CS406 - Project Final Report

Erkut Gürol - 23605
Burak Sekili - 24944
Osman Serhan Silahyürekli - 25205
Halil Tuvan Gezer - 25658

1 Introduction

The project of CS406/531, Parallel Computing, course, included the design of an algorithm which takes an undirected graph G=(V, E) as the input, inside which |V| vertices and |E| edges exist and finds every single cycle of a given length, denoted as k whose value ranges between 2 and 6, for every vertex present inside the graph, while recording the number of cycles of length k present for each vertex in the process. At the first section of the project, the graphs, which were presented in the form of text files containing all the edges present between vertices u_i and v_i, were read into CSR data structure for efficient use of memory and sequential algorithm was designed for counting the presence of vertices inside the complete sub-graphs of length k, which were then stored inside of an output text file. In addition to storing the vertex and presence of vertices inside complete sub-graphs pairs, timing of the sequential algorithm was also measured. Following the design and implementation of sequential algorithm, the algorithm was parallelized by using OpenMP pragmas and the timings, along with the speed-up values with respect to the sequential algorithm, were also recorded. At the final phases of the project, the algorithm was implemented in GPU by using CUDA, known as the parallel computing platform of Nvidia, and the performance of GPU algorithm was improved with the simultaneous use of hybrid OpenMP and CUDA parallelism, while the performance of GPU based and hybrid algorithms were measured with respect to time and speed-up parameters in the process.

2 Algorithm

The sequential algorithm, which has been designed for the given problem of finding cycles of length k for every single vertex in a given graph G, follows the procedure of finding total number of complete sub-graphs of length k, along with the present vertices in every single complete sub-graph, inside the graph G and counting the presence of vertices inside the complete graphs in the process. Below, a sample pseudocode is provided for the sequential algorithm designed for the project.

2.1 Sequential Algorithm

Algorithm 1 cycles_3(v)

```
count ← 0
neighbours(v) ← getneighbours(adj, xadj)
for i in neighbours(v2) do
  v2 ← neighbours[i]
  neighboursv2(v2) ← getneighbours(adj, xadj)
  for j in neighboursv2(v2) do
    v3 ← neighboursv2[j]
    if v3 != v1 then
      neighboursv3(v3) ← getneighbours(adj, xadj)
      for k in neighboursv3(v3) do
        v4 ← neighboursv3[k]
        if v4 == v then
          count++
        end if
      end for
    end if
  end for
end for
return count
```

Algorithm 2 cycles_4(v)

```
count ← 0
neighbours(v) ← getneighbours(adj, xadj)
for i in neighbours(v2) do
  v2 ← neighbours[i]
  neighboursv2(v2) ← getneighbours(adj, xadj)
  for j in neighboursv2(v2) do
    v3 ← neighboursv2[j]
    if v3 != v1 then
      neighboursv3(v3) ← getneighbours(adj, xadj)
      for k in neighboursv3(v3) do
        v4 ← neighboursv3[k]
        if v4 != v and v4 != v2 then
          neighboursv4(v4) ← getneighbours(adj, xadj)
          for l in neighboursv4(v4) do
            v5 ← neighboursv4[l]
            if v5 == v then
              count++
            end if
          end for
        end if
      end for
    end if
  end for
end for
return count
```

Algorithm 3 cycles_5(v)

```
count ← 0
neighbours(v) ← getneighbours(adj, xadj)
for i in neighbours(v2) do
  v2 ← neighbours[i]
  neighboursv2(v2) ← getneighbours(adj, xadj)
  for j in neighboursv2(v2) do
    v3 ← neighboursv2[j]
    if v3 != v1 then
      neighboursv3(v3) ← getneighbours(adj, xadj)
      for k in neighboursv3(v3) do
        v4 ← neighboursv3[k]
        if v4 != v and v4 != v2 then
          neighboursv4(v4) ← getneighbours(adj, xadj)
          for l in neighboursv4(v4) do
            v5 ← neighboursv4[l]
            if v5 != v and v5 != v2 and v5 != v3 then
              neighboursv5(v5) ← getneighbours(adj, xadj)
              for m in neighboursv5(v5) do
                v6 ← neighboursv5[m]
                if v6 == v then
                  count++
                end if
              end for
            end if
          end for
        end if
      end for
    end if
  end for
end for
return count
```

Algorithm 4 readFile(filePath, k)

```
xadj ← [ ]
adj ← [ ]
results ← [ ]
(xadj, adj) ← openFile(filePath)
for i in length(xadj) do
  if k == 3 then
    results[i] ← cycles3(i)
    results.txt ← write(i, results[i])
  else if k == 4 then
    results[i] ← cycles4(i)
    results.txt ← write(i, results[i])
  else if k == 5 then
    results[i] ← cycles5(i)
    results.txt ← write(i, results[i])
  else
    print("Wrong k Value!")
  end if
end for
```

2.2 Analysis of the Algorithm

The algorithm, in its essence, is based upon iterative version of depth first search method, which takes the graph G and number k as input and produces a text file for results from an array including the number of cycles for each vertex, with vertices being denoted by the corresponding indices of the array, as the output. readFile function traverses through all the vertices on the graph and tries to find all cycles with length k for the individual vertices by applying iterative version of depth first search to find whether complete sub-graphs of length k exist or not.

Iterative versions of depth first search, which has been implemented with cycles3, cycles4 and cycles5 functions, are used for searching for cycles of length k. The functions take the input vertex as the input and obtains the neighbours of the input vertex. After obtaining the neighbours for the input vertex, the algorithms, for each neighbour, ensures that the neighbour is not equal to the input vertex. Ensuring that the neighbour is not equal to the input, or in more general sense, a previously visited vertex, algorithms repeat the procedure by obtaining the neighbours of the second vertex and for each neighbour, ensures that the neighbour is not equal to previously visited vertices. The procedure is followed recursively

until a graph of length k is constructed and once length value of k is achieved, algorithm checks for whether the obtained neighbour vertex is equal to the input vertex and once the equality of two vertices are ensured, the counter value stored for input vertex is incremented. At the end of the operation, index of the input vertex, in addition to the stored counter value, are written to a text file named results.txt.

The algorithm's complexity is dependent on the number of input vertices and value of k since the algorithm iterates over all vertices inside the graph and for each vertex, constructs k different for loops to check for the neighbours. In worst case of computational complexity, input vertices have *V-1* neighbours at each individual steps of cycles functions, therefore, the upper bound for the computational complexity could be estimated as $O(V^{k+1})$ for the overall algorithm.

3 Parallelization of the Algorithm

Research by Alon, Yuster and Zwick [1] is one sample paper which explores the possibility of using the color coding, a widely known NP-Complete problem, for finding the number of complete sub-graphs with length k in a given graph. The proposed algorithm is based upon finding all colorful paths, meaning that colors of all vertices on the path differ from each other, of length k-1 inside the graph and checking whether cyclic sub-graphs could be obtained through the obtained paths. The proposed study shows that the problem can be solved with polynomial computational complexity when $k \leq 7$. In the case of the proposed algorithm for the project, computational complexity of the exact algorithm is polynomial and the degree of the polynomial bound increases with the increasing values of k. Therefore, the solution provided by the experimenters matches with the proposals and lemmas of Alon, Yuster and Zwick, who have proven the possibility for achieving polynomial computational upper bound for the problem when $k \leq 7$ on a different algorithm.

The sequential algorithm, in order to minimize the communication between threads during the design and implementation process of the parallelized algorithm, was designed with the purpose of making the execution of cycles functions independent from each other for every input vertex. Moreover, during the implementation process of the algorithm, it was observed that the majority of the overhead was due to iteration over each vertex inside the matrix, as

cycles functions iterate over sparse sets when checking for the neighbours of the vertices, and the imbalance between each iteration, as some vertices may have more neighbours compared to other vertices and iteration over those nodes would take more time during the execution. Therefore, during the design process of the parallelized algorithm, dynamic scheduling, after ensuring that best performance in terms of time was obtained when dynamic scheduling was used instead of static or guided scheduling methods, was used for minimizing the load imbalance between each iteration over the input vertices, which was the parallelized section of the algorithm due to iteration over each vertex being the majority of the overhead inside the code block. Since the iteration over each vertex is independent from each other and there is no need for synchronization between threads, reduction methods for minimizing communication and synchronization, which were previously implemented during the design process of the sequential algorithm, were not taken into consideration as major sources of overheads during the parallelization process. To implement the parallelized algorithm, pragma structures of OpenMP was used. Below, a sample pseudocode is provided for the sequential algorithm designed for the project.

Algorithm 5 parallelreadFile(filePath, k)

$xadj \leftarrow [\,]$
$adj \leftarrow [\,]$
$results \leftarrow [\,]$
$(xadj, adj) \leftarrow openFile(filePath)$
$\#\ pragma\ omp\ parallel\ for\ schedule(dynamic,\ num\_threads(nt))$
for $i\ in\ length(xadj)$ **do**
  if $k\ ==\ 3$ **then**
    $results[i] \leftarrow cycles3(i)$
    $results.txt \leftarrow write(i, results[i])$
  else if $k\ ==\ 4$ **then**
    $results[i] \leftarrow cycles4(i)$
    $results.txt \leftarrow write(i, results[i])$
  else if $k\ ==\ 5$ **then**
    $results[i] \leftarrow cycles5(i)$
    $results.txt \leftarrow write(i, results[i])$
  else
    $print("Wrong\ k\ Value!")$
  end if
end for

In theory, the upper bound for computational complexity occurs when all vertices are connected with each other, therefore, all threads inside the parallel region would, theoretically, take the same amount of time to complete their tasks. Moreover, during the theoretical estimation of computational complexity for the parallel algorithm, it was assumed that workload of all threads present inside the parallel region would be equal to each other, meaning that they would operate on equal sized chunks. Therefore, the computational complexity and speed-up of the parallelized algorithm was estimated as $O(V^{k+1}/p\ +1)$ and p respectively, with p denoting the number of processors present during the execution of the algorithm. Finally, the major overall overhead of the algorithm after parallelization was estimated as $O(p)$ due to the independency and requirement for no synchronization between the processes, also leading to more concrete speed-up estimation of $O(pV^{k+1}/(V^{k+1}\ +p))$. Therefore, it was observed that the system is scalable and to achieve constant efficiency, the overall work and the input size should change with the ratio of p'/p, where p' and p denote the number of processors during an execution and number of processors taken into consideration as the reference respectively.

4 Performance of the Algorithm

The designed sequential and parallelized algorithms are tested on input graph samples, given as dblp.txt and amazon.txt, with cycle length k values of 3, 4 and 5 respectively. The correctness of the algorithms were tested by comparing the outputs with sample output texts provided by the instructor. During the execution, performance of the sequential and parallelized algorithms with respect to the time parameter, in addition to the speed-up and efficiency values for the parallel algorithm were recorded. The parallelized algorithm was tested with 1, 2, 4, 8, 16 and 32 threads and it was observed that the speed-up values for the algorithm scaled linearly with respect to the number of threads for increasing computational complexity and denser graph structures, with linear scaling of the speed-up being visibly present up to 32 threads for cycle length of 5. Moreover, it was observed that the efficiency of the algorithm, with the increasing number of processes, decreased with moderate slope, due to the complexity of the overall overhead being significantly low compared to the processed work. However, during testing procedure, it was observed that the efficiency reduction

for increasing values of threads for sparse structures with relatively low computational complexity, such as amazon graph with cycle length 3, followed steeper slope compared to other test cases, showing that the algorithm may prove to be memory bound, instead of computation bound, for sparse graph structures for low computational complexity. Below, tables for performance of the algorithm with respect to the time performance, speed-up of the algorithm and efficiency of the algorithm are presented.

	Sequential	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
dblp 3	1.97	1.95	1.02	0.55	0.3	0.178	0.118
dblp 4	59.2	59.73	29.96	15.16	7.58	3.81	1.94
amazon 3	0.891	0.884	0.495	0.2796	0.187	0.1463	0.1284
amazon 4	5.998	6.086	3.1293	1.5637	0.8249	0.4492	0.2604
amazon 5	52.78	52.47	26.29	13.17	6.64	3.34	1.71

Table 1: Input Graph and k and Number of Threads vs. Time(s)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
dblp 3	1.0102	1.911765	3.545455	6.5	10.95506	16.52542373
dblp 4	0.9911	1.9937	3.939974	7.879947	15.67717	30.78865979
amazon 3	1.00791855	1.785859	3.16166	4.727273	6.042379	6.88735202
amazon 4	0.98554	1.944844	3.892051	7.377864	13.54853	23.37173579
amazon 5	1.00590814	1.995816	3.984055	7.902108	15.70958	30.68421053

Table 2: Input Graph and k and Number of Threads vs. Speed-Up

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
dblp 3	1.0102	0.955882	0.886364	0.8125	0.684691	0.516419492
dblp 4	0.9911	0.996829	0.984993	0.984993	0.979823	0.962145619
amazon 3	1.00791855	0.892929	0.790415	0.590909	0.377649	0.215147975
amazon 4	0.98554	0.972422	0.973013	0.922233	0.846783	0.730366743
amazon 5	1.00590814	0.997908	0.996014	0.987764	0.981849	0.958881579

Table 3: Input Graph and k and Number of Threads vs. Efficiency

5 GPU Based Implementation

Following the design and implementation of the sequential and parallelized, through the use of OpenMP pragmas, algorithms for the given problem of finding cycles of length k for every single vertex in a given graph G, the project was continued with the implementation of the system through use of CUDA. At the host function of the algorithm, required number of blocks were calculated by using the values for number of threads per block, which were selected as 256 in the case of the project, and the number of rows, or columns, present in the input matrix. Following the calculation of number of blocks and allocation of memory for device and host variables, one of the device kernels, individually designed for calculating cycles of length k for every possible vertex, were called from the host function by taking input k value. Inside the kernels designed for calculating cycles of length k for individual vertices, every single vertex was assigned to their corresponding GPU threads, determined by the global identities of the threads. Following the assignment of input vertices to threads, each thread calculated their corresponding count values and the calculated count values were retrieved back from device memory to host memory to be recorded in the file created for storing the obtained results. Below, sample pseudocodes for GPU based device kernels are provided.

Following the design and implementation procedures, the algorithm for GPU based parallelization for calculation of number of cycles per vertex was tested on input matrices dblp and amazon with cycle lengths of 3, 4 and 5 respectively. During the testing process, it was observed that GPU based algorithm's scalability with respect to the speed-up compared to the sequential algorithm decreased significantly with increasing graph density and com-

Algorithm 6 d_cycles3(G)

```
count ← 0
v ← threadIdx.x + blockDim.x x blockIdx.x
if v < size(G) then
  neighbours(v) ← getneighbours(adj, xadj)
  for i in neighbours(v2) do
    v2 ← neighbours[i]
    neighboursv2(v2) ← getneighbours(adj, xadj)
    for j in neighboursv2(v2) do
      v3 ← neighboursv2[j]
      if v3 != v1 then
        neighboursv3(v3) ← getneighbours(adj, xadj)
        for k in neighboursv3(v3) do
          v4 ← neighboursv3[k]
          if v4 == v then
            count++
          end if
        end for
      end if
    end for
  end for
end if
return count
```

Algorithm 7 d_cycles4(G)

```
count ← 0
v ← threadIdx.x + blockDim.x x blockIdx.x
if v < size(G) then
  neighbours(v) ← getneighbours(adj, xadj)
  for i in neighbours(v2) do
    v2 ← neighbours[i]
    neighboursv2(v2) ← getneighbours(adj, xadj)
    for j in neighboursv2(v2) do
      v3 ← neighboursv2[j]
      if v3 != v1 then
        neighboursv3(v3) ← getneighbours(adj, xadj)
        for k in neighboursv3(v3) do
          v4 ← neighboursv3[k]
          if v4 != v and v4 != v2 then
            neighboursv4(v4) ← getneighbours(adj, xadj)
            for l in neighboursv4(v4) do
              v5 ← neighboursv4[l]
              if v5 == v then
                count++
              end if
            end for
          end if
        end for
      end if
    end for
  end for
end if
return count
```

Algorithm 8 d_cycles5(G)

```
count ← 0
v ← threadIdx.x + blockDim.x x blockIdx.x
if v < size(G) then
  neighbours(v) ← getneighbours(adj, xadj)
  for i in neighbours(v2) do
    v2 ← neighbours[i]
    neighboursv2(v2) ← getneighbours(adj, xadj)
    for j in neighboursv2(v2) do
      v3 ← neighboursv2[j]
      if v3 != v1 then
        neighboursv3(v3) ← getneighbours(adj, xadj)
        for k in neighboursv3(v3) do
          v4 ← neighboursv3[k]
          if v4 != v and v4 != v2 then
            neighboursv4(v4) ← getneighbours(adj, xadj)
            for l in neighboursv4(v4) do
              v5 ← neighboursv4[l]
              if v5 != v and v5 != v2 and v5 != v3 then
                neighboursv5(v5) ← getneighbours(adj, xadj)
                for m in neighboursv5(v5) do
                  v6 ← neighboursv5[m]
                  if v6 == v then
                    count++
                  end if
                end for
              end if
            end for
          end if
        end for
      end if
    end for
  end for
end if
return count
```

putation complexity, while performing significantly better in terms of speed-up parameter compared to CPU level parallelized algorithms for coarse input structures and relatively low computational complexity, obtaining speed-up values greater than the speed-up values obtained through the use of 32 CPU threads for amazon graph with cycle length of 3. On the other hand, for the cases of dense input structures and higher computational complexity, determined through the input cycle length, GPU based algorithm achieved speed-up values comparable to CPU level implementation with 4, and even 2, threads, for the sample case of dblp graph with cycle lengths of 3 and 4 respectively due to increased amount of computation and increased memory access requirement to the global memory inside the kernel, resulting in significant overhead in the process. Hence, by making analysis of the obtained performances for CPU and GPU based algorithms, it was concluded that GPU based algorithm proved to be more efficient for solution of coarse structures while CPU based algorithm proved to be more efficient for solution of denser structures requiring more computation in average for every individual vertex.

To mitigate the overheads resulting from increased global memory accesses and amount of computations with GPU based implementation, an initial hybrid solution, utilizing both GPU and CPU, was proposed to improve the performance of the algorithm with respect to the speed-up parameter. The proposed algorithm divided the graph into two even sections, with length of $number\ of\ rows\ /\ 2$ and the first section of the graph was solved with utilization of GPU while the second section of the graph was solved with the utilization of CPU with multiple threads. The proposed method, for denser graph structures such as dblp, provided significant improvement on speed-up metric compared to the implementation of the algorithm utilizing only GPU for parallelization, with solution time for dblp graph with cycle length of 4 threads changing from approximately 40 to 25 s in the process, while the change in the performance of the algorithm for coarse structures with low computational complexity remained insignificant.

To further improve the performance of the GPU based algorithm, a hybrid solution utilizing both GPU and CPU was proposed for dividing the input data to chunks and finding number of cycles for coarse sections of the graph with GPU while finding number of cycles for dense sections of the graph with CPU based parallelized implementation. At the initial phases of the algorithm, the input data, given in the form of number of rows and row identities,

was partitioned into chunks of size 2048 and the total number of neighbouring vertices at each chunk were calculated and recorded. At the following section, the calculated chunks were sorted with increasing total number of vertices and the identities of starting vertices for each chunk were recorded on a vector data structure with the established order. Following the sorting procedure for the chunks, GPU and CPU were utilized in parallel such that GPU level implementation, starting from the chunk with smallest number of total neighbouring vertices, moved towards the chunks with more neighbouring vertices at each step while CPU level implementation started from the chunk with largest number of total neighbouring vertices and moved towards the chunks with less neighbouring vertices. At the end, the results from GPU and CPU level implementations were merged and recorded to the output text file. Compared to GPU based algorithm, which achieved approximately 4, 1.5 and 5.8 times speed-up for dblp and amazon input matrices with cycle lengths of 3, 4 and 5 respectively, the new proposed algorithm achieved respective approximate speed-up values of 7, 5.34 and 7.69, while, as in the case with the previously proposed algorithm, the change in the performance of the algorithm for coarse structures remained insignificant. Below, sample pseudocodes for proposed hybrid solution are provided.

Algorithm 9 populate_jobs(V)

$jobs \leftarrow PartitionChunks(xadj, Chunk\_Size)$
$sort(jobs)$

Algorithm 10 cpu_manager(V , $jobs$)

```
while True do
  j ← jobs.back()
  j ← jobs.pop_back()
  if k == 3 then
    #pragma omp parallel for schedule(dynamic)
    for i in range(j.start , j.end) do
      results[i] ← cycles_3(i)
    end for
  else if k == 4 then
    #pragma omp parallel for schedule(dynamic)
    for i in range(j.start , j.end) do
      results[i] ← cycles_4(i)
    end for
  else if k == 5 then
    #pragma omp parallel for schedule(dynamic)
    for i in range(j.start , j.end) do
      results[i] ← cycles_5(i)
    end for
  end if
end while
```

Algorithm 11 gpu_manager(V , $jobs$)

```
while True do
  j ← jobs.front()
  j ← jobs.pop_front()
  if k == 3 then
    d_cycles3 <<< NO_BLOCKS , NO_THREADS >>> (j)
  else if k == 4 then
    d_cycles4 <<< NO_BLOCKS , NO_THREADS >>> (j)
  else if k == 5 then
    d_cycles5 <<< NO_BLOCKS , NO_THREADS >>> (j)
  end if
  cudaDeviceSynchronize()
end while
```

6 Performance of GPU Based and Hybrid Algorithms

The designed GPU based and hybrid algorithms are tested on input graph samples, given as dblp.txt and amazon.txt, with cycle length k values of 3, 4 and 5 respectively. The correctness of the algorithms were tested by comparing the outputs with sample output texts provided by the instructor. During the execution, performance of the algorithms with respect to the time parameter, in addition to the speed-up values for the algorithms were recorded. During the testing procedure, it was observed that significant improvement with respect to the speed-up parameter was obtained with hybrid implementation with respect to the GPU based algorithm for the dense input graph structures with increasing computation amount. Below, tables for performance of the GPU based and hybrid algorithms with respect to the time performance and speed-up are presented.

	Sequential	GPU	GPU + CPU (Version 1)	GPU + CPU (Final Version)
dblp 3	1.97	0.496	0.343	0.28
dblp 4	59.2	39.556	25.116	11.185
amazon 3	0.891	0.0.659	0.135	0.111
amazon 4	5.998	0.755	0.46	0.75
amazon 5	52.78	9.011	5.05	6.82

Table 4: Input Graph and k and Number of Threads vs. Time(s)

	GPU	GPU + CPU (Version 1)	GPU + CPU (Final Version)
dblp 3	3.931	5.685	6.964
dblp 4	1.51	2.378	5.340
amazon 3	13.414	6.548	7.964
amazon 4	8.061	13.230	8.115
amazon 5	5.823	10.390	7.694

Table 5: Input Graph and k and Number of Threads vs. Speed-Up

7 References

[1] N. Alon, R. Yuster, and U. Zwick, "Color-coding," Journal of the ACM, vol. 42, no. 4, pp. 844–856, 1995.