Maximum Clique Solver using Bitsets on GPUs

Matthew VanCompernolle[1], Lee Barford[1,2], and Frederick Harris, Jr.[1]

[1] Department of Computer Science and Engineering, University of Nevada, Reno
[2] Keysight Laboratories, Keysight Technologies
mvancomp@nevada.unr.edu, lee_barford@ieee.org, Fred.Harris@cse.unr.edu

Abstract. Finding the maximum clique in a graph is useful for solving problems in many real world applications. However the problem is classified as NP-hard, thus making it very difficult to solve for large and dense graphs. This paper presents one of the only exact maximum clique solvers that takes advantage of the parallelism of Graphical Processing Units (GPUs). The algorithm makes use of bitsets to reduce the amount of storage space needed and take advantage of bit-level parallelism in hardware to increase performance. The results show that the GPU implementation of the algorithm performs better than the corresponding sequential algorithm in almost all cases; performance gains tend to be more prominent on larger graph sizes that can be solved using more levels of parallelism.

1 Introduction

A clique, or complete sub-graph, is a subset of the vertices in an undirected graph, such that there exists and edge between every pair of vertices in the subset. The Maximum Clique Problem (MCP) is one of the most studied NP-hard problems which consists of finding the largest possible clique in a graph. For any large or dense graph, MCP often cannot be solved in a reasonable amount of time even on high end machines due to its exponential time complexity. Despite the difficulty of the problem, finding a maximum clique has many useful applications in areas such as bioinformatics, computer vision, robotics, patterns in telecommunications traffic, and more [2, 11]. For this reason and the fact that many real-world graphs do not elicit worst-case behaviour for MCP, new algorithms and optimizations to the problem continue to be a popular field of research [8].

One emerging trend in research is the use of Graphical Processing Units (GPUs) for general purpose computing to develop high performance applications. The release of NVIDIA's CUDA, a general purpose parallel computing platform released in 2006, enabled programmers to finally take advantage of the parallelism and high computational power of NVIDIA's GPUs while enabling a low learning curve for developers familiar with standard programming languages [5].

This paper describes the implementation of a maximum clique algorithm on GPUs that utilizes bitset representations to reduce memory requirements and increase performance. The implementation does many things that most maximum clique solvers to not attempt, despite the popularity of the field of research.

The algorithm presented is one of very few maximum clique solvers that runs on GPUs, makes use of recursion on the GPU, and supports systems with multiple GPUs.

The rest of the paper is structure as follows: Section II covers background information necessary to better understand the proposed algorithm and summarizes related maximum clique algorithms. Section III explains the new maximum clique algorithm in detail. The results, comparisons between the sequential and parallel GPU algorithm, and an analysis of the results are discussed in section IV. Finally, conclusions and possible future improvements are stated in Section V.

2 Background

Basic Maximum Clique Algorithm: A basic method for searching for a maximum clique in a graph $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges that are unordered pairs (u, v) where $u, v \in V$, is a branch and bound, depth first search method. The basic implementation explained in [14] uses two global sets: one to store the current clique Q and another to store the largest clique found so far, Q_{max}. A candidate set R is also stored throughout the search to keep track of which vertices may be added to Q. When the algorithm starts, Q is empty and R contains all vertices in the graph, $R = V$. The search process beings by selected a vertex p from R and adding the vertex to Q, forming a new current clique, and then proceeds to calculate a new candidate set R that is the intersection of the current R and the vertices adjacent to p. This process occurs recursively until R is empty, which means that Q is a maximal clique and cannot grow any larger, or $|Q| + |R| <= |Q_{max}|$ which means that the Q cannot grow larger than Q_{max}. When Q is determined to be maximal, it is compared against Q_{max}, and if $|Q| > |Q_{max}|$, $Q_{max} = Q$. The search then backtracks in order to find other maximal cliques, removing p from both Q and R and then selected a new p from R and repeating the process until the search space has been exhausted.

Approximate Coloring: The basic MCP algorithm is much too slow to find the maximum clique for large or dense graphs in a reasonable amount of time. One way to increase the performance of the algorithm is to further prune the search space in order to avoid unnecessary searching. One method of doing so is to use approximate coloring of the graph vertices, such as the ones used in the MCQ, MCR, MCS, and BBMC algorithms [11, 14, 6, 15].

Bitset Representations and BBMC: Another alteration to the basic MCP implementation is the use of bitsets to represent which vertices are in a set. The BBMC algorithm along with its many variations such as BBMCI, BBMCR, BBMCL, and BBMCS all use bitsets to represent whether a vertex from the graph is in a set or not [9–11, 13]. The use of bitsets allows the algorithms to take advantage of bit-level parallelism in hardware where a single computer instruction can operate on a number of bits the size of a word on a processor at a time (usually 32 or 64 bits in a word). Algorithms using bitsets therefore have two main advantages: the use of a bitset instead of typical storage implementations reduces the memory needed by a factor equal to the size of a word on the system, and common operations can be done in parallel using bit-wise operations.

Parallel MCP Algorithms: A significant way to speed up the search for the maximum clique in a graph is to take advantage of any available parallelism in hardware. The majority of parallel implementations for MCP have been written to take advantage of multiple CPU cores. The state-of-the-art parallel exact maximum clique finder in [7] was designed for large sparse graphs to exploit characteristics of social and informational networks and is parallelized on a shared memory system, but could also be used on a distributed memory architecture. The implementation achieved approximately linear speedup with respect to the number of processors used, while sometimes achieving super linear speedup due to increased pruning caused from workers sharing upper bounds with each other.

A very similar parallel approach was proposed in [4] that searches for the maximum clique in a graph using multiple threads. The implementation is based off of the sequential BBMC algorithm that uses bitsets and is implemented in C++ using C++11 threads and shared memory parallelism.

There are also algorithms that attempt to utilize the GPU to solve MCP, though there are not every many. One implementation used a neural network and CUDA to search for the maximum clique in a graph using a GPU [1]. The algorithm was limited, only being able to handle graphs with 800 vertices maximum due to memory constraints.

3 Parallel GPU Implementation

Parallel Method: The BBMC implementation on the GPU, which will be referred to as BBMCG, has two distinct dimensions of parallelism. The first dimension uses block-wise parallelism to divide the search space of the graph into subtrees in order to traverse the search space in parallel. This method was inspired by the work queue distribution method proposed in [4], where a work queue is initially populated by splitting the search space immediately below the root. In BBMCG, each block is initialized with a current clique Q that contains one vertex from the graph, indicated by a 1 at the position for the vertex in the bitset. The corresponding initial candidate set R for each block is a set containing all the adjacent vertices to the vertex in Q. BBMCG therefore creates a number of blocks for the algorithm that is equal to the number of vertices in the graph. Each block then traverses its own search tree recursively utilizing recursion on the GPU based on the BBMC algorithm described in [6, 11, 12]. A single global variable is shared between the blocks to indicate the current maximum clique size found. If a block finds a clique in its search space larger than the current maximum, it atomically updates the global maximum using CUDA's built in atomicMax function and stores the maximum clique's bitset in a designated space for the block in global memory. By traversing the search space in parallel and communicating the largest clique found, more of the search space is potentially pruned as larger cliques are found faster.

BBMCG's second dimension of parallelism is thread-wise parallel computations for bit-wise operations. CUDA lacks a bitset class, so bitsets are implemented using arrays of unsigned integers. In a bitset, a single bit is needed for each vertex in the graph. The number of unsigned integers needed to represent a bitset is calculated by dividing the number of vertices n in the graph by the number of bits in an unsigned integer in CUDA and rounding up. Since unsigned integers are typically 32 bits long, one unsigned integer is needed to represent 32 vertices in the graph. The adjacency matrix which requires a bitset for each vertex in the graph could then be represented as a 2-D array of unsigned integers, but CUDA generally prefers working with contiguous 1-D arrays. The adjacency and inverse adjacency matrices are stored as 1-D arrays of unsigned integer mappings of the 2-D array equivalents.

Any bit-wise operation that is not sequential in nature is parallelized by performing the operations using multiple threads per block. The number of threads used per block is equivalent to the number of unsigned integers needed to represent a bitset for the graph. BBMCG maps a single thread to each unsigned integer of a bitset in order for the threads to perform bit-wise operations on subsections of the entire bitset in parallel. In the base BBMC algorithm, the majority of computations are bit-wise operations on bitsets such as finding the first set bit, getting the number of set bits, intersecting bitsets, copying bitsets, and setting or clearing a bit in a bitset. All of these operations besides setting and clearing a bit can be executed in parallel to increase performance. For example, to calculate the number of bits that are 1 in a bitset, each thread executes the function *__popc* on its corresponding unsigned integer from the bitset, which is a built in CUDA function that counts the number of bits set to 1 in an unsigned int. Results of each thread are then stored in array in shared memory, and the threads calculate the sum of all the set bits using an efficient parallel vector reduction algorithm on the array.

Memory Allocation and Management: Although the BBMCG is able to run on graph sizes larger than those in [1], due to the use of bitsets, some memory limitations were still encountered. One of the primary memory consumers in BBMCG is the storage space needed to store the run-time stack for recursion. Each recursive call in the search needs memory to store local variables, arrays to store the coloring of vertices to improve pruning, and several bitsets to store the current clique and candidate set. The maximum number of recursive calls directly corresponds to the size of the maximum clique in the graph, so the algorithm needs to allocate enough memory to reach the maximum depth to find the maximum clique in worst case scenarios. The amount of memory needed is then multiplied by the number of blocks used, which all make their own recursive calls as they search. Faster forms of memory, such as local registers and shared memory (limited to 48 KB on modern cards) are not large enough to store data at each level of recursion for all blocks. Even constant memory memory is limited to 64 KB and is too small to store the adjacency matrices as suggested in [1]. A single adjacency matrix for a graph size of 700 is approximately the maximum size that can be stored in constant memory, which would severely limit the problems BBMCG can run on.

BBMCG makes heavy use of global memory to address the memory limitations of shared and constant memory, but still uses registers to store non-array based data. Although global memory is much slower than shared and constant memory, it is necessary in order to run the algorithm on reasonable sizes. Global memory needed for the for the entire problem is pre-allocated before launching the kernel. The memory is allocated as several 1-D arrays that each store a single type of data for all of the blocks in the Kernel. Each block calculates the starting position in the 1-D array for its portion of memory and points to it using a pointer. Each block's section of memory in the array is then broken down further into subsections for levels of recursion where each subsection is big enough to store the associated data for a single level. When a block makes a recursive call, a new pointer is created that is offset the size of one subsection of data from the current pointer position, pointing to a new subsection of space reserved to store data for the new level of recursion. The 1-D array is created to be large enough so that each block has enough memory to store data for each level of recursion all the way down to the maximum calculated level. One disadvantage is that the amount of memory allocated may be much larger than what is actually used, because memory is allocated for a number of recursive calls equal to the number of vertices in the graph for each block. The smaller the maximum clique is in the graph, the less reserved memory is actually used. CUDA supports dynamic memory allocation which would only use the amount of memory that is needed by the algorithm, but it is extremely slow and not worth the storage benefits.

Preprocessing and Post-processing: Several preprocessing steps need to happen before the BBMCG kernel is launched on the GPU. First the vertices of the graph need to be initially ordered based on the desired ordering method. The initial search space subtrees for each block must be generated from the graph as a pair of bitsets holding a current clique and candidate set. On the CPU the adjacency and inverse adjacency matrices are represented using a bitset class, so they must be converted to arrays of unsigned integers to be able to be passed to the GPU. Before launching the kernel, all of the global memory must be allocated for various types of data such as current cliques, candidate sets, vertex color arrays, the adjacency matrices, local maximum cliques, and more. After memory is allocated and data is correctly represented, data for the adjacency matrices and block subtrees have to copied to global memory on the GPU. Lastly, since the algorithm supports the use of multiple GPUs, threads must be created to launch a BBMCG kernel on each GPU. The multiple GPU variation of BBMCG is primarily used to increase the amount of available memory and will be explained in the discussion section. In the post-processing stage, each thread copies the local maximums of the blocks to CPU memory and finds the largest clique among the blocks.

4 Results and Discussion

Sequential Implementation and Test Environment: The sequential version of BBMC used in the results and the preprocessing portion of BBMCG were both written in C++11. The sequential implementation is a port of the BBMC algorithm provided in [6], which provides explanations of many different sequential maximum clique solvers and their implementations in Java. It is important to note that our C++ version of BBMC is slightly slower than the Java implementation it was based on. The C++11 version is nearly identical to the Java implementation, but uses vectors from the standard template library in place of the ArrayList class in Java and Boost's dynamic_bitset class in place of Java's BitSet class. Differences in these class implementations may be the cause of the performance difference, but further investigation is needed.

The computer used to gather both sequential and parallel GPU results is composed of a Intel(R) Core(TM) i7 CPU 4790K @ 4.00 GHz, two NVIDIA GTX 970's with 1664 CUDA cores running at 1050 MHz and 4 GB of memory each, and 16 GB of DDR3 system memory. Randomly generated graphs and a subset of the DIMACS benchmark graphs [3] were used to compare the sequential BBMC and parallel BBMCG algorithms.

Results: The randomly generated graphs are used to show trends for the sequential and parallel algorithms as graph size and edge density change. Figure 1 compares the running times on randomly generate graphs with 200 vertices and edge probabilities p. In the graph each pair of vertices has a probability of p to be an edge in the graph. The results show that at a size of 200 vertices, the sequential algorithm outperforms BBMCG in all cases, but the two have similar growth rates with respect to increasing edge probability. It can also be seen the preprocessing stage for BBMCG takes approximately 0.1 seconds, and is slightly higher when running on multiple GPUs than on a single one. It is the case that preprocessing always take approximately that much time, and therefore is negligible in nontrivial problems. A final observation that can be made is that BBMCG runs nearly identical on multiple GPUs as it does on a single GPU for nontrivial problems.

Figure 2 compares the running times of BBMC and BBMCG on various graph sizes. It can be seen that the growth rate for the BBMCG algorithm with respect to graph size is much lower than the growth rate of the sequential algorithm. The parallel algorithm outperforms the sequential on all graph sizes above 400 vertices. Once again, the multiple GPU implementation performs almost identical to the single GPU implementation in almost all cases.

We ran BBMC and BBMCG algorithms to find the maximum clique in a subset of the DIMACS benchmark graphs. Results show that on 20 of the 24 selected graphs, BBMCG outperforms the sequential algorithm. The four graphs where speedup does not occur all have less than 400 vertices, which is consistent with the results shown in Figure 1 and Figure 2. Typical speedup is between the 1.4 to 4.0 range, with a few more extreme outliers. The results presented use minimum-width ordering for the initial ordering, which tends to perform best on

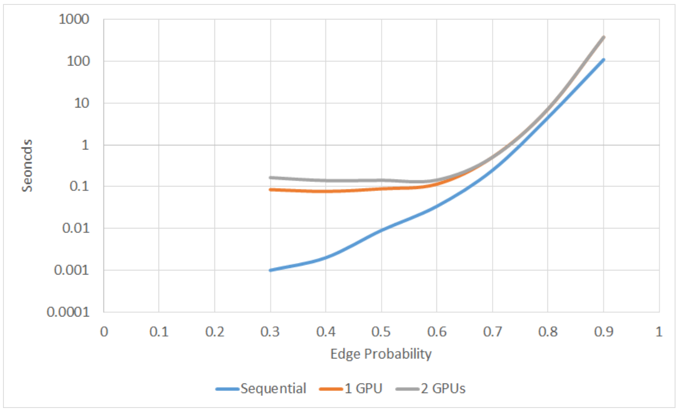

Fig. 1: Growth rate of BBMC and BBMCG run times with respect to graph density on randomly generated graphs with 200 vertices.

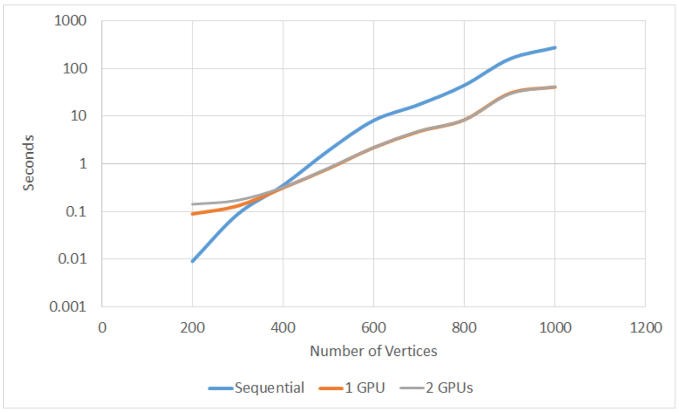

Fig. 2: Growth rate of BBMC and BBMCG run times with respect to graph size on randomly generated graphs with 0.5 edge probability.

the majority of graphs for BBMC and BBMCG. Experiments found that when using non-increasing degree or MCR methods for initial ordering the results were comparable.

Analysis: The results were consistently positive on graphs of 400 vertices or more. This seems to be the threshold where the BBMCG algorithm can start taking advantage of the parallel processing power of the GPU. As the size of the graph increases, both block-wise and thread-wise parallel components increase in parallelism. The number of blocks used in BBMCG is equivalent to the number of vertices in the graph. Therefore, larger graphs have more ways to divide up the search space and consequently can optimally use more blocks to search different subtrees. The number of subtrees used to split the search space into was experimented with, and the ideal number of blocks to use almost always turned out to be the number of vertices in the graph, not more or less. A performance gain is much more likely on larger graphs, as many different portions of the graph are all searched simultaneously with the possibility of containing the maximum clique. Searching the graph subtrees simultaneously means that large cliques can be found from subtrees that would not be searched sequentially until later in the search, further pruning the graph and removing unnecessary searching. This can sometimes lead to extreme speedup, such as in the case of san400_0.9_1 where BBMCG obtained 2,3076.9 speedup over BBMC.

Thread-wise parallelism is a much more consistent method of increasing performance. Block-wise parallelism relies on a non-deterministic search method in an attempt to randomly guess good search paths and find large cliques faster in the graph, and will in most cases, but the strategy depends heavily on the characteristics of the graph being searched. Thread-wise parallelism speeds up many forms of bit-wise operations in the algorithm and does so regardless of the characteristics of the graph. As the size of the graph increases, the number of unsigned integers needed in a bitset to represent the graph increases. Since the number of threads in a block is equivalent to the number of unsigned integers in a bitset, it means that more threads can be used as the graph size increases. In order for a block to make use of 32 threads, which is the number of threads in a warp, the graph must have at least 993 vertices. Since the GPU runs threads in groups the size of warps, most of the graphs cause BBMCG to run idling threads. Only 4 of the 24 DIMACS graphs make use of a full warp. As a result, the majority of graphs tested are not running optimally on the GPU. Another negative aspcect of thread-wise parallelism is the fact that not all operations can be parallelized, such as setting a bit, causing all but 1 thread to idle at these occurrences.

One severe limitation of the BBMCG algorithm is the lack of load balancing. When a block finishes searching its subtree, it runs out of work and terminates. It is very unlikely that each block is given an equal amount of work when the kernel is initialized, so it is very possible the most blocks only work for a fraction of the run time while a few blocks search the most difficult subtrees. A consequence of this is that the amount of parallelism reduces as the algorithm runs and performance is decreased.

The multiple GPU implementation of BBMCG suffers from the same load balancing issue and is the reason that it performs identically to a single GPU implementation. In the multiple GPU implementation, the initial subtrees for the blocks are divided among the multiple GPUs. Unified memory is used to shared the current maximum clique size between the GPUs and each GPU updates the maximum size as they find it. Unified memory has to be communicated between GPUs, causing a slight slow down. Although the additional GPUs do provide additional computational resources, it is unlikely that they have an even work load on initialization of the kernel. The benefits of finding and sharing large cliques faster across GPUs is limited by the lack of load balancing and slow communication, yielding almost no performance benefits. The block that has to search the most difficult subtree has to do it alone regardless of the addition of more GPUs. Multiple GPUs do provide additional memory however, and therefore do have some use to BBMCG. A maximum graph size of around 1500 vertices can run on a single GPU, but a maximum of around 2500 vertices can run on two GPUs.

BBMCG is also slowed down by its reliance on global memory to run. Global memory is many times slower than other available forms of memory, such as shared memory. Although shared memory is used when necessary in bitwise operations and in the coloring process, it is used seldom elsewhere. Another memory related issue is the maximum stack size CUDA allows. Even at the maximum size, it was determined that the maximum level of recursion that could be reached before memory was exhausted was approximately 500 levels. This limits BBMCG to only being able to search graphs that have a maximum clique size of about 500 maximum until it run out of memory.

5 Conclusions and Future Work

In this paper, a maximum clique algorithm for GPUs, BBMCG, was presented based off of one of the leading sequential algorithms, BBMC. The algorithm addresses memory limitations for maximum clique solvers on GPUs by using bitsets to reduce memory requirements. Two dimensions of parallelism are utilized on the GPU to divide up and search the graph's search space in parallel and to improve computational speed by parallelizing bit-wise operations. The result achieved is moderate speedup on all but small graph sizes and occasionally significant speedup on some problems.

It is promising that speedup is consistently achieved regardless of the reliance on slow global memory, a lack of a load balancing system, and small graph sizes not utilizing all of the threads in a warp. Future work will attempt to address the issues to improve performance further. Preliminary work on a load balancing system suggests that it could be a source for a large boost in performance and would have the benefit of enabling multiple GPU systems to provide further speedup. Another possibly significant alteration would be to integrate a managed memory pool system in global memory with the goal to eliminate the inefficient allocation of excess memory to blocks caused by not knowing how much memory is needed to solve the problem.

Acknowledgements

This material is based in part upon work supported by: The National Science Foundation under grant number(s) IIA-1329469, and by Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation

References

1. Cruz, R., Lopez, N., Trefftz, C.: Parallelizing a heuristic for the maximum clique problem on gpus and clusters of workstations. In: Electro/Information Technology (EIT), 2013 IEEE International Conference on. pp. 1–6. IEEE (2013)
2. Howbert, J.J., Roberts, J.: The maximum clique problem (2007), http://courses.cs.washington.edu/courses/csep521/07wi/prj/jeff_jacki.pdf, accessed: 2015-05-09
3. Johnson, D., Trick, M.: Cliques, coloring, and satisfiability, dimacs series in disc. Math. and Theoret. Comput. Sci 26 (1996)
4. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms 6(4), 618–635 (2013)
5. NVIDIA Corporation: Cuda c programming guide (2015), http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: 2015-05-09
6. Prosser, P.: Exact algorithms for maximum clique: A computational study. Algorithms 5(4), 545–587 (2012)
7. Rossi, R.A., Gleich, D.F., Gebremedhin, A.H., Patwary, M., Ali, M.: Parallel maximum clique algorithms with applications to network analysis and storage. arXiv preprint arXiv:1302.6256 (2013)
8. Rossi, R.A., Gleich, D.F., Gebremedhin, A.H., Patwary, M.M.A.: A fast parallel maximum clique algorithm for large sparse graphs and temporal strong components. CoRR abs/1302.6256 (2013), http://arxiv.org/abs/1302.6256
9. San Segundo, P., Matia, F., Rodriguez-Losada, D., Hernando, M.: An improved bit parallel exact maximum clique algorithm. Optimization Letters 7(3), 467–479 (2013)
10. San Segundo, P., Rodriguez-Losada, D.: Robust global feature based data association with a sparse bit optimized maximum clique algorithm. Robotics, IEEE Transactions on 29(5), 1332–1339 (2013)
11. San Segundo, P., Rodríguez-Losada, D., Jiménez, A.: An exact bit-parallel algorithm for the maximum clique problem. Computers & Operations Research 38(2), 571–581 (2011)
12. San Segundo, P., Rodríguez-Losada, D., Matía, F., Galán, R.: Fast exact feature based data correspondence search with an efficient bit-parallel mcp solver. Applied Intelligence 32(3), 311–329 (2010)
13. San Segundo, P., Tapia, C.: Relaxed approximate coloring in exact maximum clique search. Computers & Operations Research 44, 185–192 (2014)
14. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. Journal of Global Optimization 37(1), 95–111 (2007)
15. Tomita, E., Seki, T.: An efficient branch-and-bound algorithm for finding a maximum clique. In: Discrete mathematics and theoretical computer science, pp. 278–289. Springer (2003)
16. Trefftz, C., Santamaria-Galvis, A., Cruz, R.: Parallelizing an algorithm to find the maximal clique on interval graphs on graphical processing units. In: Electro/Information Technology (EIT), 2014 IEEE International Conference on. pp. 100–102. IEEE (2014)
17. Xiang, J., Guo, C., Aboulnaga, A.: Scalable maximum clique computation using mapreduce. In: Data Engineering (ICDE), 2013 IEEE 29th International Conference on. pp. 74–85. IEEE (2013)