Politecnico di Torino

Class LM-32 (DM270)
A.y. 2023/2024
July 2024

Parallelizing The Maximum Clique Computation for Multi and Many-core Architectures

Supervisor:
Stefano Quer
Lorenzo Cardone

Candidate:
Salvatore Di Martino

Contents

1 Introduction **5**
- 1.1 Graphs 6
- 1.2 Problems and Algorithms 8
- 1.3 Performance Evaluation 9

2 GPU Architecture **11**
- 2.1 Overview of GPU Architecture 11
  - 2.1.1 GPU Architecture 11
  - 2.1.2 GPU Memory 14

3 Clique Algorithms **17**
- 3.1 Bron-Kerbosch Algorithm 17
- 3.2 The k-core decomposition and ordering 22
- 3.3 Tomita's Pruning Strategy 22
- 3.4 San Segundo: BitBoard MaxClique Algorithm 27
- 3.5 McCreesh: Multithreaded Maximum Clique 29
- 3.6 San Segundo: BBMCPara 32
- 3.7 Tomita's Re-NUMBER algorithm 34
- 3.8 PMaxSAT based Pruning Strategy: MoMC 35
- 3.9 LMC: Large MaxClique 42
- 3.10 Multithreaded LMC 45
- 3.11 Chang: Maximum Clique Branch-Reduce and Bound 45
- 3.12 Van Copernolle: BBMCG 48
- 3.13 Almasri: Parallel MCE on GPUs 48

4 GPU parallel implementation **51**
- 4.1 From MCE to MCP 51
- 4.2 Parallel MCP for large and sparse graphs 52
- 4.3 MCP Solver for large sparse graphs 54
- 4.4 Warp-wise-parallel version 60
- 4.5 Pruning strategies 60

5 Experimental Analisys **63**
5.1 Experimental Setup . . . 63
5.2 Dataset . . . 63
5.3 Parallel CPU implementations . . . 64
5.3.1 BBMCPara . . . 64
5.3.2 LMC . . . 64
5.3.3 MC-BRB . . . 66
5.4 Experimental analisys . . . 66
5.4.1 Random Instances . . . 66
5.4.2 Real-world dataset . . . 71
5.5 Warp-wise Parallel . . . 73

6 Conclusions **77**

7 Acknowledgements **79**

Chapter 1

Introduction

Graphs are discrete mathematical structures designed to describe relationships between elements. Graphs, also known as networks, are tipically used to describe social networks, molecular structure and communication network infrastructure. The graph analisys consist of extracting such information, such as core, clique, and isomorphism. We spend our efforts in studying the maximal cliques which are special kinds of graphs or subgraphs such that every two vertices are connected by an edge. The maximum clique problem has been deeply studied problem, and many algorithms and optimizations able to compute the solution efficiently exist, These approaches also exploit modern parallel architecture such as multi-threaded CPU and GPU. The starting maximum clique algorithm is derived from the Bron-Kerbosch algorithm algorithm employed for enumerating all the maximal cliques in a graph. They have a multitude of applications in many fields, such as social network analysis [14], bioinformatics [9], coding theories [5], and economics [2]. In social network analisys, cliques can represent groups of people in social network analysis such that they know each other, and in economics can give an overview of the behavior of the stock market. They can also represent structure similarities between amino-acids of protein in biochemistry. Graphs of different applications have different topologies. We analyze many of them and analyze the behavior of the algorithm depending on their topology. For instance, social networks are usually sparse they can have little clusters with respect the overall size. This kind of network is often easily solvable by exploiting preprocessing strategies. Hard-to-solve instances include the so-called *association graphs* [10], used to solve another graph-related problem: the maximum common subgraph. This class of graphs are little-medium size with a medium density of about 60%, but the preprocessing procedure is ineffective. Another special kind of network exists, the biological networks, that are overall sparse but locally dense. Some algorithms are specifically designed to handle this kind of graph by exploiting mathematical lemmas, allowing the algorithm to reduce the search

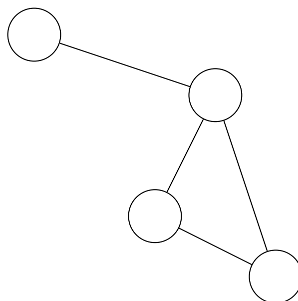


Figure 1.1: An undirected graph of 4 vertices and 4 edges

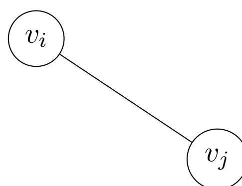


Figure 1.2: An edge between two vertices v_i and v_j.

space. This document is structured as follows. Chapter 1 introduces the basics and notions of graphs. Chapter 2 discusses the studied architectures. Chapter 3 covers the most important state-of-the-art exact maximum clique solvers. Chapter 4 presents our parallel implementation. Finally, Chapter 5 contains the experimental results, Chapter 6 provides the conclusions.

1.1 Graphs

A Graph can be described by a pair of sets $G = (V, E)$ which are respectively the set of *vertices* and the set of *edges* between vertices as shown in Figure 1.1.

Each element of E is a pair of vertices (v_i, v_j) which means that there is an edge that connects v_i to v_j, it is shown in Figure 1.2.

A graph can be directed or undirected, for an undirected graph the edge for each $(v_i, v_j) \in E$ there exist also $(v_j, v_i) \in E$ so if $(v_i, v_j) \in E(G) \iff (v_j, v_i) \in E(G)$. A directed edge can be represented by an arrow starting from vertex v_i to vertex v_j, it is shown in Figure 1.2.

A graph can be labeled, in this case, each vertex brings a label, instead when weighted, each edge has a weight that can be a real number. We focus on Cliques: which are undirected graphs or subgraphs where every two vertices of the graph or subgraphs are connected by an edge.

A clique as shown in Figure 1.4 of size n contains n vertices. It can

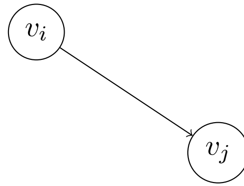


Figure 1.3: Directed edge.

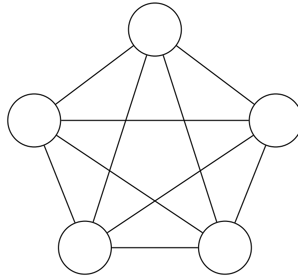


Figure 1.4: A clique.

be maximal, if, no other adjacent vertices can be added to the clique. The biggest clique is called the maximum clique, we refer to that as $\omega(G)$.

The neighborhood of a vertex v defined as $\Gamma(G, v)$ is the set of vertex adjacent to v.

$$\Gamma(G, v) = \{v_j \in V | (v, v_j) \in E\}$$

The degree of a vertex $deg(v)$ is equal to the number of its neighbor ($deg(v) = |\Gamma(G, v)|$). The degree of a graph $\Delta(G)$ instead is the maximum degree among all vertices.

A random graph of size n and density probability p is a graph with n vertices and the probability of an edge between two vertices p. This kind of graph is usually computed and generated on the fly and used to test the algorithm. A subgraph G' induced by $V' \subseteq V$, written as $G' = G[V']$ is a core of order k, or a k-core iff $deg_{G'}(v) \geq K$ for each $v \in V'$. The core number of a vertex v, denoted by $k(v)$, is the highest order of a core that contains v. The core number of a graph $G = (V, E)$, denoted by $k(G)$, is the maximum core number among the vertices of G.

We also define the *ego-network* as $N^+[v]$ as the subgraph generated by the higher-ranked neighbors of v concerning a given order.

Look at the Figure 1.5, if vertex are in the descent order $\{v_1, v_2, v_3, v_4, v_5\}$ the $N^+[v_3]$ for instance is given by the set $\{v_1, v_2\}$. A commonly used order is the degeneracy order it is given by ordering the vertices by their core number.

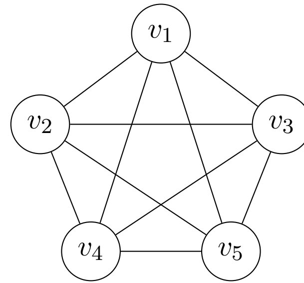


Figure 1.5: A Complete graph.

1.2 Problems and Algorithms

The clique problem is a collection of problems that aims to find cliques in a graph, there exist many related problems: The Maximal Clique Enumeration (MCE), as the name suggests, aims to list all the maximal cliques in a graph. The Maximum Clique Problem consists of finding a clique of maximum size in a graph, it is a subproblem of MCE. We also have a k-clique finding problem which is a decision problem that tells whenever there is a clique of size k in a graph, and last the Maximum Clique Enumeration problem that aims to find all the Maximum cliques in a graph. All those problems are classified as NP-Hard so no algorithm can compute the exact solution in polynomial time. Starting from the MCE problem, one of the most used algorithms to solve that problem is the Bron-Kerbosch [3] Algorithm, it is a Branch and Bound and Backtracking Algorithm, this algorithm as explained in the following chapters is the base search procedure to solve all the other problems. Through the years the algorithm has been studied and optimized, the first optimization was the choice of the "*pivot vertex*" able to cut useless branches that do not belong to any maximal clique, next we have the degeneracy vertex ordering able to reduce the dimension of each level-one induced subgraph. The Bron-Kerbosch algorithm has been used and optimized to solve the maximum clique problem, Tomita [18] designed an efficient algorithm that can compute the maximum clique by cutting most of the useless branches by employing greedy graph coloring. His algorithm is used as a reference for other algorithms that are designed for specific classes of graphs: Hard (Dense graph) and Large Sparse. Those kinds of algorithms have a huge section of independent code so they are easily extended and brought to multicore CPU. Recently MCE has been efficiently brought even to GPUs, by parallelizing the Bron-Kerbosch algorithm [1]. GPU Algorithms adopt many strategies to reduce the amount of memory employed by each thread and use the second-level independent subtree tree unrolling to get the job for each thread to be executed, in this way the load between threads is more balanced. Another problem useful to compute the

maximal clique is the *Graph Coloring* problem in which pairwise adjacent vertices are assigned different colors or labels, a greedy version of this algorithm can help to speed up the process of computing the maximum clique. The maximum clique problem can be useful to compute also the solution of the Maximum Common Subgraph Problem, it can be solved by computing the maximum clique in the association graph. The association graph [10] between two graph G and H is a graph A which has:

- Vertex set: $V(A) = \{(v, v') \in V(G) \times V(H) : (v, v) \in E(G) \iff (v', v') \in E(H)\}$ those vertices that preserve a loop on both the starting graphs or does not preserve a loop on both the starting graphs.
- Two matching nodes (vertices of A) (u, u') and (v, v') are adjacent if $u \neq v$ and $u' \neq v'$, and if they preserve both edges and non-edges, so $(u, v) \in E(G) \iff (u', v') \in E(H)$.

1.3 Performance Evaluation

Various algorithms have been proposed to solve the maximum clique problem, but few of them have been brought to parallel machines. McCreesh [11] wrote one of the fastest implementations for multithreaded CPU, based on San Segundo and Tomita implementation, they perform work donation: in which whenever the algorithm finds idle threads, the working thread donates work to the idle thread to keep them busy, this strategy is very effective especially for a kind of family of graph, this family is known as DIMACS. DIMACS graphs are a special kind of graph, randomly generated with some characteristics, they were employed for the DIMACS Challenge, they are usually small and dense, so the search tree extends also in-depth because, in addition to having a big solution, it is very difficult to recognize useless branches. The maximum clique solvers are divided into two main categories depending on the size of the problem, the first one is the one we have talked about so far, and the second refers to large and sparse instances. San Segundo wrote one algorithm especially suited for large and sparse instances called BBMCSP [13] this algorithm is a single thread algorithm, which has a preprocessing strategy that allows cutting some vertices that do not lead to any maximal clique, so it reduces the graph dimension, then it uses BBMC with sparse bitset as search algorithm to find the best solution with a greedy coloring procedure designed for bitsets. This algorithm was parallelized in BBMCPara, he shows how effective was this kind of algorithm, it works for a very large set of datasets, and we talk about some million vertices and billions of edges. This implementation uses a sparse bitset encoding of vertices, it consists of keeping track only of the active set of bits useful for the computation, this implementation allows to use of less memory for each thread. It is important to say that usually the development and test of the

effectiveness of a new pruning strategy is done in randomly generated hard instances, and then the corresponding algorithm for real-world networks is written based on the developed one for dense instances. Following this procedure other optimizations have been developed to solve the MCP, which uses the Partial MaxSAT strategy to compute a better upper bound to cut more vertices in the search tree, the algorithm is called MoMC (Mixed order Maximum Clique) [7], it is a single thread and very fast used for little and dense graph, next it was adapted to large and sparse graphs. LMC [6], it uses a very light preprocessing strategy to reduce the number of vertices processed by the search procedure. The last algorithm we cite is MC-BRB [4] it is a new framework algorithm called Branch-Reduce and Bound, it still applies the same pruning technique of the previously mentioned but with an additional Reduce technique, which is particularly effective on sparse graphs which is locally dense.

Chapter 2

GPU Architecture

This chapter briefly explains the differences between the CPU and GPU Architecture, then explains the main concepts of the GPU architecture such as threads, blocks, warps and a brief explanation of its memory hierarchy.

2.1 Overview of GPU Architecture

To better understand how to bring the maximum clique solver on GPUs we need to differentiate the two architectures, indicating the main components and explaining the existing memory hierarchy.

2.1.1 GPU Architecture

Before going into detail on how the implementation works on GPU is it important to explain how the GPU Architecture works. CUDA is a library that allows you to interface with the GPU, it provides the general functions able to allocate memory and execute kernels on GPUs. The kernels are CUDA Programs that will be executed on GPU. The GPU architecture is devoted to programs able to execute most of its operations in parallel, the architecture is a little bit different from the CPU.

The CPU and the GPU architectures are both composed of Cores, control unit L1, L2 Cache, and DRAM. By contrast, the GPU architecture has many more cores than the CPU one but most of the cores have the same control unit and L1 Cache, furthermore, the GPU does not have L3 Cache, and the L2 one is shared among all cores.

Inside the GPU the execution of a single execution flow is represented by a thread as for the CPU, but inside the GPU we can classify a group of threads called *block*, a block represents a variable group of threads decided by the programmer, usually number assigned are proportional to the number of thread that the warp can manage, a warp is a group of threads in which his length is fixed and depend by the GPU Architecture, an usual value is

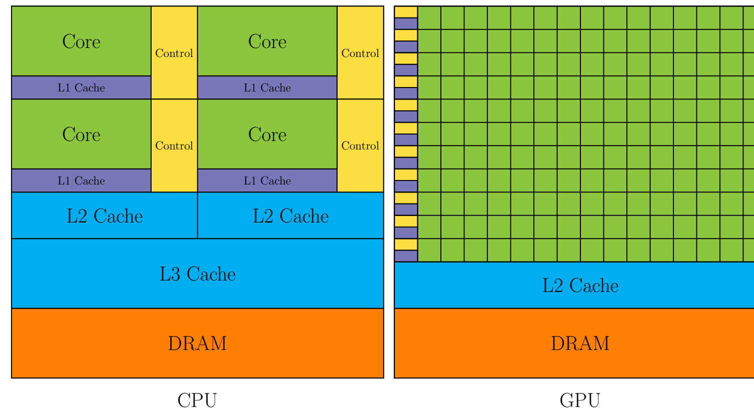


Figure 2.1: Main difference between CPU and GPU Architectures [12]

32 threads. Threads inside a warp can execute a common path, if the path of each thread diverges the paths are executed serially one after another. A set of Blocks corresponds to a grid, the grid dimension is specified when the kernel is launched and it is up to the programmer to decide its size.

Each block of threads, when the kernel is launched, is assigned to the Streaming Multiprocessor (SM), which executes the block. The number of SMs influences the number of concurrent blocks that the GPU can run, A GPU which have more SMs can run many more blocks concurrently.

Blocks and grids can have more than one dimension, it is up to the programmer to define the number of dimensions in which threads or blocks are indexed. We can choose up to three dimensions.

For instance, if we launch a block size of (4,4) this means that we have 16 threads launched per block each one indexed (0, 0), (0, 1), (0, 2), ..., (3, 3). As just said also grid can be launched with multiple dimensions so again if we launch a grid of size (2, 3) we are launching 6 blocks of threads each one indexed from (0, 0), (0, 1), (0, 2), (1, 0), ..., (1, 2).

When a block is assigned to an SM it is subdivided into warps, each warp is scheduled by the corresponding warp scheduler of the SM, Each streaming multiprocessor has the same number of warp schedulers, in modern NVIDIA cards an SM has 4 warp schedulers able to schedule 4 warps or 128 threads at a time. Pay attention that as said before each thread in a warp keeps a program counter and manages its execution flow, if we have branches and the same thread does not get inside the branch threads are disabled until the branch finishes, so the warp keeps also a mask of active threads.

All the application that involves the utilization of the GPU are programmable with CUDA the library has a programmable API written in C/C++ Language. So our application has been written in C/C++;

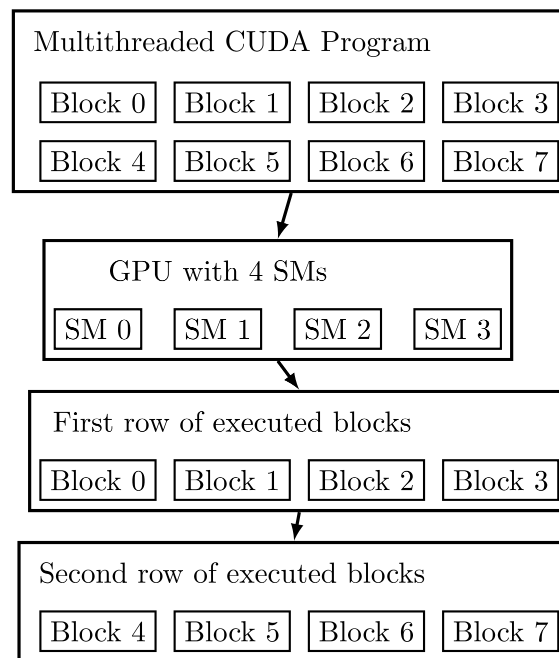


Figure 2.2: How a multithreaded CUDA program is handled by the GPU, we see that each block is assigned to an SM, and all the blocks will be executed in two cycles. Faster GPUs scale over the number the SMs, this is to give an idea of how it works, because actually, SMs can handle more than one block at a time, but if we exceed that limit they will be executed in order after terminating the precedents, pay attention that the case shown in figure each block of the row terminates together.

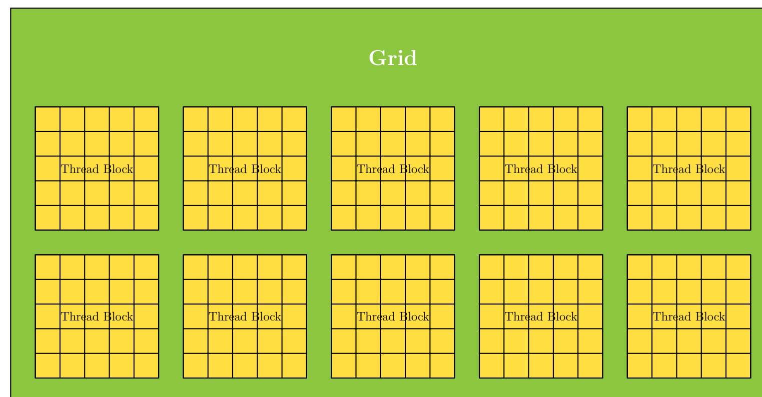


Figure 2.3: Thread hierarchy, in green the entire grid, in yellow small squares are thread, a group of thread represents a Thread Block. In figure a Thread Block of (5, 5) in a grid of dimension (5, 2).

2.1.2 GPU Memory

The GPU card owns different types of memory on board they are both inside or outside the chip: The main memory outside the chip is the **Global Memory** also known as VRAM, this kind of memory is the largest one, it can reach sizes of tens of gigabytes depending on the price of the GPU card. We mention our testing GPU Card that has 8 GBs of Global memory. The chip has more than one type of memory:

- L1 cache
- Shared Memory
- L2 Cache
- Constant Memory
- others that not will be used as texture cache.

The L1 cache and the Shared memory own the same memory hardware, the amount of L1 cache available depends on how much Shared memory your program will use. As said shared memory and L1 cache belong to the chip they are placed in the Streaming Multiprocessor and it is shared between threads of the same block, since it is on the chip it is much faster than the global memory, and it is usually employed to store data that will be used as soon as possible. It reaches dimensions of tens of Kilobytes up to hundreds. The L2 cache is located outside the streaming multiprocessor and it is common to all the streaming multiprocessors its dimensions are larger than the L1 because it is common to all SMs. The Constant memory has its cache

memory which is independent of the L2, it usually stores immutable data, so data that cannot change during the runtime of the program. Last but not least just mention texture memory is also resident in global memory that has its cache usually employed for multidimensional data, it also provides some specific functionality like hardware interpolation, etc.

Chapter 3

Clique Algorithms

As said before the maximum clique algorithm has been extensively studied over the years since 1973. This chapter describes the evolution of the main algorithm introducing the most important optimizations able to efficiently compute the solution, starting from the enumeration problem.

3.1 Bron-Kerbosch Algorithm

Bron-Kerbosch is the most popular algorithm to find all the maximal cliques in a graph. The algorithm has been analyzed, it is a Backtracking algorithm, so it looks for any possible solution. We can try to explain the algorithm by starting from the Algorithm that generates the powerset of a set:

Algorithm 1: PowerSet

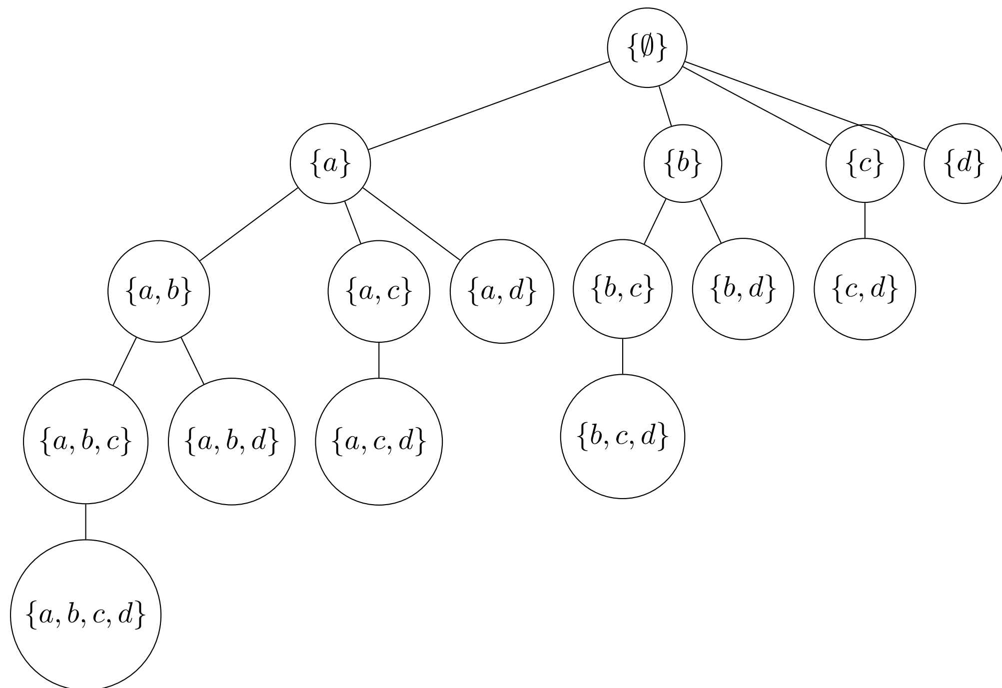
Input : Set V, Set S, where S starts as $\emptyset$
Output: List all elements of the powerset of V

```
1 begin
2     Print S
3     for v ∈ V do
4         PowerSet(V \ v, S ∪ {v})
5         V ← V \ {v}
6     return
```

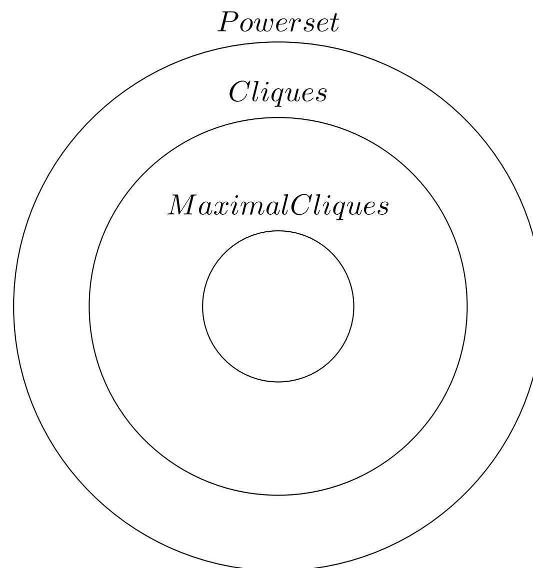
For example: given a set $V = \{a, b, c, d\}$ the PowerSet of V will be:

$$
\begin{aligned}
&\{\{\emptyset\},\\
&\{a\},\{a,b\},\{a,b,c\},\{a,b,c,d\},\{a,b,d\},\{a,c\},\{a,c,d\},\{a,d\},\\
&\{b\},\{b,c\},\{b,c,d\},\{b,d\},\\
&\{c\},\{c,d\},\\
&\{d\}\}
\end{aligned}
$$

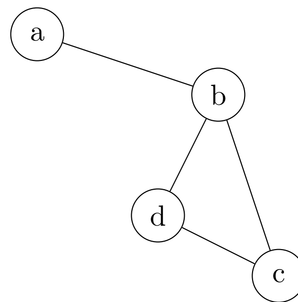
The search tree, with the set S at each node, will be:



The powerset algorithm is a generalization of the Bron-Kerbosch algorithm, its output is a subset of the powerset.



Let's consider a graph $G = (V, E)$ of just 4 nodes:



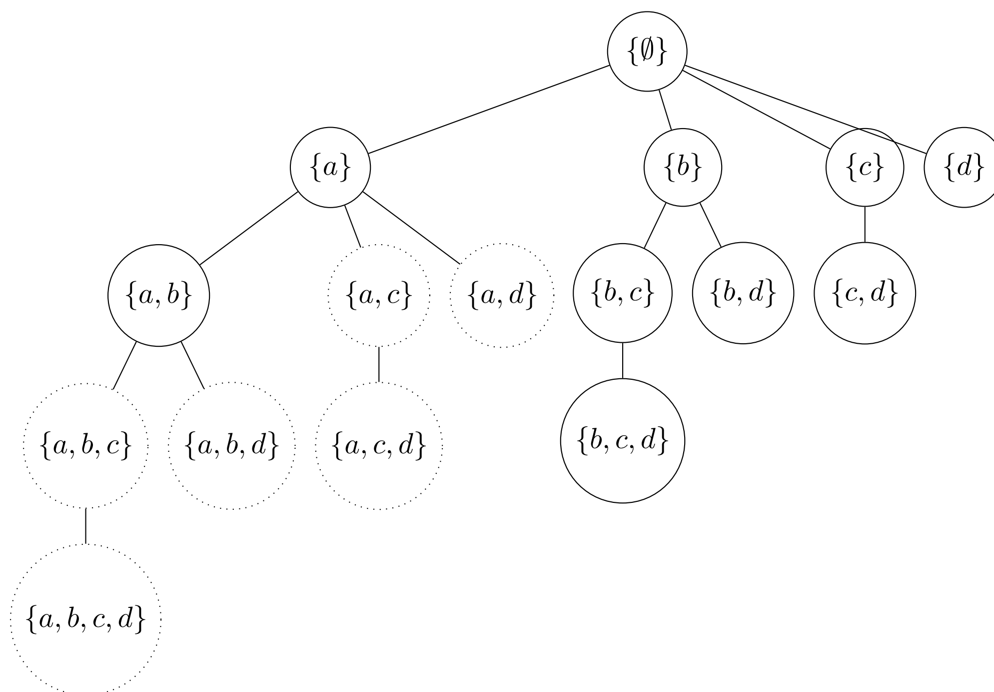
Now slightly modify the algorithm of the powerset by introducing the P set that represents the set of potential vertices that can be in a clique, and the set S that is the current solution:

Algorithm 2: Enumerate all cliques of Graph G

Input : Graph G, Set P, Set S, where P starts with V and S as $\emptyset$
Output: List all cliques of G

```
1 begin
2     Print S
3     for v ∈ P do
4         Cliques(G, P ∩ N(G, v), S ∪ {v})
5         P ← P \ {v}
6     return
```

Following the algorithm the search tree will be slightly different, some branch were cut.



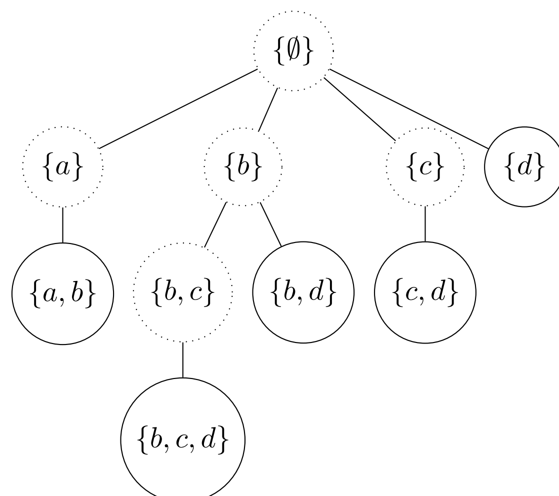
To print out just the leaf nodes where the maximum clique stays we need to insert the print inside the termination condition $P = \emptyset$.

Algorithm 3: Enumerate all cliques of Graph G

Input : Graph G, Set P, Set S, where P starts with V and S as $\emptyset$
Output: List all cliques of G

```
1 begin
2     if P = ∅ then
3         Print S
4     for v ∈ P do
5         Cliques(G, P ∩ N(G, v), S ∪ {v})
6         P ← P \ {v}
7     return
```

Dotted those nodes that do not will be printed out.



To further reduce the output to the maximal cliques we have to introduce another set X the set of the vertex already served. This set should be updated each time we already have explored a vertex. Furthermore, $P = \emptyset$ is not a sufficient condition to say that S is a maximal clique, so we require that X is also $\emptyset$.

Algorithm 4: BronKerbosch: Enumerate all maximal cliques of Graph G from [1]

Input : Graph G, Set P, Set S, Set X, where P starts with V, S and X as $\emptyset$
Output: List all maximal cliques of G

```
1 begin
2     if P = ∅ ∧ X = ∅ then
3         Print S
4     for v ∈ P do
5         BronKerbosch(G, P ∩ N(G, v), S ∪ {v}, X ∩ N(G, v))
6         P ← P \ {v}
7         X ← X ∪ {v}
8     return
```

Pay attention when we recur over the subtree we have to consider only vertices already served but also those that are in common to chosen vertex (v) so the following operation makes sense $X \cap (G, v)$. At this point the search tree will print just the sets $\{a, b\}$ and $\{b, c, d\}$.

To further reduce the search space is it possible to select at each recursion level a "*piovot vertex*": the pivot is a chosen vertex in which its neighborhood is removed from the branching vertices since a maximal clique

composed by the pivot and its neighbor will be found by just branching in the pivot vertex, the pivot vertex is selected based on the dimension of its neighborhood. In the maximum clique algorithm, this optimization is replaced by the coloring pruning strategy that Tomita introduces, this is further explained in the next sections. Is it possible to improve the BK algorithm by computing a certain vertex order called "*degeneracy order*", this order is coupled with another optimization, the "*first-level independent subtree*" commonly used even in the maximum clique algorithm, thanks to this two optimization is it possible to parallelize the algorithm running it on Multi-cores architectures and reduce the size of the P set computed in the first level of the recursion tree.

Algorithm 5: BronKerbosch: First-level independent subtree [1]

Input : Graph $G = (V, E)$
Output: List all maximal cliques of G

```
1 begin
2   for v_i ∈ V with respect to the degeneracy order do
3     P ← N(G, v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|}}
4     X ← N(G, v_i) ∩ {v_1, v_2, ..., v_{i-1}}
5     BronKerbosch(G, P, {v_i}, X)
6   return
```

3.2 The k-core decomposition and ordering

As for the Bron-Kerbosch algorithm the sparse versions of the maximum clique search procedure make use of the k-core to compute the ordering and an initial clique for reducing the input graph, this ordering is computed in linear time by the Algorithm 6.

Vertices in V are initially ordered in ascending order with respect to their degree, then the algorithm extracts the vertex v_i with minimum degree from V and removes it from the graph G updating the vertices degrees, the algorithm loops until V is empty. The vertices in O are ordered concerning the degeneracy order extracted for V, and the core number is exactly the degree when the vertex has been extracted from V. Actually, it is not required to remove a vertex from V at each loop. we just need to update their degrees.

3.3 Tomita's Pruning Strategy

One of the first and fastest algorithms to find the maximum clique was written by Tomita (2003). How It has already been said it is based on a

Algorithm 6: k-cores decomposition.

Input : Graph $G = (V, E)$
Output: Returns a vertex ordering O, an initial clique C_0 and core numbers

```
1  begin
2      O ← ∅
3      deg() ← Compute the degrees of each vertex v ∈ V
4      Sort the vertices in increasing degree ordering.
5      for v_i ∈ V in increasing degree ordering do
6          if deg(v_i) + 1 = |V| − i then
7              C_0 ← {v_j ∈ V | j ≥ i}
8              for v_j ∈ C_0 do
9                  append(v_j, O)
10                 core(v_j) ← deg(v_i)
11             break
12         append(v_i, O)
13         core(v_i) ← deg(v_i)
14         for v_j ∈ N(G, v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|}} do
15             deg(v_j) ← deg(v_j) − 1
16             Sort V and re-index vertex v_j in V with respect to the
                 degree contained in deg() order.
17     return (C_0, O, core())
```

pruning strategy with a greedy graph coloring which gives us an upper bound on the size of the maximal clique found by branching on that vertices. The algorithm is described in Algorithm 7.

Algorithm 7: MCQ: Find the maximum clique of a Graph G [18]

Input : Graph G
Output: Returns the maximum clique of G

```
1 begin
2     global Q = ∅
3     global Q_max = ∅
4     Sort vertices of V in descending order with respect to their
        degree
5     for i = 1 to Δ(G) do
6         C[V[i]]=i
7     for i = Δ(G) + 1 to |V| do
8         C[V[i]]=Δ(G) + 1
9     EXPAND(V, C) return Q_max
```

The procedure EXPAND described Algorithm 8, recalls the one of Bron-Kerbosch, in which it loops over the vertices in P but instead of just recurring, EXPAND executes the pruning strategy based on the color number, After the return from the sublevel it checks if the incumbent is greater than the maximum clique at this point it store the maximum clique. Keeping the set X is useless because we explore the tree in a depth-first manner and we keep track of the maximum clique. The coloring strategy has been refined in the subsequent paper.

The pruning strategy procedure is described in Algorithm 9. From the Tomita's paper [18]: "This procedure assigns in advance for each $p \in R$ a positive integer $N[p]$ called the *Number* or *Color* of p with the following property:

- If $(p, r) \in E$ then $N[p] \neq N[r]$
- $N[p] = 1$, or if $N[p] = k > 1$, then there exist vertices $p_1 \in N(G, p)$, $p_2 \in N(G, p), ..., p_{k-1} \in N(G, p)$ in R with $N[p_1] = 1$, $N[p_2] = 2, ..., N[p_{k-1}] = k - 1$.

Consequently, we know that $\omega(R) \leq Max\{N[p]| \; p \in R\}$, and hence if $|Q| + Max\{N[p]| \; p \in R\} \leq |Q_{max}|$ holds then we can disregard such R.

The value of $N[p]$ for each $p \in R$ is assigned step by step in the following manner: Assume that vertices in $R = \{p_1, p_2, ..., p_m\}$ are arranged in this order. First let $N[p_1] = 1$. Subsequently, let $N[p_2] = 2$ if $p_2 \in N(D, p_1)$ else

Algorithm 8: EXPAND: Find the maximum clique of a Graph G [18]

Input : Graph G, Set R, Array C
Output: Returns the maximum clique of $G(R)$

```
1  begin
2      while R ≠ ∅ do
3          assign to p the last vertex of R
4          if Q + C[p] > |Q_max| then
5              Q ← Q ∪ {p}
6              R_p ← R ∩ N(G, p)
7              if R_p ≠ ∅ then
8                  copy C to C'
9                  NUMBER-SORT(R, C')
10                 EXPAND(R_p, C')
11             else if |Q| > |Q_max| then
12                 Q_max ← Q
13             Q ← Q \ {p}
14         else
15             return
16         R ← R \ {p}
17     return
```

$N[p_2] = 1$, ..., and so on. After Numbers are assigned to all vertices in R, we sort these vertices in ascending order with respect to their *Numbers.*"

Algorithm 9: NUMBER-SORT: Assign colors with a greedy strategy to vertices in R [18]

Input : Graph G, Set R, Array C
Output: Returns the color number in C as ordered array

```
1  begin
2      maxno=1
3      //NUMBER
4      C_1 ← ∅
5      C_2 ← ∅
6      ...
7      C_m ← ∅
8      while R ≠ ∅ do
9          assign to p the first vertex in R
10         k = 1
11         while C_k ∩ N(G,p) ≠ ∅ do
12             k = k + 1
13         if k > maxno then
14             maxno = k
15             C_{maxno+1} ← ∅
16         C[p] = k
17         C_k ← C_k ∪ {p}
18         R ← R \ {p}
19     //SORT
20     i = 1
21     for k = 1 to maxno do
22         for j = 1 to |C_k| do
23             R[i] = C_k[j]
24             i = i + 1
25     return
```

The time complexity of the procedure is $O(|R|^2)$. Also Tomita says that [18] "The quality of such sequential coloring depends heavily on how the vertices are ordered. The last operation (sorting) is executed in $O(|R|)$ time".

3.4 San Segundo: BitBoard MaxClique Algorithm

San Segundo presents another exact maximum clique algorithm implementation, based on bitset [17]. Bitset is the new set representation in memory in which vertices in the set are represented by bits set to 1, on a set of a word dimension we can represent at most 32 vertices so if the $n - th$ bit is set to 1, this means that the v_n vertex is in that set. In this way, all the operations performed on sets such as Unions or Intersections become bitwise operations between registers, speeding up that process. The algorithm slightly changes, it is described in Algorithm 10.

Algorithm 10: BB-MaxClique: Find the maximum clique on a graph G [17]

Input : Graph G, BitSet U_{BB}, Set U_L, Set S_{max}
Output: Returns the Set that represents the maximum clique

1 /* Initially U_{BB} is a bit model for a set of vertices in the input graph $G = (V, E)$ and U_L is a standard encoding of V */
2 **begin**
3   **while** $U \neq \emptyset$ **do**
4   select a vertex v from U_L in order
5     $U_{BB} \leftarrow U_{BB} \setminus \{v\}$
6   **if** $|S| + C(v) > |S_{max}|$ **then**
7       $S \leftarrow S \cup \{v\}$
8   **if** $U_{BB} \cap N_{BB}(G, v) \neq \emptyset$ **then**
9   *BB-Color*$(U_{BB} \cap N_{BB}(G, v), U_L, C', |S_{max}| - |S| + 1)$
10   *BB-MaxClique*$(U_{BB} \cap N_{BB}(G, v), U_L, C', S_{max})$
11   **else if** $|S| > |S_{max}|$ **then**
12         $S_{max} \leftarrow S$
13       $S \leftarrow S \setminus \{v\}$
14   **return**

The operation between the potential vertices that can represent a clique and the neighborhood of a vertex this time are bitwise $(U_{BB} \cap N_{BB}(G, v))$. All the sets denoted by $X_B B$ are bitsets.

Also, the coloring procedure has been adapted and now presents bitwise operation. The operation presented in Algorithm 11: $(\overline{N_{BB}(G(Q_{BB}), v)})$ is the complement of the set given by the neighborhood of v.

Algorithm 11: BB-Color: Assign a color to a set of vertices U_{BB} [17]

Input : Graph G, BitSet U_{BB}, Set U_L, Set k_{min}
Output: Returns the colors of vertices in U_{BB}

```
1  /* Vertices in the input candidate set U_BB must be in the same
      order as in the initial input graph */
2  begin
3      Q_BB ← U_BB
4      k = 0
5      while U_BB ≠ ∅ do
6          C_k ← ∅
7          while Q_BB ≠ ∅ do
8              select the first vertex v ∈ Q_BB
9              C_k ← C_k ∪ {v}
10             Q_BB ← Q_BB \ {v}
11             Q_BB ← Q_BB ∩ overline(N_BB(G(Q_BB), v))
12         U_BB ← U_BB \ C_k
13         Q_BB ← U_BB
14         if k ≥ k_min then
15             U_L ← C_k
16             C[v] = k
17         k = k + 1
18     return
```

3.5 McCreesh: Multithreaded Maximum Clique

Ciaran McCreesh presents a parallel version of the Maximum clique algorithm, this version archives near linear and superlinear speed-ups, It has been tested on Hard graphs (little up to about 30,000 vertices with variable density), He solved some DIMACS challenges that required some days to be solved. The parallel version is described in Algorithm 12.

Algorithm 12: A threaded algorithm to deliver the max clique [11]

Input : Graph G
Output: Returns the Set of integers that represent the maximum clique

```
1  begin
2  │ shared C_max ← ∅
3  │ shared q ← an empty Queue of (Set, Set)
4  │ permute G so that vertices are in non-increasing degree order
5  │ launch the populating thread do expand(G, q, ∅, V(G), C_max)
6  │ launch multiple worker threads do
7  │ while there is work left do
8  │ │ (C, P) ← dequeue(q)
9  │ └ expand(G, q, C, P, C_max)
10 │ join all threads
11 └ return C_max
```

It is based on a simple strategy in which a job is defined by the process of coloring and performing bitset operation in a node in the recursion graph, this process is called a thread job, now the algorithm maximizes the time in which each thread is busy, so the algorithm whenever find an idle thread it creates new work for the idle threads and keep them busy, instead if there are no idle threads this thread keep working by recurring in the sub-level of the tree. Some version uses also bitsets to speed up set operation. Then the *colorOrder* procedure is the same as San Segundo, but the names of the sets used are different, the McCreesh's implementation is described in Algorithm 14.

Even if the speedup seems linear or superlinear hype-threading complicates things. Hyper-threading makes the physical processor appear as two logical processors, this kind of optimization makes our speedup increase by only up to 30%, this number is taken from benchmarks.

Algorithm 13: expand: main search procedure for max clique [11]

Input : Graph G, Queue of (Set, Set) q, Set C, Set P, Set C_{max}
Output: Returns the Set of integers that represent the maximum clique

```
1  begin
2    populate ← true if we are the populating thread, and |C| = 1,
       otherwise false
3    (colour, order) ← colourOrder(G, P)
4    for i ← |P| down to 1 do
5      if |C| + colour[i] > |C_max| then
6        v ← order[i]
7        C ← C ∪ {v}
8        P' ← P ∩ N(G, v)
9        if P' = ∅ then
10         if |C| > |C_max| then
11           C_max ← C
12       else
13         if the populating thread is done, and q is empty,
             and there are idle workers then
14           populate ← true
15         if populate then
16           enqueue(q, (C, P'))
17         else
18           expand(G, q, C, P', C_max)
19       C ← C \ {v}
20       P ← P \ {v}
21   return
```

Algorithm 14: colorOrder: vertex coloring [11]

Input : Graph G, Set P

Output: Returns two sets, one that represents the color number and the other the name of the vertices

```
1  begin
2      color ← array of integer
3      order ← array of integer
4      P′ ← P
5      k = 1
6      while P′ ≠ ∅ do
7          Q ← P′
8          while Q ≠ ∅ do
9              v ← the first element of Q
10             P′ ← P′ \ {v}
11             Q ← Q \ {v}
12             append k to color
13             append v to order
14             Q ← Q ∩ N̄(G, v)
15         k = k + 1
16     return (color, order)
```

3.6 San Segundo: BBMCPara

San Segundo presents BBMCPara a parallel version of another algorithm BBMCSP that aims to find the maximum clique in large and sparse graphs. BBMCSP uses the new representation of *sparse bitsets*, in which a bitset contains just bitset blocks that have at least one bit set to 1, words with value 0 are omitted to use less memory during runtime and make bitset operations faster. BBMCSP uses also a preprocessing strategy that aims to reduce the number of candidate vertices, both to speed up computation and to reduce the amount of memory used by each thread. BBMCSP is described in algorithm 15.

Algorithm 15: BBMCSP [13]: max clique computation for large sparse graphs

Input : Graph G
Output: Returns the maximum clique

```
1  begin
2      Perform core analysis and compute K(G)
3      S_max ← INITIAL_CLIQUE(G)
4      V' ← V \ {v ∈ V : K(v) < |S_max|}
5      Sort vertices in G' = G[V'] according to degeneracy
6      while |V'| ≠ 0 do
7          select vertex w from V' in reverse order
8          if K(w) ≥ |S_max| then
9              INITIAL_SEARCH(G', w)
10         V' ← V' \ {v}
11     return
```

BBMCSP starts by performing the core analysis which consists of determining the k-core of the graph and the core number of each vertex: Then it computes an initial clique with a maximum clique greedy algorithm this initial clique will be used as a threshold to filter those vertices with a core number greater than the initial clique size. Then for each vertex of V, it calls the *INITIAL_SEARCH* procedure to further process the graph.

The parallel version of BBMCPara uses the first level of the recursion tree as a job for the thread. Benchmark results show that speedup is linear with the number of threads. The Search and the color procedure remain unchanged with respect to BBMC so will be not reported. As we can see from the algorithm inside the search procedure during the first level subtree the subgraph is further processed and reduced, and the main search procedure (similar to BBMC) starts.

Algorithm 16: BBMCSP [13]: max clique computation for large sparse graphs

Input : Graph G
Output: Returns the maximum clique

1 **begin**
2 $U_v \leftarrow N(G, v)$
3 **if** $|U_v| < |S_{max}|$ **then**
4 **return**
5 $col \leftarrow$ Compute the size of greedy sequential coloring SEQ(U_v)
6 **if** $col < |S_{max}|$ **then**
7 **return**
8 $W \leftarrow U_v \cup \{v\}$
9 Perform core analysis and compute $K(G[W])$
10 **if** $K(G[W]) < |S_{max}|$ **then**
11 **return**
12 $U'_v \leftarrow W \setminus \{w \in W : K(w) < |S_{max}|\} \setminus \{v\}$
13 $L \setminus U'_v$ in degenracy ordering
14 $C(U'_v) \leftarrow W \setminus \{c(w)|C(w) \leftarrow K(w) \forall w \in U'_v \subseteq W\}$
15 $SPARSE_SEARCH(U'_v, \{v\}, S_{max}, C(U'_v), L)$
16 **return**

3.7 Tomita's Re-NUMBER algorithm

Tomita's NUMBER Algorithm subdivides the branching vertices into k independent sets C_i each vertex of the independent set has color number i, the maximum color is given by the number of independent sets k it is as already said an upped bound of the maximum clique, a good quality coloring provide an upper bound that is as near as possible to the chromatic number, to further improve this bound Tomita designed another procedure Re-NUMBER [19] that initially subdivide the branching vertices in k_{min} independent sets when a branching vertices v cannot be added to any independent set numbered from 1 to k_{min} where k_{min} is given by $|C_{max}| - |C|$, it try to re-number vertices in those independent set making space for the new branching vertices, the algorithm is described in Algorithm 17.

Algorithm 17: Re-NUMBER [19] Try color v with color less equal than k_{min}

Input : Graph G, Vertex v, Sets $C_1, C_2, ..., C_{k_{min}}$
Output: Returns true if v has been colored false otherwise

```
1  begin
2    for i from 1 to k_min − 1 do
3      if |C_i ∩ N(G, v)| = 1 then
4        w ← vertex from C_i ∩ N(G, v)
5        for j from i + 1 to k_min do
6          if |C_j ∩ N(G, w)| = 0 then
7            C_i ← C_i \ {w}
8            C_j ← C_j ∪ {w}
9            C_i ← C_i ∪ {v}
10           return true
11   return false
```

This procedure is recalled inside the NUMBER-SORT procedure, it promises to improve the runtimes for locally dense graphs by at least an order of magnitude. This procedure has been implemented also by San Segundo in its BBMC framework, it promises to improve runtimes on locally dense instances, the algorithm that owns this procedure is called BBMCR [16]. In BBMCR this procedure is applied under different conditions.

3.8 PMaxSAT based Pruning Strategy: MoMC

Li focuses on reducing the number of branches of the search tree, which can be archived by using a dynamic vertex reordering strategy because a static strategy can be effective for certain kinds of graphs. He will show that combining these two strategies dynamic and static can result in more effectiveness than using just one.

In $DoMC_0$ the general algorithm remembers those previously described it is used to compute the maximal clique for little and dense graphs it is described in algorithm 18:

Algorithm 18: MC: max clique computation for dense graph from [7]

Input : Graph G, Set P, Ordering O_0, Set C, Set C_{max}
Output: Returns the maximum clique C_{max}

```
1  begin
2      if P = ∅ then
3          return C
4      B ← GetBranches(G[P], |C_max| − |C|, O_0)
5      if B = ∅ then
6          return C_max
7      A ← P \ B
8      Let B = {b_1, b_2, ..., b_|B|}andb_1 < b_2 < ... < b_|B| w.r.t.O_0
9      for i = |B| downto 1 do
10         C_1 ← MC(G, N(G, b_i)∩({b_{i+1}, ..., b_|B|} ∪ A), O_0, C ∪ {b_i},
               C_max)
11         if |C_1| > |C_max| then
12             C_max ← C_1
13     return C_max
```

The algorithm $GetBranches_{d0}$ is relatively simple it builds the set of independent set Π until the size of Π reaches r then it applies the Re-NUMBER procedure to further reduce the set of branching vertices B_{d0}. DoMC is the second algorithm presented it uses incremental MaxSAT reasoning to detect conflict and further reduce the set of branching vertices. So given a graph G we first apply the independent set partitioning as the precedent algorithm then it builds the PMaxSAT instance ϕ to detect conflicts. From Li et al. [7] "A **literal** is a propositional variable x or its negation $\overline{x}$. A **clause** is a disjunction of literal, and a CNF formula is a conjunction of clauses. A truth assignment I satisfies a literal if its assignment is 1 (if literal is x), satisfies a clause c if it satisfies at least a literal in the clause, and

Algorithm 19: *GetBranches*$_{d0}$: Compute the set of branching vertices [7]

Input : Graph G, Int r, Ordering O_0
Output: Returns the maximum set B of branching vertices

```
1  begin
2      B_d0 ← ∅
3      Π ← ∅
4      while V ≠ ∅ do
5          v ← the greatest vertex of V w.r.t. the ordering O_0
6          V ← V \ {v}
7          if there is an independent set D in Π in which is not
             adjacent to any vertex then
8              D ← D ∪ {v}
9          else
10             if |Π| < r then
11                 create a new independent set D = {v}
12                 Π ← Π ∪ {D}
13             else
14                 if there is a D in which v has only one adjacent
                     vertex u, and u can be inserted into another
                     independent set D' then
15                     D' ← D' ∪ {u}
16                     D ← D ∪ {v}
17                 else
18                     B_d0 ← B_d0 ∪ {v}
19     return B_d0
```

satisfies a CNF formula if satisfy all its clauses. A Partial MaxSAT instance is a set of clauses in which some of them are declared to be soft and some hard. Given a partial MaxSAT instance, the partial MaxSAT problem consists in finding a truth assignment that satisfies all its hard clauses, and the maximum number of soft clauses". The MaxClique problem can be reduced in a MaxSAT instance as follows: the vertices of the graph represent the propositional variable, and a propositional variable is assigned true if it belongs to the maximum clique false otherwise. The Partial MaxSAT instance contains a hard clause $\overline{x_i} \wedge \overline{x_j}$ for each pair of non-adjacent vertices and a soft unit clause for each vertex $v \in V$. Li and Quan(2010) [8], improved this encoding by declaring a soft clause for each independent set in G, which is the disjunction of variables for each vertex of the independent set. The conflicts are detected in the new *IncMaxSAT* that is called by $GetBranches_d$ procedure described in Algorithm 20.

Algorithm 20: $GetBranches_d$: Compute the set of branching vertices [7]

Input : Graph G, Int r, Ordering O_0
Output: Returns the maximum set B of branching vertices

```
1 begin
2     B_d0 ← GetBranches_d0(G, r, O_0)
3     if B_d0 = ∅ then
4         return ∅
5     else
6         A_d0 ← V \ B_d0
7         B_d ← IncMaxSAT(G, O_0, A_d0, B_d0)
8         return B_d
```

IncMaxSAT, described in Algorithm 21, aims to find a subset of conflicting soft clauses. Once the conflict has been detected the set of soft clauses has to be weakened before detecting the next conflict, after the weakening process the number of conflicts after weakening the soft clauses is reduced by 1. The weakening process is described in Algorithm 21 Lines: 19 - 20. The weakening process is useful to remove the precedent conflict and make the detection of the next conflict independent from the previously one. SoMC is the other algorithm proposed, it is based on the assumption that branching in a smaller vertex w.r.t the ordering in O_0 is a much easier branching in a vertex which is bigger w.r.t the ordering in O_0. The function $GetBranches_s$ implements this assumption: it looks for the greatest vertex in B_d and if in A_d finds a vertex less than max in B_d moves that vertex back and returns B_s which is B_d plus that vertex. This procedure is described in the following Algorithm:

Algorithm 21: IncMaxSAT: Further reduce the set of branching vertices [7]

Input : Graph G, Ordering O_0, Set A, Set B
Output: Returns the maximum set B of branching vertices

1 **begin**
2 $\phi \leftarrow$ the partial MaxSAT encoding og G without including the soft clauses for the vertices in B **while** $B \neq \emptyset$ **do**
3 $b \leftarrow$ the greatest vertex in B w.r.t. O_0
4 add a soft unit clause $\{b\}$ int ϕ and push $\{b\}$ into an empty stack S
5 **while** $S \neq \emptyset \wedge$ *empty clause is not derived* **do**
6 pop a unit clause u from S
7 $l \leftarrow$ the only literal in u record u as the reason for the value the variable in l satisfying l
8 **foreach** *clause* c *that contains* $\bar{l}$ **do**
9 **remove** *ovelinel* from c
10 **if** c *became a unit clause* **then**
11 **push** c into S
12 **if** c *is empty* **then**
13 $B \leftarrow B \setminus \{b\}$
14 **foreach** *clause* c' *in the same order they were pushed into* Q **do**
15 **foreach** *removed literal* l' *in* c' **do**
16 **if** *the reason* r *for literal* l' *is not in* Q **then**
17 **push** r into Q
18 restore all removed literals in their clauses
19 $\{\{b\}, c_1, ..., c_q\} \leftarrow$ the set of soft clauses in Q
20 let $z_b, z_{c_1}, ..., z_{c_q}$ be new variable $\phi \leftarrow (\phi \setminus \{\{b\}, c_1, ..., c_q\}) \cup (\{b \vee z_b\} \cup \{c_1 \vee z_{c_1}, ..., c_q \vee z_{c_q}\}) \cup \{z_b + z_{c_1} + ... + z_{c_q} = 1\}$
21 **break**
22 **if** *no empty clause is derived* **then**
23 **return** B
24 **return** B

Algorithm 22: *$GetBranches_s$*: Compute the set of branching vertices [7]

Input : Graph G, Int r, Ordering O_0
Output: Returns the maximum set B of branching vertices

```
1 begin
2     B_d ← GetBranches_d(G, r, O_0)
3     if B_d = ∅ then
4         return ∅
5     else
6         v ← the greatest vertex in B_d w.r.t. the ordering in O_0
7         B_s ← {u ∈ V | u ≤ v w.r.t. O_0}
8         return B_s
```

A lack point of $DoMC_0$, DoMC, and SoMC is that they do not exploit sufficiently the result of the previous search, to overcome this lack of incrementality they have been incorporated into a more efficient representation of the adjacency matrix, and an incremental upper bound. The adjacency matrix is $O(|V|^2)$ space-consuming, so it cannot fit entirely in cache memory. To use efficient cache memory is to explore the tree in order w.r.t O_0, this can be archived only by SoMC which uses static ordering of the branching vertices. To overcome this limitation an optimization from MCS has been adopted, it consists of the reconstruction of the set P, so that it appears in the same order as the adjacency matrix. this optimization can be time-consuming if it is performed in each level of the search tree, to it has been adopted just in the first level, leading to increasing performance by 10-15%. To use the result of the previous search an array has been adopted, called vertexUB, to store the upper bound of the clique obtained by branching in vertex v_i, so if the clique found is s then if the $vertexUB[v_i] \leq s$ the search can be pruned. The vertexUB is determined by the following rules taken from Li et al. [7]:

- **Inheritance rule**: If vertexUB[v_i,O] is defined on a set U_i and $Vi \subseteq Ui$, then vertexUB[v_i,O] can be defined on V_i with the same value as in U_i.
- **Incremental rule**: We define a function called *IncUB*(v_i,O) as follows. If $V_i \cup N(G, v_i) = \emptyset$, then *IncUB*($v_i$, O) = 1. Otherwise, *IncUB*(v_i, O) = 1 + *max* $\in V_i \cup N(G, v_i)$ vertexUB[u], provided that vertexUB[u] was already defined for each $u \in V_i \cap$ N(G, v_i) in the previous search. Obviously, vertexUB[v_i] can be defined to be *IncUB*(v_i, O). In other words, vertexUB[v_i,O] can be defined to be the maximum vertexUB of its neighbors in V_i plus 1, because any clique containing

v_i is formed by v_i and some of its neighbors.

- **Coloring rule**: If V_i can be partitioned into r independent sets, then vertexUB[v_i, O] can be defined to be r.
- **MaxSAT rule**: If V_i can be partitioned into r independent sets and t disjoint conflicting subsets of independent sets can be detected using MaxSAT reasoning, then vertexUB[vi, O] can be defined to be $r - t$
- **Branching rule**: vertexUB[vi, O] can be defined to be $|C_1| - |C|$ after BnB algorithm branches on vi with the growing clique C to obtain a clique C_1
- **Compatibility rule**: Let O and O' be two different vertex orderings. Let v_i be a vertex such that vertexUB[v_i, O] has been defined. If it holds that $\{u|v_i < uw.r.t.O'\} \subseteq \{u|v_i < uw.r.t.O\}$, then vertexUB[$v_i$,$O'$] can be defined to be vertexUB[v_i,O]

All this optimization leads to the following algorithms $DoMC2_0$, DoMC2, and SoMC2, those algorithm has the procedure MC2 in common described in Algorithm 23.

DoMC2 and SoMC2 are complementary algorithms both effective, so they can be combined, then another algorithm is introduced MoMC, which uses the procedure $GetBranches_m$ it is based on a parameter $\alpha = |B_d|/|B_s|$ to control the choice of using the $GetBranches_d$ or $GetBranches_s$, it is described in Algorithm 24.

This algorithm (MoMC) might be parallelized following the guidelines of McCreesh and Prosser to achieve lower runtimes. It has been tested on little and dense graphs like DIMACS and compared with the current state-of-the-art algorithms. In the end, it shows how it outperforms those kinds of algorithms.

Algorithm 23: MC2: max clique computation for dense graph [7]

Input : Graph G, Set P, Ordering O_0, Set C, Set C_{max}, Ordering O'
Output: Returns the maximum clique C_{max}

```
1  begin
2      if P = ∅ then
3          return C
4      B ← GetBranches(G[P], |C_max| − |C|, O_0)
5      if B = ∅ then
6          return C_max
7      A ← P \ B
8      Let A = {a_1, a_2, ..., a_|A|} and B = {b_1, b_2, ..., b_|B|} in increasing
         order w.r.t. O_0
9      define a new vertex ordering O' in P
10     for i = |A| down to 1 do
11         // compatibility rule and incremental rule
12         vertexUB[a_i, O'] ← min(vertexUB[a_i, O],|C_max| − |C|,
             IncUB(a_i, O'))
13     if |C| < 1 then
14         reconstruct the adjacency matrix to make the vertices in
             G[P] consecutive in the matrix w.r.t. the ordering in O'
15     for i = |B| downto 1 do
16         // incremental rule
17         vertexUB[v_i, O'] ← IncUB(b_i, O_i)
18         if vertexUB[b_i, O'] < |C_max| − |C| then
19             if b_i smaller than vertices in {b_{i+1}, ..., b_|B|} ∪ A w.r.t. O
                 and vertexUB[b_i, O] < |C_max| − |C| then
20                 // compatibility rule
21                 vertexUB[b_i, O'] ← vertexUB[b_i, O]
22             else
23                 C_1 ← MC2(G, N(G, b_i)∩({b_{i+1}, ..., b_|B|} ∪ A), O_0,
                     C ∪ {b_i}, C_max, O')
24                 vertexUB[b_i, O'] ← |C_1| − |C|
25                 if |C_1| > |C_max| then
26                     C_max ← C_1
27     return C_max
```

Algorithm 24: *GetBranches*$_m$: Compute the set of branching vertices [7]

Input : Graph G, Int r, Ordering O_0
Output: Returns the maximum set B of branching vertices

```
1  begin
2      B_d ← GetBranches_d(G, r, O_0)
3      if B_d = ∅ then
4          return ∅
5      else
6          v ← the greatest vertex in B_d w.r.t. the ordering in O_0
7          B_s ← {u|u ∈ V, u ≤ v w.r.t. O_0}
8          if |B_d|/|B_s| < α then
9              return B_d
10         else
11             return B_s
```

3.9 LMC: Large MaxClique

MoMC algorithm has been designed for little and dense graphs, it can be adapted also for large and sparse graphs, by changing the memory graph representation and performing some preprocessing to cut some vertices that do not lead to the maximum clique. For this reason, a new algorithm was introduced it is called LMC. The preprocessing is performed by a procedure called *Initialize* it performs:

- Derive a vertex ordering for search
- Find an initial clique
- Reduce the input graph

This procedure is described in Algorithm 25.

Is it important to remark that the complexity of this procedure is $O(|E|)$ with respect to that one of BBMCSP which is $O(\Delta(G)|E|)$, where $\Delta(G)$ is the maximum degree of the graph.

The procedure *Initialize* is called not only when preprocessing the graph but also for each vertex of the first level subtree (recursion tree). LMC Uses an adjacency list to store all the graphs in main memory, then for each subgraph $G[N(G, v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_{|V|}\}]$ reduced by the *Initialize* procedure it computes an adjacency matrix to speedup vertex access operations. The main procedure is described in Algorithm 26.

This algorithm has been compared with BBMCSP and it is outperformed most instances.

Algorithm 25: *Initialize*: A preprocessing for large and sparse graph [6]

Input : Graph G, Int lb
Output: Returns an initial Clique C_0, a core number of G, a reduced graph G' of G, and an initial vertex ordering O'

```
1  begin
2    Sort V in increasing degree ordering
3    cur_core ← deg(v_1)
4    for i = 1 to |V| do
5      if deg(v_i) > cur_core then
6        cur_core ← deg(v_i)
7      core_number[v_i] ← cur_core
8      if deg(v_i) = |V| − i then
9        for j = i + 1 to |V| do
10         core_number[v_j] ← cur_core
11       break
12     foreach v ∈ (N(G, v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|}} do
13       deg(v) ← deg(v) − 1
14       Move v and re-index vertices {v_{i+1}, v_{i+2}, ..., v_{|V|}} for
           keeping the incresing degree ordering
15   C_0 ← {v_{i+1}, v_{i+2}, ..., v_{|V|}}
16   max_core ← the maximum core number in vertices of V
17   if |C_0| > lb then
18     lb ← |C_0|
19   G' ← G[{v ∈ V : k(v_i) < lb}]
20   O_0 ← the ordering in which the core number of each vertex is
       computed
21   return (C_0, max_core, G', O_0)
```

Algorithm 26: *LMC*: A BnB algorithm for computing the max clique [6]

Input : Graph G
Output: Returns the maximum clique

```
1  begin
2      (C_0, k(G), G', O_0) ← Initialize(G, 0)
3      if |C_0| = k(G) + 1 then
4          return C_0
5      C_max ← C_0
6      V' ← the vertex set of G'
7      Order V' w.r.t the initial ordering O_0
8      for i = |V'| down to 1 do
9          P ← N(G, v_i) ∩ {v_{i+1}, ..., v_{|V'|}}
10         (C'_0, k(G[P]), G'', O'_0) ← Initialize(G[P], |C_max| − 1)
11         if |C'_0| ≥ |C_max| then
12             C_max ← C'_0 ∪ {v_i}
13         if k(G[P]) + 1 ≥ |C_max| then
14             Construct the adjacency matrix for G''
15             C' ← SearchMaxClique(G'', C_max, {v_i}, O'_0)
16             if |C'| > |C_max| then
17                 C_max ← C'
18     return C_max
```

Algorithm 27: *SearchMaxClique*: A BnB algorithm for computing the max clique greater than $|C_{max}|$ [6]

Input : Graph G, Set C_{max}, Set C, Ordering O
Output: Returns the maximum clique

```
1  begin
2      if |V| = 0 then
3          return C
4      B ← GetBranches_m(G, |C_max| − |C|, O)
5      if B = ∅ then
6          return C_max
7      A ← V \ B
8      Let B = {b_1, b_2, ..., b_|B|} in increasing oredering w.r.t. O
9      for i = |B| downto 1 do
10         P ← N(G, b_i) ∩ ({b_{i+1}, b_{i+2}, ..., b_|B|} ∪ A)
11         C′ ← SearchMaxClique(G[P], C_max, C ∪ {b_i}, O)
12         if |C′| > |C_max| then
13             C_max ← C′
14     return C_max
```

3.10 Multithreaded LMC

LMC seems to be the fastest single-thread algorithm for Maximum Clique computation, from the paper MoMC might be parallelized so even LMC should be. Different attempts have been performed with good results. It might be parallelized by performing in parallel the search at the first level of the recursion tree (Algorithm 28).

The results show a near-linear speedup over the number of threads on locally sparse instances.

3.11 Chang: Maximum Clique Branch-Reduce and Bound

The most recently designed algorithm is MC-BRB (MaxClique Branch-Reduce and Bound) it is designed to solve large sparse graphs but locally dense. For those kinds of graphs, MC-BRB outperforms any other previous algorithm. MC-BRB is the new main search procedure in which it maps an instance of k-clique finding for a dense subgraph on a maximum clique search for a large sparse graph. Chang also designed a greedy algorithm able to find a near-maximum clique in polynomial time. The latter procedure is called MC-EGO it both finds an initial clique and an upper bound

Algorithm 28: *MLMC*: A BnB algorithm for computing the max clique

Input : Graph G
Output: Returns the maximum clique

```
1  begin
2      (C_0, k(G), G', O_0) ← Initialize(G, 0)
3      if |C_0| = k(G) + 1 then
4          return C_0
5      C_max ← C_0
6      V' ← the vertex set of G'
7      Order V' w.r.t the initial ordering O_0
8      for i = |V'| down to 1 do in parallel
9          P ← N(G, v_i) ∩ {v_{i+1}, ..., v_{|V'|}}
10         (C'_0, k(G[P]), G'', O'_0) ← Initialize(G[P], |C_max| − 1)
11         if |C'_0| ≥ |C_max| then
12             C_max ← C'_0 ∪ {v_i}
13         if k(G[P]) + 1 ≥ |C_max| then
14             Construct the adjacency matrix for G''
15             C' ← SearchMaxClique(G'', C_max, {v_i}, O'_0)
16             if |C'| > |C_max| then
17                 C_max ← C'
18     return C_max
```

used to reduce the memory requirements. The speedup is archived thanks to its "reduce" technique, this technique is applied when the graph is locally dense, and it can be applied when the degree of the subgraph is quite the same large as the length of the set. The reducing technique discards vertices from the solution and thanks to lemmas and theorems proves that there exists a maximum clique of that dimension. The authors also say that the algorithm can be further improved by adapting the PMaxSAT pruning strategy of LMC. As said the reducing technique is composed of reduction rules, those reduction rules exploit some lemmas, the reduction rules state the following: Given an integer k the instance of MC-BRB search for the existence of a k-clique on a subgraph induced by the first level subtree where k is the dimension of the maximum clique found so far $k = |C_{max}| - |C| + 1$.

- **Low Degree Reduction Rule:** if the degree of the vertex $deg(u) < k - 1$ means that the vertex u cannot belong to a maximum clique greater than k so can be discarded from the possible vertices that can form a larger clique.

- **All Connection Reduction Rule:** if $deg(u) = |V(G)| - 1$ then u is a vertex that for sure belongs to the maximum clique so can be removed from G and decrease k by 1.

- **One Neighbour Missing Reduction Rule:** if $deg(u) = |V(G)| - 2$ then given the only missing neighbour of u, u_1 the latter does not belong to the maximum clique instead u yes. So we can safely remove both from the resulting graph and decrease k by one.

- **Two Neighbours Missing Reduction Rule:** if $deg(u) = |V(G)| - 3$ then let u_1, u_2 the two not neighbors of u, there exist two conditions: if $(u_1, u_2) \notin E(G)$ then u_1 and u_2 cannot belong to the maximum clique so we can discard $\{u, u_1, u_2\}$ and decrease k by 1. Otherwise if $(u_1, u_2) \in E(G)$ then the maximum clique includes either u or $\{u_1, u_2\}$ so the proposed technique suggest to discard $\{u, u_1, u_2\}$ and adding a super vertex obteined by contracting $\{u_1, u_2\}$, $u_{1,2}$, the contraction operation keeps the edges of u_1 and u_2, finally we can decrease k by 1.

- There is another rule **Three Neighbours Missing Reduction Rule:** It can be applied following the baseline of the precedent rule, then let u_1, u_2, u_3 the three missing neighbors the application depends on how many edges the subgraph $G[\{u_1, u_2, u_3\}]$ has.

All those rules can be applied one after another to the potential set until there are no changes. The algorithm has been parallelized over the first level independent subtree, and archives linear speedup over the number of threads. The algorithm performs less level of recursion with the result of a more balanced approach (when running over locally dense graphs), this

kind of algorithm is much more difficult to parallelize with the technique shown by McCreesh since it requires to modify the subgraph by contracting vertices and substituting new ones, with the result of much heavy context to donate.

3.12 Van Copernolle: BBMCG

The only parallel implementation for the maximum clique search existing on GPU is explained in the paper "*Maximum Clique Solver on GPU using bitsets*", the author M.Van Copernolle inspired by other work decided to parallelize BBMC the algorithm of San Segundo on GPUs since the Maximum clique problem is very unbalanced, the search tree is unbalanced, threads in a block that explores the tree cannot execute different path in the search tree, all the different path can be executed serially, causing warp divergence, since GPU threads are grouped in blocks, it decides to use two levels of parallelism to allow to traverse the search space, Blocks of threads are used to traverse the search tree in parallel, each block can take different paths, threads inside the Block are employed to compute in parallel bitset and reduce operation, this kind of parallelization reveals advantageous for two reasons:

- The context is per Block instead of per thread with the result of less memory requirements
- Reduced warp divergence: a group of threads inside the warp takes the same path over the search tree.

The author shows a significant speedup with respect to the CPU single-thread version.

3.13 Almasri: Parallel MCE on GPUs

Many attempts have been made to bring the Bron-Kerbosch algorithm to GPUs, the best performing one is explained in the paper "*Parallelizing Maximal Clique Enumeration on GPUs*". The result shows a significant speedup over the parallel CPU counterpart, proving that the GPU hardware can significantly help in accelerating this kind of algorithm, Alsmari takes its idea of parallelization from BBMCG, where it uses blocks to traverse the search tree in parallel, it further implements the concept of work donation inspired by the Multithreaded state-of-the-art maximum clique solver of McCreesh, with a little difference: to keep each thread busy and reduce the context of donation, it implements this helping strategy once the first level subtree is exhausted, and instead of using a shared queue of work to donate, uses a queue of idle blocks, where the work is donated to idle bloks once dequeued

from the queue. It also shows that the problem of the MCE can gain a little performance if the thread task is parallelized over the second level subtree, in this way the work done by each block is more balanced. Since it provides also the source code, this will be our starting point for parallelizing the Maximum Clique on GPU.

Chapter 4

GPU parallel implementation

This chapter describes how the current parallelized GPU implementation works, describing its implementation details and drawbacks.

4.1 From MCE to MCP

As seen in the previous section the Bron-Kerbosch algorithm can be modified and adapted to the maximum clique algorithm. All the state-of-the-art maximum clique solvers use the graph coloring to prune the search, this is the main difference with respect to the Bron Kerbosh algorithm. The maximum clique solvers are subdivided into two main categories, depending on the size of the input problem.

- For dense and little networks MCP algorithm employs the **adjacency matrix** representation of the input graph.
- For sparse and large networks MCP algorithms employs the **edge list** representation of the input graph

The adjacency matrix is a graph representation in memory, it is a matrix A where each element $a_{i,j} \in \{0, 1\}$. This kind of data structure has advantages and disadvantages, the main disadvantage is that it occupies in memory $O(|V|^2)$ space the space can be reduced by a factor of 8 if we use bit-vectors or bitstring. The advantages came in terms of performance where the operation of checking whenever $(v_i, v_j) \in E$ is $O(1)$. Also, the operation of accessing the neighborhood is $O(1)$ because it is reduced to access just the row of the selected vertex. The edge list instead is employed when the graph is too big and its adjacency matrix does not fit in memory. The adjacency list is a couple of vectors a, b where an element of the vector a a_i, is the offset of the vertex i in b. Its space complexity is $O(|V| + |E|)$. If

we want to access the neighbor of a vertex i it can be done in $O(1)$ but if we want to check whenever an edge $(v_i, v_j) \in E$ it requires time $O(\Delta(G))$. The MCP solver for large and sparse networks also uses a preprocessing strategy, which aims to reduce the input graph and determine a vertex ordering. The preprocessing strategy is solved in three steps:

- Derive a vertex ordering usually degeneracy one as for Bron Kerbosh
- Determine an initial solution as big as possible $\omega_0(G)$.
- Reduce the input graph based on the initial solution found.

These three steps can be done in linear time complexity so they are usually faster with respect to the main search procedure. The main search procedure is a little bit different for little dense or large sparse networks. In the MCP solver for the little and dense networks the algorithm receives just the color pruning strategy, instead for the large and sparse network the main search procedure is explicitly separated by levels of recursion, for the first level subtree, it uses the same Bron-Kerbosch optimizations plus additional pre-processing procedure, first we compute the greedy coloring of the induced subgraph $G'[N^+(v_i)]$ where G' is the subgraph reduced by the preprocessing steps, the greedy coloring aims to compute an upper bound of the maximum clique if the upper bound is little that the current maximum solution found so far, it will cut the corresponding branch, next it will perform k-core decomposition to both compute the upper an upper bound and further reduce the induced subgraph G'. Finally, the same search procedure for little and dense graphs is launched.

4.2 Parallel MCP for large and sparse graphs

As just said we started from the code of the paper "Parallelizing Maximal Clique Enumeration on GPUs", so as for the Bron Kerbosh algorithm the MCP solver computes an initial ordering: the algorithm to compute the ordering has been already presented in the previous section in sequential fashion order. It is already implemented in the code in its parallel version is described in Algorithm 29.

The algorithm makes use of two general Queues Q_{bucket} and $Q_{current}$, those two queues are filled with vertices. The Q_{bucket} stores in general vertices whose degree is between a certain range (see the algorithm) instead of $Q_{current}$ stores the vertices with the same degree which will be removed from the graph in the current iteration. We do not modify the graph, since the only information needed is the degree we update the degree of neighbors once the vertices have been removed, this operation must be performed in an atomic transaction.

Algorithm 29: Parallel k-cores decomposition

Input : Graph $G = (V, E)$

Output: Returns a vertex ordering O, and the core number

```
1  begin
2      O ← ∅
3      deg() ← Compute the degrees of each vertex v ∈ V
4      current_core ← min degree of vertex in V
5      Q_bucket ← ∅
6      bucket_level_size ← 0
7      todos ← |V|
8      while todos > 0 do
9          if Q_backet = ∅ then
10             bucket_level_size ← bucket_level_size + 128
11             Q_bucket ← {v ∈ V | current_core ≤ deg(v) <
                 current_core + backet_level_size}
12         Q_current ← {v ∈ Q_backet | current_core = deg(v)}
13         while Q_current ≠ ∅ do
14             Q_bucket ← Q_bucket \ Q_current
15             todos ← todos − |Q_current|
16             Q_next ← ∅
17             /* Warp-wise parallel                                  */
18             for v_i ∈ Q_current do in parallel
19                 append({v_i}, O)
20                 core(v_i) ← deg(v_i)
21                 /* Thread-wise parallel                            */
22                 for v_j ∈ N(G, v_i) do in parallel
23                     deg(v_j) ← deg(v_j) − 1
24                     if deg(v_j) = current_core then
25                         Q_next ← Q_next ∪ {v_j}
26                         if current_core ≤ deg(v_j) <
                             current_core + bucket_level_size then
27                             Q ← Q_bucket ∪ {v_j}
28             swap(Q_current, Q_next)
29         current_core ← current_core + 1
30     return (O, core(), current_core − 1)
```

Algorithm 30: Find Heuristic Clique

Input : Graph $G = (V, E)$, int max_core
Output: Returns an initial clique

```
1  begin
2      Q_current ← {v ∈ V | max_core = deg(v)}
3      deg() ← compute vertices degrees of G[Q_current]
4      while Q_current ≠ ∅ do
5          u ← min degree vertex from Q_current
6          if |Q_current| = deg(u) + 1 then
7              C_0 ← Q_current
8              break
9          Q_current ← Q_current \ {u}
10         for u_i ∈ N(G, u) ∩ Q_current do in parallel
11             deg(u_i) ← deg(u_i) − 1
12     return C_0
```

This algorithm has been extended to compute also the initial solution $\omega_0(G)$ (Algorithm 30):

This algorithm is an adaptation to the one used in the pre-processing stage for LMC. After both the ordering is computed, and an initial clique is found the input graph G is shrunk and ordered in degeneracy order.

4.3 MCP Solver for large sparse graphs

The main search procedure is then applied to G' where $G' = G[\{v \in V | core(v) + 1 \geq |C_0|\}]$, Where C_0 is the initial solution. The algorithm is described in Algorithm 31.

The algorithm is running on GPU and starts with a *parallel for* over GPU blocks. The intersection in Line 4 can be pre-computed in the preprocessing stage and is embedded in the CSR (Con Sparse Row) representation of G. Next, to allow the usage of bitset we compute the adjacency matrix of the induced subgraph $G[P]$ this operation is done with a sub-level of parallelism *warp-parallel* the algorithm has already been implemented and found on the starting code. The Line 6 computes the branching vertices B that are by definition $B = \{v \in P | color(v) > k_{min}\}$, the algorithm is described in Algorithm 32. The algorithm 32 is really simple it performs coloring following the algorithm of San Segundo where all the operations inside the bitset are in parallel when each thread ith performs the corresponding operation in the bitset block ith. Coming back to the Algorithm 31 after the coloring procedure is performed if the set B is empty we cut the branch by color.

Algorithm 31: Pseudo-code for the parallel MC algorithm. Reported the first-level independent subtree.

```
Input: Graph G = (V, E), Set C_0, Order O_0
Output: Maximum clique C_max
1  begin
2      C_max ← C_0
3      for v_i ∈ V with respect to the order O_0 do in parallel
4          P = Γ(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|}}
5          Compute adjacency matrix of G[P]
6          B ← Color(G[P], |C_max| − 1)
7          if B = ∅ then
8              continue
9          (O, core()) ← blockParallelKCoreDecomposition(G[P])
10         if max(core()) + 1 ≤ |C_max| then
11             continue
12         P′ = {u ∈ P|core(u) + 1 ≥ |C_max|}
13         Sort P′ based with respect to the order O
14         Compute adjacency matrix of G[P′]
15         SearchMaxClique(G[P′], P′, {v_i}, C_max, O)
16     return C_max
```

Algorithm 32: Color: Assign a color to a set of vertices P

```
Input  : Graph G = (V, E), Int k_min
Output: Returns the set of branching vertices B
1  begin
2      Bitset B ← V
3      k = 1
4      while B ≠ ∅ and k ≤ k_min do
5          Bitset C ← B
6          while C ≠ ∅ do
7              select the first vertex v ∈ C
8              B ← B \ {v}
9              C ← C \ {v}
10             C ← C ∩ N̅(G, v)
11         k ← k + 1
12     return B
```

Next, we will perform the k-core decomposition: this procedure is described in Algorithm 33 which by differences with respect to the Preprocessing stage is shorter and uses just two queues $Q_{current}$ and Q_{next} this algorithm uses as the previous in preprocessing stage two level of parallelism the external for by warps and the internal by threads inside the warp.

Algorithm 33: Block Parallel k-cores decomposition

Input : Graph $G = (V, E)$
Output: Returns a vertex ordering O, and the core number

```
1  begin
2      O ← ∅
3      deg() ← Compute the degrees of each vertex v ∈ V
4      current_core ← min degree of vertex in V
5      todos ← |V|
6      while todos > 0 do
7          Q_current ← {v ∈ V | current_core = deg(v)}
8          while Q_current ≠ ∅ do
9              todos ← todos − |Q_current|
10             Q_next ← ∅
11             /* Warp-wise parallel                         */
12             for v_i ∈ Q_current do in parallel
13                 append({v_i}, O)
14                 core(v_i) ← deg(v_i)
15                 /* Thread-wise parallel                   */
16                 for v_j ∈ N(G, v_i) do in parallel
17                     deg(v_j) ← deg(v_j) − 1
18                     if deg(v_j) = current_core then
19                         Q_next ← Q_next ∪ {v_j}
20             swap(Q_current, Q_next)
21         current_core ← current_core + 1
22     return (O, core())
```

Coming back to the Algorithm 31 after the k-core decomposition procedure is performed we can check if the max core which is an upper bound of the maximum clique is less than the maximum clique found so far then we can cut the branch by k-core. In our implementation, the P' set is computed inside the k-core decomposition phase, furthermore, it is already ordered with respect to the ordering O, but we have to make these steps explicit in the algorithm. Finally the Search procedure in Line 15 takes place.

The main recursive procedure is described in Algorithm 34. This proce-

Algorithm 34: Our parallel MC computation.

Input: Graph $G = (V, E)$, Set P, Set C, Shared Set C_{max}, Ordering O
Output: Maximum clique C_{max}

```
1  begin
2      if P = ∅ then
3          if |C| > |C_max| then
4              C_max ← C
5          return C_max
6      B ← Color(P, |C_max| − |C|);
7      if B = ∅ then
8          return C_max
9      A ← P \ (B = {b_1, b_2, ..., b_|B|})
10     for b_i ∈ B with respect to the ordering in O do
11         P' ← A ∪ (Γ(b_i) ∩ {b_{i+1}, b_{i+2}, ..., b_|B|})
12         SearchMaxClique(G, P', C ∪ {b_i}, C_max);
13     return C_max
```

dure starts by checking if the P set is empty this is the stopping condition that corresponds if the clique cannot further grow, if it is greater than the maximum one, we overwrite the current maximum found so far and then return to the previous level. Next, the procedure to compute the branching vertices takes place in Line 6 it is the same as described in Algorithm 32 so it not will be further commented. Next, the set A will be computed from P and B, next with a bitset scan we loop over the set B searching for a bigger clique than the incumbent C, so the set P' is computed and a recursive call to *SearchMaxClique* is performed. The described algorithm performs the search without the optimization of the state-of-the-art parallel maximum clique solver proposed by McCreesh, this optimization is suited and effective for a dense graph or for a graph that has a clique of big dimensions, McCreesh's variant cannot be applied directly to the sparse case. Since the adjacency matrix will be different for each first-level induced subgraph, performing work donation could lead to higher runtimes because most of the time donation time will be employed for copying the context between threads or blocks. But since we would keep load balancing between threads, we can demand the procedure of donation after all the induced subgraphs are created and then cannot change, it will be reduced just by donating the correct induced subgraph memory location. The changes in the algorithm are described in Algorithm 35 and 36.

The donation starts in Algorithm 35 where the blocks that end the first level subtree enqueue their *blockIdx* and go in a wait state. Please pay attention that the wait is not infinite they go in a waiting state for a short time. The currently running threads in Algorithm 34 when the first level subtree

Algorithm 35: Pseudo-code for the parallel MC algorithm. Reported the first-level independent subtree.

Input: Graph $G = (V, E)$, Set C_0, Order O_0

Output: Maximum clique C_{max}

1. **begin**
2. $C_{max} \leftarrow C_0$
3.   **for** $v_i \in V$ *with respect to the order* O_0 **do in parallel**
4. $P = \Gamma(v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_{|V|}\}$
5.   Compute adjacency matrix of $G[P]$
6. $B \leftarrow$ Color($G[P]$, $|C_{max}| - 1$)
7.   **if** $B = \emptyset$ **then**
8.   **continue**
9. $(O, core()) \leftarrow$ blockParallelKCoreDecomposition($G[P]$)
10.   **if** $max(core()) + 1 \leq |C_{max}|$ **then**
11.   **continue**
12. $P' = \{u \in P | core(u) + 1 \geq |C_{max}|\}$
13.   Sort P' based with respect to the order O
14.   Compute adjacency matrix of $G[P']$
15.   SearchMaxClique($G[P']$, P', $\{v_i\}$, C_{max}, O)
16.   `/* New Section below: */`
17.   **while** *!QueueFull()* **do**
18.   Enqueue on idle blocks queue
19.   **while** *!WorkReady and !QueueFull()* **do**
20.   Wait()
21.   **if** *WorkReady* **then**
22.   WorkReady = false
23.   Setup stack for block
24. $P' \leftarrow$ Computed by the donor block
25. $C \leftarrow$ Computed by the donor block
26.   SearchMaxClique($G[P']$, P', C, C_{max}, O)
27.   **return** C_{max}

Algorithm 36: Our parallel MC computation with the donation.

```
   Input: Graph G = (V, E), Set P, Set C, Set C_max, Ordering O
   Output: Maximum clique C_max
 1 begin
 2   if P = ∅ then
 3     if |C| > |C_max| then
 4       C_max ← C
 5     return C_max
 6   B ← Color(P, |C_max| − |C|)
 7   if B = ∅ then
 8     return C_max
 9   A ← P \ (B = {b_1, b_2, ..., b_|B|})
10   for b_i ∈ B with respect to the ordering in O do
11     /* Start new section:                                    */
12     if first level subtree explored then
13       D ← {b ∈ B|b < b_i with respect to the order O}
14       if blockList ← QueueDequeue(|D|) then
15         for d_j ∈ D with respect to the order in O do
16           P′ ← A ∪ (Γ(d_j) ∩ {d_{j+1}, d_{j+2}, ..., d_|B|})
17           Donate P′ and C to jth block in blockList
18           Set WorkReady to true for jth block in blockList
19         break
20     /* End new section:                                      */
21     P′ ← A ∪ (Γ(b_i) ∩ {b_{i+1}, b_{i+2}, ..., b_|B|})
22     SearchMaxClique(G, P′, C ∪ {b_i}, C_max);
23   return C_max
```

tries to dequeue $|D|$ blocks from the queue if those blocks are available to start the donation, it donates a work $|C| + 1$ level subtree for each branching vertices in D. The donating threads set *WorkReady* variable for the jth block dequeued from the *blockQueue*. At the end when all block goes out from the first level subtree and enqueue their index on the *blockQueue* the termination condition becomes false and all blocks terminate its execution.

4.4 Warp-wise-parallel version

The block parallel version is not the only developed parallel algorithm, as we can see since most of our graphs are locally little small $V(G[P'])$ is it possible to go deep down to one level of parallelism, then reimplement all the work done per block on warps. This solution can be enhanced since for a certain graph the first level subtree can slow down the operation of Coloring and k-core decomposition, then an enhanced approach that uses warps just from the second level subtree has been developed the warp version allows for the second level subtree inter block warp work donation as for the block parallel version. and since warps are in general more than the maximum number of blocks (about 2x) this solution can speed up in certain cases of over 2x the previous solution, this is because the occupancy increases with respect to the block parallel version. Since the bitset operation is constrained by the fact that sometimes the bitset dimension is less than the number of threads on the block, threads do not perform any job and remain inactive, by employing warps we can increase the number of used threads by a factor of two. However, this kind of parallelization strategy uses a greater amount of memory than the block one. But the good news is that since the warp-wise parallelism is from the second level subtree we can share the induced subgraph adjacency between warps without spending time in recomputing it per warp, so less memory usage with increased external parallelism.

4.5 Pruning strategies

As seen before there exists more than one pruning strategy. During the algorithm development, all those pruning strategies have been implemented and tested so compared. They can be always faster on CPU but unfortunately not so simple on GPUs. Among the chosen pruning strategies we remember:

- NUMBER
- Re-NUMBER
- San Segundo Color
- Re-Color just used in BBMCR

- Coloring + Reduce

All those pruning strategies have been adapted to the maximum clique solver: NUMBER and Re-NUMBER use more memory than the San Segundo's coloring strategy because it stores all the computed independent sets. Also, Re-Color needs to store the computed independent sets but as bitsets so occupy less memory, since often for dense subgraph independent sets are usually sparse they both waste a lot of memory and increase the access time from memory. so the solution would be to use *Watched Bitset* that keeps track by two indices of the non-empty block region of the bitset, furthermore, to avoid further access to the global one we can use shared memory to store neighborhoods of selected nodes, this mechanism increases shared memory utilization so reduce L1 cache but should decrease the runtimes because of the fast access. The last pruning strategy is an adaptation of the Reduce pruning strategy from MC-BRB. We cannot use all the Lemmas proposed by MC-BRB because most of them require modifying the graph by contracting and removing nodes, but we can exploit some of them to improve pruning, and discard at most vertex that does not belong to a maximum clique. The lone drawback is that computing the degrees for each node of the recursion tree requires access to the entire graph so can lower the runtimes for memory speed issues.

Chapter 5

Experimental Analisys

This chapter describes the setup used to test our parallelized algorithms, it shows how the algorithms scale over the number of threads on the CPU. Finally, test the developed GPU implementation by comparing it with the existing parallelized state-of-the-art CPU Solvers explaining why and where the developed implementation (GPU) can outperform the CPU one. The experiments and the related software is available on Github[1].

5.1 Experimental Setup

The experiments have been done with a machine with: Intel Core-i9 10900KF with 64GBs of RAM the machine is equipped with an NVIDIA RTX 3070 GPU with 8GBs GDDR5 VRAM. All our tools were compiled with g++ and nvcc, the NVIDIA toolkit installed is version 12.2. All the testing library is written in Python and provides methods to collect statistics.

5.2 Dataset

All the tested instances were taken from the network repository [15] [2]. The dataset is listed in Table 5.1. All the statistics have been computed by a tool written in C++, and collected by standard output by the OS python library. The dataset has been chosen to represent the majority of network types in our world, they all have different characteristics and properties that allow us to show when the GPU approach is effective.

With another custom tool written in C++ random graphs have been generated by varying density and number of nodes, It has generated random graphs of 250, 1,000, 3,000, 10,000, and 100,000 nodes respectively. The graph with 250 nodes has been tested with higher density to show how the

[1] https://github.com/salvatore-dimartino/mcp-gpu
[2] https://networkrepository.com

Instance	$\|V\|$	$\|E\|$	density	max. degree	avg. degree	$\|\omega(G)\|$
co-papers-dblp	540,486	15,245,730	0.000104	3,299	56	337
web-uk-2002-all	18,520,486	298,113,763	2e-06	194,956	32	944
c-62ghs	41,731	300,538	0.000345	5,061	14	2
web-it-2004	509,338	7,178,414	5.5e-05	469	28	432
aff-orkut-user2groups	8,730,857	327,037,488	9e-06	318,268	75	6
rec-yahoo-songs	136,737	49,770,696	0.005324	31,431	728	19
soc-livejournal-user-groups	7,489,073	112,307,386	4e-06	1,053,749	30	9
socfb-konect	59,216,214	92,522,018	0	4,960	3	6
aff-digg	872,622	22,624,728	5.9e-05	75,715	52	32
soc-orkut	2,997,166	106,349,210	2.4e-05	27,466	71	47
soc-sinaweibo	58,655,849	261,321,072	0	278,491	9	44
wiki-talk	2,394,385	5,021,411	2e-06	100,032	4	26
bn-human-Jung2015_M87113878	1,772,034	76,500,873	4.9e-05	6,899	86	227
bn-human-BNU_1_0025864_session_2-bg	1,827,241	133,727,517	8e-05	8,444	146	266
soc-flickr	513,969	3,190,453	2.4e-05	4,369	12	58
tech-p2p	5,792,297	147,830,699	9e-06	675,080	51	178
bn-human-BNU_1_0025864_session_1-bg	1,827,218	143,158,340	8.6e-05	15,114	157	293
soc-flickr-und	1,715,255	15,555,043	1.1e-05	27,236	18	98
socfb-A-anon	3,097,165	23,667,395	5e-06	4,915	15	25
bn-human-Jung2015_M87126525	1,827,241	146,109,301	8.8e-05	8,009	160	220
bio-human-gene1	22,283	12,345,964	0.049731	7,940	1,108	1,335
bio-human-gene2	14,340	9,041,365	0.087942	7,230	1,261	1,300
soc-LiveJournal1	4,847,571	68,475,392	6e-06	22,887	28	321
web-wikipedia_link_it	2,936,413	104,673,034	2.4e-05	840,650	71	870
web-indochina-2004-all	7,414,866	194,109,312	7e-06	256,425	52	6,848

Table 5.1: The table shows the statistics about our dataset, i.e., number of vertices, number of edges, density, maximum and average degree of the nodes, and the size of maximum clique of the graph.

GPU approach behaves at those densities. Then for each chosen density and number of nodes it has been generated 10 different instances.

5.3 Parallel CPU implementations

This section shows how the parallelized counterpart performs over the selected dataset of real-world graphs.

5.3.1 BBMCPara

Starting with BBMCPara we run our dataset the following tables show the actual runtimes by varying the number of threads you will see the corresponding speedup. BBMCSP has been parallelized following the guideline in its paper since it is targeted just for sparse graphs a linear parallelization has been employed. As described in its paper BBMCPara uses the OpenMP library for multithreading. Runtimes are reported in Tables 5.2 and 5.3.

5.3.2 LMC

This section shows how LMC performs by varying the number of threads, Tables 5.4 and 5.5 show two important statistics runtime and the corresponding speedup. LMC has been parallelized as shown in the previous

	1 Threads		2 Threads		4 Threads	
Instance	Search Time	Speed-up	Search Time	Speed-up	Search Time	Speed-up
aff-digg	498.929	1.0	305.82	1.631	161.96	3.081
aff-orkut-user2groups	959.511	1.0	515.918	1.86	320.924	2.99
bio-human-gene1	T.O.	-	T.O.	-	T.O.	-
bio-human-gene2	T.O.	-	T.O.	-	1767.08	-
bn-human-BNU_1_0025864_session_1-bg	T.O.	-	T.O.	-	T.O.	-
bn-human-BNU_1_0025864_session_2-bg	359.28	1.0	202.085	1.778	100.395	3.579
bn-human-Jung2015_M87113878	27.328	1.0	13.899	1.966	7.346	3.72
bn-human-Jung2015_M87126525	865.677	1.0	503.263	1.72	319.573	2.709
c-62ghs	0.012	1.0	0.008	1.495	0.005	2.326
co-papers-dblp	-	-	-	-	-	-
rec-yahoo-songs	728.392	1.0	375.84	1.938	194.15	3.752
soc-LiveJournal1	0.01	1.0	0.014	0.727	0.016	0.607
soc-flickr	2.057	1.0	1.398	1.472	0.763	2.696
soc-flickr-und	58.753	1.0	41.184	1.427	25.488	2.305
soc-livejournal-user-groups	496.203	1.0	257.73	1.925	161.903	3.065
soc-orkut	37.044	1.0	18.613	1.99	9.838	3.765
soc-sinaweibo	71.936	1.0	37.079	1.94	20.31	3.542
socfb-A-anon	6.597	1.0	3.352	1.968	1.811	3.643
socfb-konect	0.642	1.0	0.338	1.902	0.18	3.571
tech-p2p	T.O.	-	T.O.	-	T.O.	-
web-indochina-2004-all	0.136	1.0	0.079	1.725	0.037	3.641
web-it-2004	-	-	-	-	-	-
web-uk-2002-all	-	-	-	-	-	-
web-wikipedia_link_it	0.021	1.0	0.021	1.003	0.022	0.99
wiki-talk	0.355	1.0	0.196	1.814	0.108	3.302

Table 5.2: Runtimes of BBMCPara algorithm over increasing number of threads

	8 Threads		10 Threads		20 Threads	
Instance	Search Time	Speed-up	Search Time	Speed-up	Search Time	Speed-up
aff-digg	90.602	5.507	78.632	6.345	55.898	8.926
aff-orkut-user2groups	306.986	3.126	285.404	3.362	O.O.M	-
bio-human-gene1	T.O.	-	T.O.	-	T.O.	-
bio-human-gene2	795.261	-	510.788	-	289.05	-
bn-human-BNU_1_0025864_session_1-bg	T.O.	-	T.O.	-	T.O.	-
bn-human-BNU_1_0025864_session_2-bg	54.288	6.618	47.667	7.537	38.278	9.386
bn-human-Jung2015_M87113878	4.055	6.739	3.412	8.01	2.498	10.939
bn-human-Jung2015_M87126525	178.843	4.84	158.499	5.462	128.318	6.746
c-62ghs	0.004	3.098	0.004	2.908	0.025	0.493
co-papers-dblp	-	-	-	-	-	-
rec-yahoo-songs	104.557	6.966	92.986	7.833	81.467	8.941
soc-LiveJournal1	0.015	0.642	0.014	0.703	0.011	0.906
soc-flickr	0.511	4.026	0.419	4.908	0.225	9.155
soc-flickr-und	17.127	3.43	13.52	4.346	9.575	6.136
soc-livejournal-user-groups	129.463	3.833	123.021	4.033	120.697	4.111
soc-orkut	5.685	6.516	4.915	7.537	3.461	10.702
soc-sinaweibo	13.296	5.41	11.946	6.022	10.617	6.776
socfb-A-anon	1.029	6.409	0.856	7.709	0.587	11.23
socfb-konect	0.101	6.35	0.093	6.945	0.09	7.134
tech-p2p	T.O.	-	T.O.	-	T.O.	-
web-indochina-2004-all	0.021	6.602	0.017	8.142	0.014	9.838
web-it-2004	-	-	-	-	-	-
web-uk-2002-all	-	-	-	-	-	-
web-wikipedia_link_it	0.021	1.017	0.023	0.934	0.024	0.902
wiki-talk	0.058	6.108	0.048	7.438	0.032	11.135

Table 5.3: Runtimes of BBMCPara algorithm over increasing number of threads

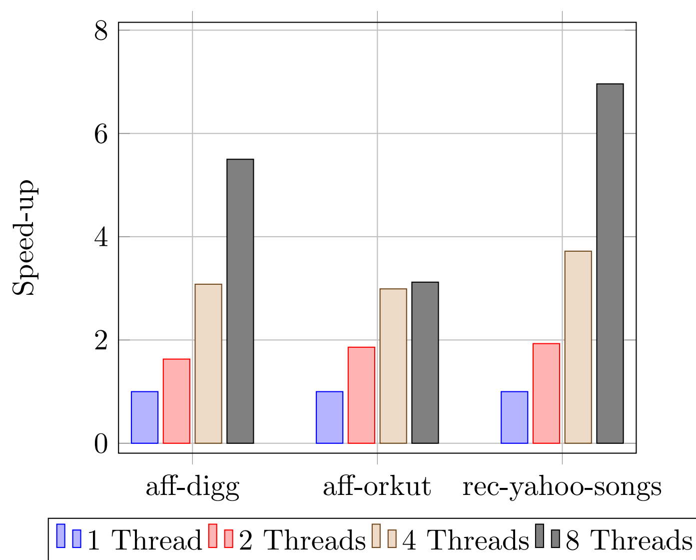


Figure 5.1: Bar chart showing speedup over the number of threads

section, since the program is written in C the POSIX thread library has been employed. Then following just a linear parallelization work can be a little unbalanced between threads if the graph produces a deep search tree.

5.3.3 MC-BRB

Also, MC-BRB has been parallelized following the guideline of its article, Both MC-EGO the algorithm running in the preprocessing stage, and MB-BRB have been linearly parallelized. The parallel versions use the OpenMP library. As the table shows it reaches a linear speedup over the number of threads. Tables 5.6 and 5.7 show the runtimes over the number of threads.

5.4 Experimental analisys

This section is subdivided into two subsections the first one focuses on random instances, we analyze the behavior of the algorithm in random instances, and the following section tests the algorithm in the real-world dataset.

5.4.1 Random Instances

All the algorithms have been tested under many conditions, this can be archived by running our tool over random instances. Random instances have been generated by us, and the results shown on the plotted line have been interpolated to create a smooth curve. First of all, we ran our instances

Instance	1 Thread		2 Threads		4 Threads	
	Search Time	Speed-up	Search Time	Speed-up	Search Time	Speed-up
aff-digg	95.55	1.0	51.93	1.84	31.75	3.009
aff-orkut-user2groups	127.98	1.0	65.52	1.953	36.12	3.543
bio-human-gene1	99.67	1.0	62.4	1.597	46.44	2.146
bio-human-gene2	52.07	1.0	37.02	1.407	28.47	1.829
bn-human-BNU_1_0025864_session_1-bg	306.85	1.0	159.33	1.926	82.31	3.728
bn-human-BNU_1_0025864_session_2-bg	266.27	1.0	131.97	2.018	69.07	3.855
bn-human-Jung2015_M87113878	141.14	1.0	71.86	1.964	39.36	3.586
bn-human-Jung2015_M87126525	143.64	1.0	72.51	1.981	37.21	3.86
c-62ghs	0.01	1.0	0.01	1.0	0.01	1.0
co-papers-dblp	0.0	-	0.0	-	0.0	-
rec-yahoo-songs	60.02	1.0	30.53	1.966	15.81	3.796
soc-LiveJournal1	0.05	1.0	0.03	1.667	0.01	5.0
soc-flickr	1.14	1.0	0.59	1.932	0.31	3.677
soc-flickr-und	12.77	1.0	6.82	1.872	3.69	3.461
soc-livejournal-user-groups	17.11	1.0	8.86	1.931	4.88	3.506
soc-orkut	12.93	1.0	7.08	1.826	3.75	3.448
soc-sinaweibo	10.43	1.0	7.73	1.349	4.11	2.538
socfb-A-anon	2.0	1.0	1.05	1.905	0.55	3.636
socfb-konect	0.09	1.0	0.1	0.9	0.1	0.9
tech-p2p	98.14	1.0	53.32	1.841	26.68	3.678
web-indochina-2004-all	36.88	1.0	17.66	2.088	9.8	3.763
web-it-2004	0.0	-	0.0	-	0.0	-
web-uk-2002-all	0.0	-	0.0	-	0.0	-
web-wikipedia_link_it	0.16	1.0	0.09	1.778	0.05	3.2
wiki-talk	0.23	1.0	0.12	1.917	0.06	3.833

Table 5.4: Runtimes of Multithreaded LMC algorithm over increasing number of threads

Instance	8 Threads		10 Threads		20 Threads	
	Search Time	Speed-up	Search Time	Speed-up	Search Time	Speed-up
aff-digg	25.1	3.807	23.24	4.111	13.71	6.969
aff-orkut-user2groups	24.05	5.321	22.45	5.701	21.25	6.023
bio-human-gene1	46.26	2.155	56.53	1.763	169.23	0.589
bio-human-gene2	19.0	2.741	17.05	3.054	27.21	1.914
bn-human-BNU_1_0025864_session_1-bg	43.81	7.004	37.53	8.176	31.52	9.735
bn-human-BNU_1_0025864_session_2-bg	35.82	7.434	30.21	8.814	24.36	10.931
bn-human-Jung2015_M87113878	22.47	6.281	19.24	7.336	16.89	8.356
bn-human-Jung2015_M87126525	19.16	7.497	15.86	9.057	12.87	11.161
c-62ghs	0.01	1.0	0.01	1.0	0.01	1.0
co-papers-dblp	0.0	-	0.0	-	0.0	-
rec-yahoo-songs	8.82	6.805	7.66	7.836	6.44	9.32
soc-LiveJournal1	0.01	5.0	0.01	5.0	0.01	5.0
soc-flickr	0.16	7.125	0.13	8.769	0.09	12.667
soc-flickr-und	2.1	6.081	1.69	7.556	1.3	9.823
soc-livejournal-user-groups	2.89	5.92	2.57	6.658	2.37	7.219
soc-orkut	2.19	5.904	1.89	6.841	1.68	7.696
soc-sinaweibo	2.4	4.346	2.58	4.043	2.5	4.172
socfb-A-anon	0.33	6.061	0.28	7.143	0.24	8.333
socfb-konect	0.12	0.75	0.12	0.75	0.12	0.75
tech-p2p	16.18	6.066	14.5	6.768	12.8	7.667
web-indochina-2004-all	5.84	6.315	4.91	7.511	4.57	8.07
web-it-2004	0.0	-	0.0	-	0.0	-
web-uk-2002-all	0.0	-	0.0	-	0.0	-
web-wikipedia_link_it	0.02	8.0	0.02	8.0	0.02	8.0
wiki-talk	0.03	7.667	0.03	7.667	0.02	11.5

Table 5.5: Runtimes of Multithreaded LMC algorithm over increasing number of threads

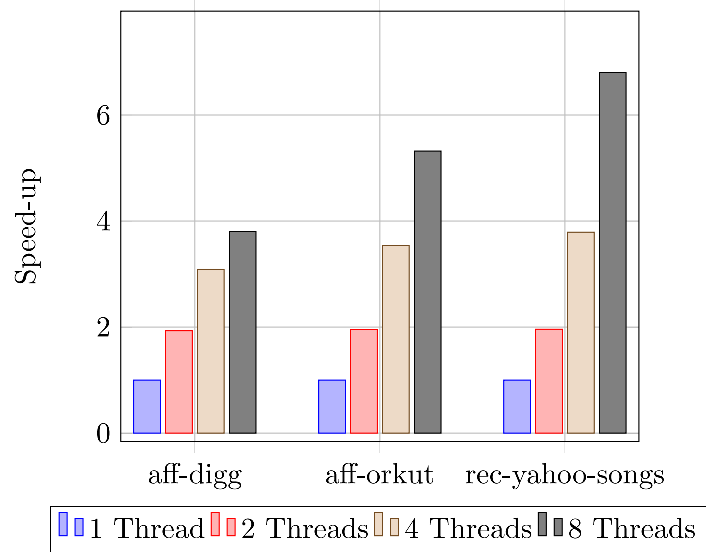


Figure 5.2: Bar chart showing speedup over the number of threads

	1 Thread				2 Threads				4 Threads			
	Runtimes		Speed-ups		Runtimes		Speed-ups		Runtimes		Speed-ups	
Instance	Init.	Search	Init.	Search	Init.	Search	Init.	Search	Init.	Search	Init.	Search
aff-digg	35.051	539.181	1.0	1.0	17.938	266.951	1.954	2.02	9.856	138.247	3.556	3.9
aff-orkut-user2groups	280.512	0.169	1.0	1.0	172.695	0.173	1.624	0.979	107.774	0.294	2.603	0.576
bio-human-gene1	28.37	12.695	1.0	1.0	14.92	6.484	1.901	1.958	8.035	3.278	3.531	3.872
bio-human-gene2	16.081	7.147	1.0	1.0	8.666	3.64	1.856	1.964	4.874	1.827	3.299	3.911
bn-human-BNU_1_0025864_session_1-bg	214.506	35.35	1.0	1.0	111.58	17.735	1.922	1.993	59.721	9.009	3.592	3.924
bn-human-BNU_1_0025864_session_2-bg	193.042	1.793	1.0	1.0	94.094	0.856	2.052	2.095	50.095	0.437	3.853	4.1
bn-human-Jung2015_M87113878	58.166	0.022	1.0	1.0	30.568	0.014	1.903	1.619	16.843	0.011	3.453	2.046
bn-human-Jung2015_M87126525	46.963	0.468	1.0	1.0	25.373	0.237	1.851	1.973	14.718	0.135	3.191	3.461
c-62ghs	0.003	-	1.0	-	0.003	-	0.998	-	0.004	-	0.979	-
co-papers-dblp	0.01	-	1.0	-	0.011	-	0.953	-	0.01	-	0.997	-
rec-yahoo-songs	86.328	78.937	1.0	1.0	45.151	39.74	1.912	1.986	25.042	20.429	3.447	3.864
soc-LiveJournal1	1.204	-	1.0	-	1.116	-	1.079	-	1.048	-	1.149	-
soc-flickr	1.261	1.487	1.0	1.0	0.702	0.754	1.795	1.972	0.407	0.381	3.1	3.908
soc-flickr-und	10.715	17.529	1.0	1.0	5.768	9.039	1.858	1.939	3.308	4.584	3.239	3.824
soc-livejournal-user-groups	38.304	0.027	1.0	1.0	23.876	0.072	1.604	0.377	17.196	0.164	2.227	0.165
soc-orkut	18.393	0.022	1.0	1.0	12.894	0.057	1.427	0.393	11.031	0.138	1.667	0.162
soc-sinaweibo	24.541	0.09	1.0	1.0	23.212	0.198	1.057	0.454	22.363	0.48	1.097	0.187
socfb-A-anon	1.565	0.004	1.0	1.0	1.337	0.011	1.17	0.324	1.29	0.022	1.213	0.168
socfb-konect	4.809	-	1.0	-	4.786	-	1.005	-	4.698	-	1.024	-
tech-p2p	78.166	98.971	1.0	1.0	45.151	49.892	1.731	1.984	28.05	25.406	2.787	3.895
web-indochina-2004-all	0.49	-	1.0	-	0.505	-	0.971	-	0.507	-	0.966	-
web-it-2004	0.005	-	1.0	-	0.005	-	1.011	-	0.005	-	1.024	-
web-uk-2002-all	1.166	-	1.0	-	1.15	-	1.014	-	1.183	-	0.986	-
web-wikipedia_link_it	3.15	-	1.0	-	3.192	-	0.987	-	3.154	-	0.999	-
wiki-talk	0.441	0.086	1.0	1.0	0.315	0.043	1.399	1.981	0.242	0.023	1.825	3.697

Table 5.6: Runtimes of Multithreaded MC-BRB algorithm over increasing number of threads

	8 Threads				10 Threads				20 Threads			
	Runtimes		Speed-ups		Runtimes		Speed-ups		Runtimes		Speed-ups	
Instance	Init.	Search	Init.	Search	Init.	Search.	Init.	Search	Init.	Search	Init.	Search
aff-digg	5.668	69.992	6.184	7.704	4.777	60.192	7.338	8.958	3.954	43.518	8.864	12.39
aff-orkut-user2groups	83.889	0.475	3.344	0.357	82.875	0.618	3.385	0.274	99.628	0.64	2.816	0.265
bio-human-gene1	4.901	1.776	5.788	7.149	4.478	1.443	6.336	8.797	4.242	1.377	6.688	9.22
bio-human-gene2	3.099	0.984	5.189	7.264	2.877	0.846	5.589	8.446	2.802	0.801	5.74	8.924
bn-human-BNU_1_0025864_session_1-bg	33.318	4.569	6.438	7.736	29.126	3.829	7.365	9.232	23.367	3.102	9.18	11.394
bn-human-BNU_1_0025864_session_2-bg	27.786	0.225	6.947	7.981	24.054	0.201	8.025	8.912	19.834	0.189	9.733	9.493
bn-human-Jung2015_M87113878	9.844	0.011	5.909	1.956	8.449	0.024	6.884	0.945	7.257	0.039	8.015	0.577
bn-human-Jung2015_M87126525	9.083	0.085	5.171	5.536	8.255	0.075	5.689	6.253	6.919	0.091	6.788	5.119
c-62ghs	0.004	-	0.964	-	0.004	-	0.97	-	0.004	-	0.965	-
co-papers-dblp	0.01	-	1.009	-	0.01	-	0.999	-	0.011	-	0.968	-
rec-yahoo-songs	15.443	10.462	5.59	7.545	13.598	8.601	6.349	9.177	12.299	6.868	7.019	11.494
soc-LiveJournal1	1.072	-	1.123	-	1.187	-	1.014	-	1.057	-	1.139	-
soc-flickr	0.263	0.194	4.788	7.665	0.229	0.155	5.495	9.568	0.201	0.113	6.284	13.122
soc-flickr-und	2.02	2.318	5.304	7.563	1.764	1.873	6.075	9.36	1.505	1.435	7.118	12.214
soc-livejournal-user-groups	13.8	0.282	2.776	0.096	13.687	0.36	2.798	0.075	13.961	0.468	2.744	0.058
soc-orkut	10.31	0.268	1.784	0.084	9.573	0.33	1.921	0.068	9.333	0.408	1.971	0.055
soc-sinaweibo	21.836	0.717	1.124	0.125	21.277	0.464	1.153	0.194	21.526	1.604	1.14	0.056
socfb-A-anon	1.218	0.042	1.285	0.086	1.259	0.055	1.243	0.066	1.282	0.083	1.221	0.044
socfb-konect	4.828	-	0.996	-	4.854	-	0.991	-	4.902	-	0.981	-
tech-p2p	20.877	13.019	3.744	7.602	18.743	10.644	4.17	9.298	17.682	9.411	4.421	10.516
web-indochina-2004-all	0.502	-	0.977	-	0.511	-	0.96	-	0.501	-	0.978	-
web-it-2004	0.005	-	0.994	-	0.005	-	0.982	-	0.005	-	1.024	-
web-uk-2002-all	1.187	-	0.983	-	1.168	-	0.998	-	1.182	-	0.986	-
web-wikipedia_link_it	3.205	-	0.983	-	3.131	-	1.006	-	3.308	-	0.952	-
wiki-talk	0.217	0.013	2.035	6.853	0.203	0.01	2.175	8.607	0.191	0.009	2.307	10.022

Table 5.7: Runtimes of Multithreaded MC-BRB algorithm over increasing number of threads

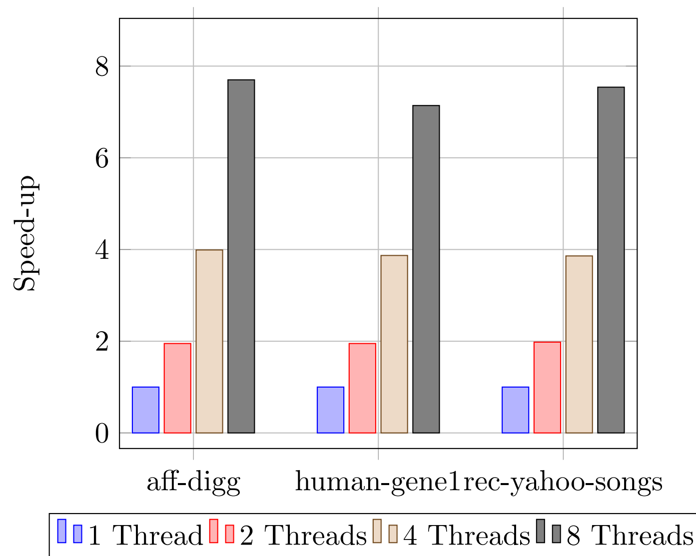


Figure 5.3: Bar chart showing speedup over the number of threads

on small and denser instances so a random graph of 250 nodes each one with density varying from $[0.8, 0.99]$.

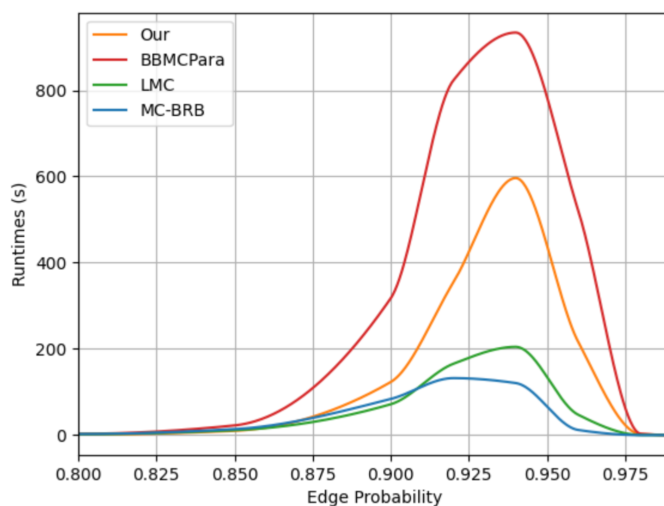
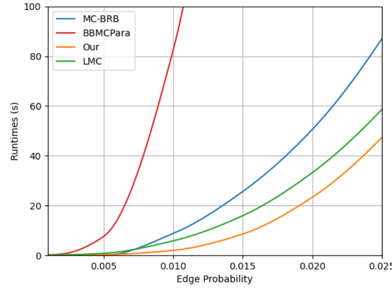


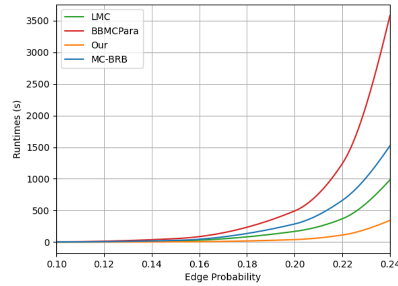
Figure 5.4: Runtimes over edge probability for random graphs of 250 vertices

In Figure 5.4 Runtimes have been plotted over edge probability, runtimes initially grow exponentially til they reach a maximum of about 0.94 of edge probability then the problem becomes easier and decreases til the runtime becomes instant. MC-BRB and LMC perform better under these conditions (high density) Their pruning strategies such as MaxSAT and Reduce framework are effective and cut more branches with respect to just using coloring. The problem has been widely studied, when we increase the size of the random instances the runtimes will change. for those small instances, runtimes for density lower than 0.8 cannot be seen because the problem is too easy. But increasing the size can make the problem harder increasing runtimes.

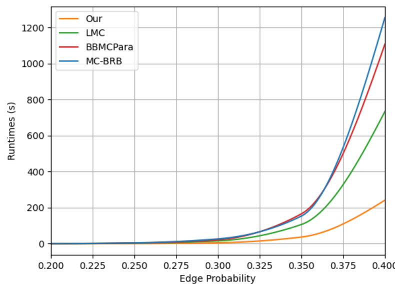
Plots in Figure 5.5 show how all algorithms behave with bigger instances, the graph size ranges from 1,000 to 100,000 vertices, varying density. We terminate our test when runtimes go over the set timeout of 3,600 seconds. From the plots we can see that MC-BRB and LMC behave badly with respect to our GPU implementation this is because their pruning technique becomes ineffective and all algorithms become equal, so the kind optimization is the threads scaling factor of the GPU, so the speedup is given by the hardware rather than the software optimization.



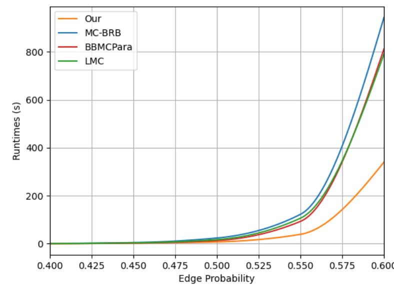
(a) Instances of 100k vertices



(b) Instances of 10k vertices



(c) Instances of 3k vertices



(d) Instances of 1k vertices

Figure 5.5: Runtimes of our algorithms (on the y-axis) on random graphs as a function of the graph density (reported on the x-axis). All times are reported in seconds.

5.4.2 Real-world dataset

This section explores runtimes over the real dataset by comparing each approach. Following the result obtained in a random graph we can tell where each version performs better. So we collect the related statistics like the density of the induced subgraph in which the dense search procedure is applied.

Table 5.8 shows The main properties of the graphs we can see that the instances are ordered with respect to the induced subgraph density to facilitate the reading of the table. The related runtimes are reported in Table 5.9. The runtimes are coherent with respect to the runtimes obtained in random instances, we can see that with lower density the MaxSAT Pruning strategy and the reduced one become less effective instead for higher density it is the opposite (see bio-human-gene1). MC-BRB is much faster in dense instances while LMC is a trade-off between dense and sparse, while our GPU version performs better in sparse instances.

The last Table (Table 5.10) shows the speed-up with respect to the BBM-

Instance	$\lvert V(G') \rvert$	$\lvert E(G') \rvert$	Cut vertices			$\lvert V(G'') \rvert$		$d(G'')$	
			L1 color	k-core	color	Max.	Avg.	Max.	Avg.
web-uk-2002-all	-	-	-	-	-	-	-	-	-
c-62ghs	27,147	488,444	25,956	0	0	-	-	-	-
web-indochina-2004-all	6,985	48,769,254	-	-	-	-	-	-	-
co-papers-dblp	-	-	-	-	-	-	-	-	-
web-it-2004	-	-	-	-	-	-	-	-	-
rec-yahoo-songs	136,736	99,393,144	110,386	0	2,090,108	4,152	651	0.373239	0.128711
aff-orkut-user2groups	6,828,633	650,268,524	4,165,348	1,018	3,260	434	59	1.0	0.245209
aff-digg	138,836	38,128,250	105,303	0	44,283,740	2,169	417	0.507453	0.321143
soc-livejournal-user-groups	2,968,508	212,812,908	2,399,124	399	1,743	183	41	1.0	0.330844
soc-orkut	2,204,867	198,472,626	687,697	1	43,821	1,291	209	0.981481	0.345562
soc-sinaweibo	7,002,886	306,871,826	707,546	53	23,842	521	142	0.955556	0.363507
wiki-talk	15,807	1,588,498	13,937	0	5,667	366	130	0.697643	0.473348
bn-human-Jung2015_M87113878	189,786	140,358,954	127,431	0	35,425,870	2,217	837	1.0	0.545527
soc-flickr	13,093	2,841,634	10,551	0	102,737	625	293	0.708151	0.578541
bn-human-BNU_1_0025864_session_2-bg	247,953	213,379,384	165,034	0	1,958,461	3,265	1,071	1.0	0.580141
tech-p2p	165,511	108,048,864	157,859	0	415,196,791	2,229	972	0.803262	0.589305
bn-human-BNU_1_0025864_session_1-bg	246,815	227,442,558	162,418	0	6,660,746	4,187	1,182	0.989505	0.597659
soc-flickr-und	48,771	18,145,866	35,142	0	1,330,516	1,367	519	0.811057	0.604331
socfb-A-anon	390,144	29,024,572	349,993	5	122	261	81	0.944086	0.630807
socfb-konect	499,734	6,099,972	253,585	15	3	34	16	0.904762	0.697649
bn-human-Jung2015_M87126525	303,434	194,931,398	229,888	0	529,666	3,835	665	0.917156	0.707567
bio-human-gene1	4,551	13,174,682	2,491	0	15,082,028	2,701	2,203	0.971053	0.948887
bio-human-gene2	3,814	10,107,000	1,969	0	1,208,590	2,389	2,024	0.975569	0.959575
soc-LiveJournal1	474	211,150	148	0	36	392	388	0.994685	0.994568
web-wikipedia_link_it	938	876,240	67	0	23	892	892	0.999477	0.999477

Table 5.8: Runtime statistics: The table reports the main characteristics of the graph G' and $G'' = G'[P']$ obtained from the original graph G by our algorithm 31. They give some hints on the efficacy of each step of our procedure in simplifying the original graph during the MC computation.

Instance	Our				BBMCPara			LMC			MC-BRB		
	$\lvert \omega_0(G) \rvert$	Init	Search (Block)	Search (Warp)	$\lvert \omega_0(G) \rvert$	Init	Search	$\lvert \omega_0(G) \rvert$	Init	Search	$\lvert \omega_0(G) \rvert$	Init	Search
aff-digg	27	0.777	12.408	5.993	25	15.731	57.206	29	0.87	14.25	30	4.089	43.037
aff-orkut-user2groups	2	16.041	1.786	1.664	5	-	O.O.M.	2	25.59	21.36	6	95.701	0.695
bio-human-gene1	1,327	0.472	466.889	271.383	1,268	4.717	T.O.	1,328	0.27	164.72	1,335	4.217	1.381
bio-human-gene2	1,292	0.441	50.028	46.633	1,229	2.659	292.425	1,290	0.18	28.32	1,300	2.751	0.79
bn-human-BNU_1_0025864_session_1-bg	221	2.508	31.765	27.427	276	106.205	T.O.	222	6.45	33.36	283	23.376	3.097
bn-human-BNU_1_0025864_session_2-bg	199	2.689	11.422	14.131	271	95.686	38.707	199	6.15	26.08	271	19.632	0.173
bn-human-Jung2015_M87113878	140	1.47	65.681	67.416	221	49.459	2.501	133	3.52	18.49	227	7.413	0.038
bn-human-Jung2015_M87126525	195	2.184	4.817	4.635	206	73.561	129.21	196	5.55	13.76	219	7.07	0.082
c-62ghs	2	0.059	0.002	0.002	2	0.081	0.006	2	0.01	0.01	2	0.004	-
co-papers-dblp	337	0.066	-	-	337	0.247	-	337	0.11	0.0	337	0.011	-
rec-yahoo-songs	16	1.67	3.053	4.0	11	44.484	80.39	16	2.66	6.49	18	12.236	6.755
soc-LiveJournal1	320	0.335	0.098	0.104	316	20.61	0.009	320	2.43	0.01	321	1.072	-
soc-flickr	54	0.183	0.055	0.052	40	1.499	0.224	52	0.1	0.09	57	0.199	0.113
soc-flickr-und	74	0.488	1.119	0.7	68	9.007	9.303	74	0.63	1.27	96	1.484	1.397
soc-livejournal-user-groups	5	11.711	0.256	0.254	8	367.839	120.238	4	7.65	2.39	9	14.0	0.45
soc-orkut	18	3.404	0.238	0.234	46	75.43	3.649	17	9.24	1.73	46	8.934	0.521
soc-sinaweibo	8	6.323	0.256	0.237	41	532.532	10.669	8	20.48	2.42	44	20.847	1.655
socfb-A-anon	23	0.495	0.032	0.033	24	14.625	0.587	23	1.33	0.24	25	1.339	0.093
socfb-konect	6	0.864	0.006	0.007	6	57.679	0.089	6	7.76	0.12	6	4.801	-
tech-p2p	173	2.149	856.811	453.74	153	214.064	T.O.	172	12.92	13.24	175	17.615	9.733
web-indochina-2004-all	6,848	1.095	O.O.M.	O.O.M.	6,848	27.29	0.026	6,848	1.87	4.58	6,848	0.499	-
web-it-2004	432	0.041	-	-	432	0.068	-	432	0.03	0.0	432	0.005	-
web-uk-2002-all	944	0.657	-	-	944	7.28	-	944	5.93	0.0	944	1.168	-
web-wikipedia_link_it	869	1.253	0.974	0.958	869	29.296	0.023	869	2.43	0.02	870	3.316	-
wiki-talk	24	0.228	0.02	0.013	16	4.504	0.043	25	0.13	0.02	26	0.186	0.008

Table 5.9: Runtime statistics: The table reports the preprocessing and the runtime for all algorithms. The time-out (TO) is set to 3,600 seconds, i.e., one hour. The quantity of memory available is 8GBytes, and beyond that limit, we have an out-of-memory error (OOM). For our algorithm, we report data for both the block-based and warp-based versions.

CPara version.

Instance	Our			LMC		MC-BRB	
	Init	Search (Block)	Search (Warp)	Init.	Search	Init	Search
aff-digg	20.25	4.61	9.55	18.08	4.01	3.85	1.33
aff-orkut-user2groups	-	-	-	-	-	-	-
bio-human-gene1	10.0	-	-	17.47	-	1.12	-
bio-human-gene2	6.03	5.85	6.27	14.77	10.33	0.97	370.3
bn-human-BNU_1_0025864_session_1-bg	42.35	-	-	16.47	-	4.54	-
bn-human-BNU_1_0025864_session_2-bg	35.58	3.39	2.74	15.56	1.48	4.87	223.88
bn-human-Jung2015_M87113878	33.64	0.04	0.04	14.05	0.14	6.67	66.04
bn-human-Jung2015_M87126525	33.68	26.82	27.88	13.25	9.39	10.4	1570.37
c-62ghs	1.37	2.75	2.52	8.1	0.6	22.84	-
co-papers-dblp	3.74	-	-	2.25	-	23.16	-
rec-yahoo-songs	26.63	26.34	20.1	16.72	12.39	3.64	11.9
soc-LiveJournal1	61.58	0.09	0.09	8.48	0.9	19.23	-
soc-flickr	8.2	4.04	4.34	14.99	2.49	7.53	1.98
soc-flickr-und	18.47	8.31	13.28	14.3	7.33	6.07	6.66
soc-livejournal-user-groups	31.41	469.65	472.77	48.08	50.31	26.27	267.19
soc-orkut	22.16	15.34	15.59	8.16	2.11	8.44	7.01
soc-sinaweibo	84.22	41.73	45.07	26.0	4.41	25.54	6.45
socfb-A-anon	29.53	18.39	17.83	11.0	2.45	10.92	6.29
socfb-konect	66.73	14.17	12.4	7.43	0.74	12.01	-
tech-p2p	99.62	-	-	16.57	-	12.15	-
web-indochina-2004-all	24.91	-	-	14.59	0.01	54.73	-
web-it-2004	1.67	-	-	2.27	-	15.1	-
web-uk-2002-all	11.08	-	-	1.23	-	6.24	-
web-wikipedia_link_it	23.38	0.02	0.02	12.06	1.15	8.83	-
wiki-talk	19.77	2.18	3.4	34.65	2.15	24.28	5.33

Table 5.10: Speed-up of the three main strategies implemented with respect to the BBMCPara algorithm. The table reports the speed-ups for the initialization and search phases separately and the overall speed-up of the entire algorithm (i.e., of the two phases together). For our algorithm, we report data for both the block-based and warp-based versions.

5.5 Warp-wise Parallel

The last version presented is the warp-wise parallel, in which we go deep into one level of parallelism explicitly using warps. Runtimes have been reported in Table 5.9 And show that when the width of the induced subgraph is smaller the block size times 32 becomes more effective. this approach outperforms the previous version of about 2x in some instances. This division increases the effectiveness of the work donation since more warps than blocks participate in solving dense instances. To make more explicit the effect of the warp-wise parallelism we test it also on random instances to see the effect. We start from the random instances of 250 nodes and compare just the warp and block parallel versions.

All random instances have been tested to see when this kind of optimization is performant:

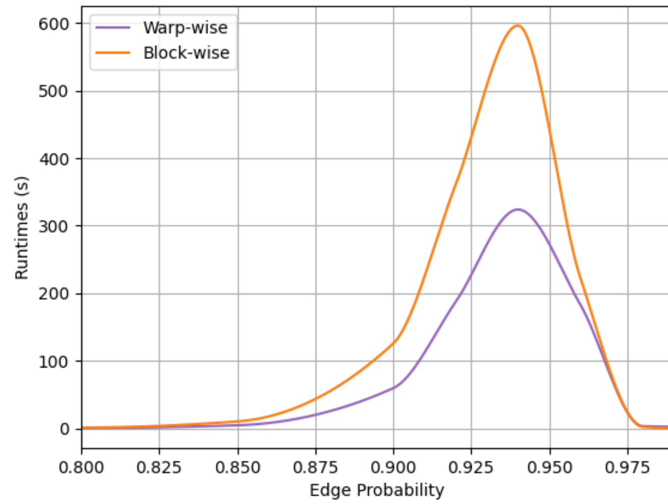
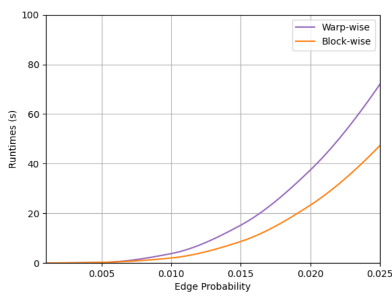
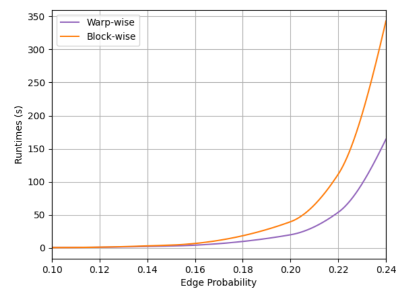


Figure 5.6: Runtimes over edge probability for random graphs of 250 vertices

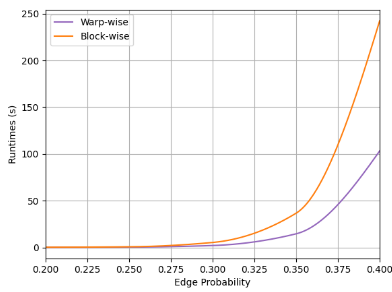
Warp-wise parallel implementation performs well up to random with 10,000 vertices, we can see a speedup of about 2x but paying in memory to increase the number of independent tasks run in parallel.



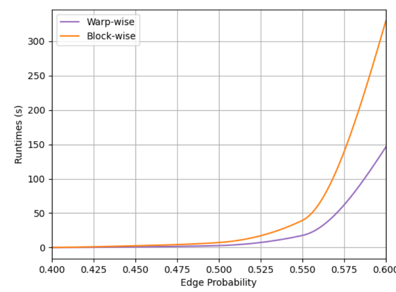
(a) Instances of 100k vertices



(b) Instances of 10k vertices



(c) Instances of 3k vertices



(d) Instances of 1k vertices

Figure 5.7: Runtimes of our algorithms (on the y-axis) on random graphs as a function of the graph density (reported on the x-axis). All times are reported in seconds.

Chapter 6

Conclusions

To summarize we describe the maximum clique problem and its related application, we go through the entire story of the algorithm describing the main optimizations able to significantly increase the performance, and finally, we describe an approach suitable for GPU and introduce a possible optimization of this approach to maximize GPU occupancy. Many of the current coloring strategies have been tested and studied its portability on GPU, showing that Global Memory can affect the runtimes slowing the process of discarding vertices. So pay attention and make a tradeoff of the time spent to compute better coloring. The related GPU version has been tested against the faster state-of-the-art solvers we show that the GPU outperforms all CPU versions in some locally sparse instances, but we could further improve its performance by using MaxSAT reasoning pruning strategy proposed by LMC or the reducing technique proposed by MC-BRB. A new implementation of the SAT on GPUs could improve the performance, but be careful in implementing it, this implementation has to minimize the memory access on Global Memory, because as seen in the ReColor pruning strategy this can lead to wasting much time rather than gain that. Overall we can say that the parallel strategy proposed by BBMCG is effective in speed-up this kind of backtracking algorithm on GPU, we show that it is possible to go down one level of parallelism to take benefit from those threads that keep inactive because of the low local degree of the graph. Finally is important to say that GPU architecture can be exploited to execute this kind of algorithm with lower runtimes, but not all the optimization available on CPU can fit well on this architecture. As said before better speedups can be achieved by increasing the occupancy, and bitsets allow parallel bit set operations, this is one of the main reasons for the incredible speedup and reduced memory usage. Can be interesting to see how the GPU behaves without the bitsets, with bitsets where most of the bits are set to 0 we can make the thread perform redundant work, this can happen when we go into a deeper level of recursion. Without bitsets, we could have to deal

with warp divergence and non-coalesced memory access that can slow down the set operations.

Chapter 7

Acknowledgements

I would like to thank my family: Emanuele, Mary (Maria), Marika, Nikole, Romeo and Marley, Salvatore (grandpa), Salvatrice (grandma), Maria (grandma), Paolo (grandpa), my uncles: Corrado, Carmen, Laura, Salvatore, Paolo and all my cousins: Greta, Julia, and Erika for supporting me over these years, My supervisors for guiding me through the development of this thesis, and the Politecnico of Turin for teaching me all the subject related to Computer Science. Those years have been the most important of my life because I learned a lot of beautiful things that gave me the basics to enrich my knowledge. Finally, I would like to do a special thank my grandpa Salvatore for giving me the possibility to start and continue through this path. Thank you.

List of Algorithms

1 PowerSet 17
2 Enumerate all cliques of Graph G 19
3 Enumerate all cliques of Graph G 20
4 BronKerbosch: Enumerate all maximal cliques of Graph G from [1] 21
5 BronKerbosch: First-level independent subtree [1] 22
6 k-cores decomposition. 23
7 MCQ: Find the maximum clique of a Graph G [18] 24
8 EXPAND: Find the maximum clique of a Graph G [18] 25
9 NUMBER-SORT: Assign colors with a greedy strategy to vertices in R [18] 26
10 BB-MaxClique: Find the maximum clique on a graph G [17] . 27
11 BB-Color: Assign a color to a set of vertices U_{BB} [17] 28
12 A threaded algorithm to deliver the max clique [11] 29
13 expand: main search procedure for max clique [11] 30
14 colorOrder: vertex coloring [11] 31
15 BBMCSP [13]: max clique computation for large sparse graphs 32
16 BBMCSP [13]: max clique computation for large sparse graphs 33
17 Re-NUMBER [19] Try color v with color less equal than k_{min} . 34
18 MC: max clique computation for dense graph from [7] 35
19 $GetBranches_{d0}$: Compute the set of branching vertices [7] . . 36
20 $GetBranches_{d}$: Compute the set of branching vertices [7] . . 37
21 IncMaxSAT: Further reduce the set of branching vertices [7] . 38
22 $GetBranches_{s}$: Compute the set of branching vertices [7] . . 39
23 MC2: max clique computation for dense graph [7] 41
24 $GetBranches_{m}$: Compute the set of branching vertices [7] . . 42
25 *Initialize*: A preprocessing for large and sparse graph [6] . . . 43
26 *LMC*: A BnB algorithm for computing the max clique [6] . . 44
27 *SearchMaxClique*: A BnB algorithm for computing the max clique greater than $|C_{max}|$ [6] 45
28 *MLMC*: A BnB algorithm for computing the max clique . . . 46

29 Parallel k-cores decomposition 53

30 Find Heuristic Clique 54
31 Pseudo-code for the parallel MC algorithm. Reported the first-level independent subtree. 55
32 Color: Assign a color to a set of vertices P 55
33 Block Parallel k-cores decomposition 56
34 Our parallel MC computation. 57
35 Pseudo-code for the parallel MC algorithm. Reported the first-level independent subtree. 58
36 Our parallel MC computation with the donation. 59

Bibliography

[1] Mohammad Almasri, Yen-Hsiang Chang, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen mei Hwu. Parallelizing maximal clique enumeration on gpus, 2023.

[2] Vladimir Boginski, Sergiy Butenko, and Panos M Pardalos. Mining market data: A network approach. *Computers & Operations Research*, 33(11):3171–3184, 2006.

[3] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communication of the ACM*, 16(9):575–577, 1973.

[4] Lijun Chang. Efficient maximum clique computation and enumeration over large sparse graphs. *The VLDB Journal*, pages 999–1022, 2020.

[5] Tuvi Etzion and Patric RJ Ostergard. Greedy and heuristic algorithms for codes and colorings. *IEEE Transactions on Information Theory*, 44(1):382–388, 1998.

[6] Hua Jiang, Chu-Min Li, and Felip Manyà. Combining efficient preprocessing and incremental maxsat reasoning for maxclique in large graphs. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ECAI'16, page 939–947, NLD, 2016. IOS Press.

[7] Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017.

[8] Chu-Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):128–133, Jul. 2010.

[9] Noël Malod-Dognin, Rumen Andonov, and Nicola Yanev. Maximum cliques in protein structure comparison, 2009.

[10] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Michel Rueher, editor, *Principles*

and Practice of Constraint Programming, pages 350–368, Cham, 2016. Springer International Publishing.

[11] Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013.

[12] NVIDIA. CUDA Programming Guide.

[13] San Segundo Pablo, Lopez Alvaro, Jorge Artieda, and Panos M. Pardalos. A parallel maximum clique algorithm for large and massive sparse graphs. *Optimization Letters*, 11:343–358, 2017.

[14] Jeffrey Pattillo, Nataly Youssef, and Sergiy Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.

[15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[16] Pablo San Segundo, Fernando Matia, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7:467–479, 03 2011.

[17] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[18] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In Cristian Calude, Michael J. Dinneen, and Vincent Vajnovszki, editors, *Discrete Mathematics and Theoretical Computer Science, 4th International Conference, DMTCS 2003, Dijon, France, July 7-12, 2003. Proceedings*, volume 2731 of *Lecture Notes in Computer Science*, pages 278–289. Springer, 2003.

[19] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In Md. Saidur Rahman and Satoshi Fujita, editors, *WALCOM: Algorithms and Computation*, pages 191–203, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.