OpenMp

Topic

2023

Graph Groups Algorithm

By

Heba Aljedayeh

Razan Hayajneh

supervised by

Dr.Fadi Ghanim

Table Of Content

List of Figure

Chapter 1

Introduction algorithm with pseudo code

1.1 Introduction

In this project, we aim to parallelize the Group Graph algorithm using OpenMP. The Group Graph algorithm is a graph-based algorithm that analyzes relationships and connections within a group of entities. By parallelizing this algorithm, we can leverage the power of multiple threads to expedite the processing and analysis of large-scale group graphs.

OpenMP is a popular shared-memory parallel programming model that provides directives and libraries for developing parallel applications in C, C++, and Fortran. It allows us to exploit the multi-core architecture of modern CPUs and distribute the workload among multiple threads for improved performance.
It is worth mentioning here that the use of the GPU speeds up the performance of the CPU of the unit, as shown in **[The Figure 1.1]**.

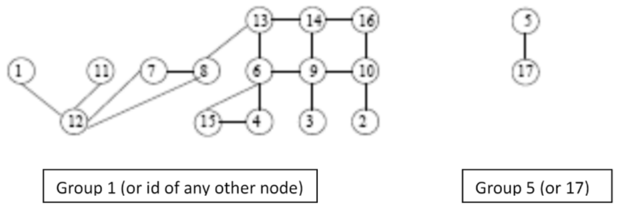

Figure 1.1: Example of Algorithm

1.1.1 Description of the algorithm

lgorithm Name: Graph Grouping: graph G = (V,E), where V = 1, ..., n and —E— = m, find all groups of nodes reachable from any node within same group.

Input: GraphNodes[i].noofedges = no. of edges adjacent to node i GraphNodes[i].starting = start location in Edges, where the list of edges outgoing from vertex i begins. Edge[j] = the destination Node for this edge. Source is obtained from the source GraphNodes covering this node

Output: group[i] = ID of the start node of the group to which node i belongs.

1.2 pseudo code

The provided algorithm aims to group nodes in a graph. It begins by creating arrays to store information about nodes and edges. The algorithm iterates through the nodes, selecting a node randomly that does not have a group ID assigned. It then marks that node as its own group and proceeds to traverse its edges. If a neighboring node does not have a group ID assigned, it assigns the group ID of the source node to that neighbor. If the neighbor already has a group ID, it updates the group ID of the source node to match the group ID of the neighbor. This process continues until all nodes are grouped. The algorithm utilizes parallelism to optimize the grouping process.

```
# Create array of nodes and array of edges
GraphNodes = []
Edges = []
//Initialize an integer array "group"
of size "n" with all elements as "-1"
for i = 0 to n-1
    group[i] = -1
// Upper loop to group nodes and Start a while loop
with condition "!done".
done = false
while not done do
    done = true
    source=-1
//pick a node with no group ID at random
    for i = 0 to n do parallel
        if group[i] == -1 then
            done = false
            source=i
            break
        end if
   end loop
    // If all nodes are already grouped, exit
    if source ==-1
    break
    end if
    // Mark the source node as its own group
    group[source]=source
      for j = GraphNodes[source].starting to GraphNodes[source].starting
            dest = Edges[j][1]
            if  group[dest] == -1 then
                group[dest] = source
            end if
                     else
           group[source]=group[dest]
           end ifelse
        end loop end while
```

1.3 Serial Code in algorithm

The algorithm aims to group nodes in a graph. It follows these steps: If you want to go to the file in which the code is displayed, click here . https://github.com/fghanim/just_pc2023_g5/blob/df929fc52447aa4ff861272042b48139477d0789/Project/graph_groups.h

1.3.1 Performance for Serial code

sample file :graph65536.txt , result algorithm :21126 and time Cpu 0.400572.
sample large file:Group1MW6.txt , result algorithm :21126 and time Cpu 0.400572.

Chapter 2

Parallel Code

2.1 Device Specifications

In general, increasing processing power and available memory can have a positive impact on the performance of algorithms in OpenMP parallel programs. It allows for better task distribution and improves execution speed. On the other hand, when device specifications are limited, it can restrict the algorithm's ability to exploit parallelism and achieve optimal acceleration.

- Processor 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz
- Installed RAM 16.0 GB (15.7 GB usable).
- System type 64-bit operating system, x64-based processor.
- Use virtual machine and install ubuntu 22.04.2

2.2 Challenges

The challenges we faced in converting the code to Parallel .challenges in load balancing on threads include determining the optimal number of threads, ensuring data consistency and integrity, addressing load imbalance, and maintaining reproducible results. It requires careful consideration of the workload, data dependencies, synchronization mechanisms, and balancing techniques to achieve efficient and reliable parallel execution.

2.3 version 1 for parallel code

The parallelization is achieved using OpenMP directives. The

`omp_set_num_threads`

The code utilizes various OpenMP directives, such as omp atomic write, **omp atomic**, and **omp parallel for**, to enable concurrent execution and synchronization between threads. These directives help improve the performance of the algorithm by leveraging parallel processing capabilities. At the end of the code, the execution time is measured using the

`omp_get_wtime`

In summary, this code demonstrates a parallel implementation of a graph grouping algorithm using OpenMP directives (# **pragma omp parallel** and # **pragma omp for**)to improve performance by leveraging parallel processing capabilities. If you want the code in detail, here is the **link to access** it .
https://github.com/fghanim/just_pc2023_g5/blob/df929fc52447aa4ff861272042b48139477d0789/Project/omp_graph_groups.h

2.3.1 Performance for version 1

Sample file : Group65536.txt number of thread :3

Solution method	Performance for second
serial code	0.2456
use omp for	0.90212
use omp for schedule (static)	0.9073
use omp for schedule (dynamic ,10)	9.053
use omp for schedule (guided)	1.08922
for parallel Atomic	0.0017
parallel with schedule (dynamic)	0.0017

Table 2.1: Performance for version 1 file Group65536.txt

Sample file : Group1MW6.txt number of thread :3

Solution method	Performance for second
serial code	287.076
use omp for	117.02
use omp for schedule (static)	0.9073
for parallel Atomic	0.0256
parallel with schedule (dynamic)	0.032726

Table 2.2: Performance for version 1 file Group1MW6.txt

2.4 version 2 for parallel code

To ensure thread safety and avoid data races, locks are used in this implementation. An **OpenMP lock (omplockt)** is declared and initialized using ompinitlock. The lock is acquired using **omp set**

lock before accessing and modifying shared variables, and it is released using **omp unset lock** after the critical section is executed.

The parallelization is achieved using OpenMP directives. The **omp set num threads** function sets the number of threads to be used for parallel execution. **The omp parallel for and omp parallel directives** are used to distribute the workload among the available threads and ensure parallel execution within the loop.

At the end of the code, the execution time is measured using the **omp get wtime** function, and the results are printed, including the execution time and the number of groups formed. If you want the code in detail, here is the **link to access** it .
https://github.com/fghanim/just_pc2023_g5/blob/df929fc52447aa4ff861272042b48139477d0789/Project/omp_graph_groups.h

Sample file : Group65536.txt number of thread :3

Solution method	Performance for second
serial code	0.2456
use omp for	0.90212
use for or parallel with lock	0.003123

Table 2.3: Performance for version 2 file Group65536.txt

Sample file : Group1MW6.txt number of thread :3

Solution method	Performance for second
serial code	287.076
use omp for	117.02
use for or parallel with lock	0.06223

Table 2.4: Performance for version 2 fileGroup1MW6.txt

To review the results and performance clearly, here is the excel sheet file at this link.
https://github.com/fghanim/just_pc2023_g5/blob/main/Project/performancedata.xlsx

2.5 Conclusion

Both approaches(versions) aim to improve performance by leveraging parallelism. The first approach uses atomic operations to handle concurrent updates, while the second approach uses locks to ensure exclusive access to shared variables. The choice between the two depends on the specific requirements of the algorithm and the nature of the shared data. But here, in comparison with the results, we chose the first approach.