8 threads).



Our system’s architecture has 8 threads. Also we used “**omp_get_max_threads()**” to ensure that in our code before setting the number of threads we passed in the command line.

3.3 Results

- **g++:** This is the command for invoking the GNU C++ compiler.
- **-O3**: This flag tells the compiler to apply aggressive optimizations. Level "3" indicates a high level of optimization.
- **-o output.out**: This flag specifies the output file name. In this case, the executable will be named "output.out".
- **-fopenmp**: This flag enables OpenMP support. OpenMP is a parallel programming model that allows for multi-threading and shared-memory multiprocessing in C++ programs.
- **TopologicalSort.cc**: This is the input C++ source file that you want to compile.

As we mentioned previously, we measure the execution time multiple times and take the average or analyze the distribution of the results.

You can find the results attached in this link:

https://github.com/fghanim/just_pc2023_g1/blob/4f92b2f2b400d0cb3cf69c8041376c581d8c23e4/Project/Execution%20time%20results/After_adding_O3_in_commandline/Comparing_Results_After_Adding_O3_in_commandline.xlsx

We compared the 4 codes together according to the size of the input file. The following table shows the difference between them.

Table 1:

8 Nodes	1 thread	2 Threds	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
Serial Code	0.0000044							
Single/Task	0.0005426	0.000154	0.0002568	0.0002982	0.0003762	0.0006726	0.0008974	0.1390402
Nested parallel	0.0000188	0.000169	0.0002162	0.0002592	0.0003	0.000344	0.000722	0.150573
V3 private next frontier	0.0000204	0.000178	0.0002376	0.0003086	0.0003072	0.000406	0.0008324	0.1396046

Table 2:

4096 Nodes	1 thread	2 Threds	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
Serial Code	0.0024498							
Single/Task	0.0022366	0.0110596	0.026779	0.0396954	0.0444862	0.0684848	0.1023332	0.1939
Nested parallel	0.0031848	0.0086506	0.0096896	0.0112388	0.0129114	0.0223956	0.025041	0.31278
V3 private next frontier	0.002721	0.0046958	0.0071212	0.0045554	0.0119344	0.0394688	0.033231	0.10855

Table 3:

65K Nodes	1 thread	2 Threds	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
Serial Code	0.0173598							
Single/Task	0.0128126	0.0832076	0.1607744	0.1925556	0.2315436	0.2800954	0.3197554	0.3880874
Nested parallel	0.029443	0.0254682	0.0415534	0.0563314	0.065875	0.069387	0.0968186	0.1300372
V3 private next frontier	0.027317	0.0218988	0.0197486	0.0204978	0.0289792	0.0394358	0.0686158	0.0119872

Table 4:

1M Nodes	1 thread	2 Threds	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
Serial Code	0.812676							
Single/Task	0.5828514	1.7167442	2.7645896	3.3679872	3.6023494	3.8299808	4.2565724	4.6381782
Nested parallel	0.9995308	0.914532	0.9891176	1.074198	1.0868374	1.0835568	1.1098858	1.2083506
Nested parallel (before edit)	5.1293794	2.851527	1.9387626	1.7125792	1.6161514	1.5325866	1.5044362	1.5033376
V3 private next frontier	0.9044114	0.6215666	0.516063	0.4604916	0.4556392	0.4106954	0.3554076	0.3432498

3.4 Results' Discussion

Using "pragma_omp_single" and "pragma_omp_tasks" First Version:

"pragma_omp_single" and "pragma_omp_tasks" directives have more than one problem while using, so we must be aware when using it. It produces an additional overhead for managing the parallel environment, with a small amount of work or a limited number of tasks the overhead of creating and managing tasks will be more than the benefits of parallel using, and the synchronization overhead which includes the waiting for dependences and merging the results of the tasks.

In this version we thought about our algorithm structure and how it works. Then we think if we use this way to implement the parallel code, that will give us good results. Unfortunately, the overhead which we mentioned has been appeared clearly and gave us bad results when using more than one thread.

Just in a case of using one thread in this parallel code the performance appears with **speed up = 1.0953** in input size of (4096). Also, in data size of (65K) **speed up = 1.3549.**

That was because the single and task directives provide the compiler with additional information about parallelism. This information can enable the compiler to apply optimizations, such as loop unrolling or instruction scheduling, that can improve performance even when running on a single thread.

Using "#pragma omp parallel for" Second Version:

Following the initial results of the first version, we tried to rework the algorithm by employing different directives. We utilized "#pragma omp parallel" to distribute the loop among the threads. As we discussed previously, our algorithm contains a loop that iterates over frontier vectors, making it suitable candidates for parallelization. So, we tried to test this way and add 2 "#pragma omp parallel" (nested) as shown previously.

Subsequently, we evaluated the performance of the second version with the one million input file as shown on table 4, we can see that the second "#pragma omp parallel" increased the overhead too much. To address this issue, we remove line 95 from the second version.

Also, using the critical section to avoid the race condition they introduced overhead due to synchronization among the threads. We noticed that the second version did not yield desired improvements in the code, which led us to develop the third version.

Adding private next frontier instead of using critical section Third Version:

What happened in the second version was the starting point of thinking in the third version. In this version we tried to focus on the reason which led to bad results in the last version.

The most significant point we focus on is the critical section part. So, we try to modify the code by declaring a new private variable “private_next_frontier”. In this way each thread can modify its own “next_frontier” vector.

That leads to good results which appear clearly in the input data with size (65K) with **speed up was 1.44819** using 8 threads. Also, in input data with size (1M) the **speed up was 2.3675** using 8 threads**.**

4. Group work

In our project we work as a group utilizing the development of technology and the appearance of tools and platforms which help us to facilitate efficient collaboration. **GitHub** has been used during all the previous periods. That allows our group to work simultaneously on the same project files.
That affects effectively with make changes and perform modifications. Also, that was important in tracking all files history by checking commits. This appears significantly in the process of improvement which is done in topological sort code. While trying to improve performance using more than one directive from omp libraries. Moreover, we used **OneDrive** to share excel and word files. Excel files contained an execution time result associated for all versions with variation in number of threads used.

5. Conclusion

In conclusion, we learn how to write a code using openMP just by adding a few directives and instructions to the code. Implementing algorithms using openMP to increase the performance of the code by allowing to write parallel programs using different directives.

In this project, we showed what is the topological sort and how it works. Then, illustrate what is the main difference between the topological sort and the customized topological sort. And how we implemented the customized topological sort using openMP.
As we talked about in the previous sections, there is a different way to write the code using openMP, and each way has its advantages and disadvantages.

This project has valuable knowledge. We tried to do our best efforts to achieve a good performance.

Note: We evaluated our work at first without using (-O3) causing us to reevaluate the code. We uploaded the results before and after adding (-O3) to the command line to GitHub.