Computing Maximum Cliques in Parallel

Team Perplexingly Parallel
Akshay Sharma, Srinath Obla, Vishal Kole
acs1246@rit.edu, sso8260@rit.edu, vvk3025@rit.edu
CSCI 654: Foundations of Parallel Computing

Topic Overview

In an undirected graph, a clique is a subgraph which is complete, that is, every vertex in the subgraph is connected to every other vertex. A maximum clique is the largest clique in the entire graph. Maximum cliques are widely used to analyse graphs in fields such as bioinformatics, food webs, communication networks and even chemistry. This is generally done by finding all the maximal cliques in graph. A maximal clique is a clique which can no longer be augmented by adding another neighbouring vertex, that is, it isn’t a subset of a larger clique. Our goal is to find the maximum clique of a graph in reasonable time using parallel computing.

Description of the Computational Problem Solved

Cliques are a subset of vertices present in an undirected graph such that each vertex is directly connected to all the other vertices in the set, thereby making it a complete graph.
In the above figure, there are several cliques present, but the light blue polygons are all maximal cliques and the dark blue polygons are maximum cliques. Maximal clique are cliques which cannot be extended by including another adjacent vertex of the graph in the set. Some authors define cliques to be maximal by definition. Maximal cliques should not be confused with maximum cliques. A Maximum clique is a clique in a graph such that there cannot be no clique with more vertices in it. This is also called the clique number of a graph. A maximum clique is always maximal, the converse is not always true.

In the figure given above the vertices on the left form a maximal clique while on the right form maximum clique in the example.

Finding maximum clique in graphs is a known NP complete problem. Finding maximum cliques can give important insights in the graph and could be very useful. Clique problem has been in the graph theory domain for years now and there are several algorithms to find the maximum clique in the graphs. Although, due to the computational complexity it is impossible to run a sequential algorithm on a large dataset and expect a reasonable runtime. We tried to reduce the total runtime to compute the maximum clique in a graph with parallel computing. Bron-Kerbosch was modified to run in parallel and solve the problem to find the maximum clique in parallel.
Paper Analyses

Analysis of the First Research Paper

This paper dealt entirely with the problem of finding the maximum cliques in sparse graphs with specific attention paid to social and collaboration networks. The authors came up with an algorithm which runs in two independent phases. For particular graphs the first phase alone could easily suffice. The first portion is the heuristic portion, it incrementally adds vertices to a set in a greedy manner as long as a larger clique is formed. Vertex addition here follows degeneracy order. At this point, the computation can be stopped if an approximate solution suffices. In case an exact solution is essential, the second portion which uses a branch and bound approach, can be used. It initially prunes the graph to remove all vertices that have a core number less than the size of the clique found in phase one of execution. This step was found to substantially reduce the workload going forward since a massively reduced graph is the result. The last step does further pruning by applying tight upper bounds on the size of a possible maximum clique. The relationships between the K-core number, greedy graph colouring and vertex neighbourhoods are used in this process. This step further reduces the graph such that only the maximum clique remains at the end. If multiple maximum cliques exist in the graph, both will remain and one will be picked at the end. Novel contributions include: the exploitation of relationships between the core number, the colour number obtained by greedy colouring and the reduced neighbourhood graphs to obtain a tight upper bound on the possible size of the maximum clique. The algorithms uses these bounds to aggressively prune the graph, resulting quite in a highly reduced graph to work with. We learned how the heuristic method could greatly reduce run time of the algorithm overall but ultimately decided not to use the results due to the potential sequential overhead a pre-processing step would require.

Analysis of the Second Research Paper

Similar to the previous paper, this paper also uses the PMC algorithm to find the maximum clique in a graph and exploits the relation between the chromatic number and the maximum clique. But an interesting contribution was the implementation of the data structure that improved the usage of memory while processing these graphs. A sparse bitstring implementation is proposed, where data is allocated only to the non-empty block of bits. This improves performance in comparison using a regular adjacency list, which the previous paper does.

Another useful contribution of the paper was their results from the DIMACS dataset. In our search for datasets, we were limited to those graphs where the maximum clique number was reported. The experiments in the paper were conducted on many other graphs from the DIMACS dataset and report the maximum clique number. This novel contribution enabled us to possibly use more graphs from the DIMACS dataset.
Analysis of the Third Research Paper

This paper was more practically oriented towards parallel implementation of the maximum clique finding algorithm - Bron-Kerbosch. They gave the theoretical evidence for the fast performance of the Bron–Kerbosch with the proof of linear time computation of the algorithm with respect to the maximum cliques you could have in the graph.

The time bound is nearly equal to the worst-case Big-Oh bound of \((n - d)^{d/3}\) on the number of maximal cliques when \(d\) is a multiple of 3 and \(n \geq d + 3\). For graphs with degeneracy \(d\) and maximum clique size \(k\). The idea of parameterised complexity was something that was never discussed before and proved before. The boiled down the complexity to a fixed parameter-degeneracy of the graph- and mapped it to the execution time of the graph with Bron-Kerbosch.

But the parameterised complexity did not reduce the complexity but bounded it tightly to the degeneracy of the graph, the time bound may still grow exponentially or worse with degeneracy but is independent of the input size.

The efficiency and the computational complexity is also because of the ordering of the vertices and smart selection of the vertices from the list of the vertices we have.
Design and Operation of the Sequential Program

The Bron-Kerbosch algorithm is a recursive implementation to find the maximum cliques in a graph. The algorithm uses three sets of vertices to build a clique step by step. The sets are named $R$, $P$, and $X$ and contain the following vertices:

- $R$: The set of vertices that are part of the current maximum clique,
- $P$: The set of vertices that are candidates for $R$,
- $X$: The set of vertices that are excluded for the current maximum clique.

Below is the pseudocode that described the Bron-Kerbosch algorithm:

\[
\text{BronKerbosch}(R, P, X): \\
\quad \text{if } P \text{ and } X \text{ are both empty:} \\
\quad \quad \text{report } R \text{ as a maximal clique} \\
\quad \text{for each vertex } v \text{ in } P: \\
\quad \quad \text{BronKerbosch}(R \cup \{v\}, P \cap N(v), X \cap N(v)) \\
\quad \quad P := P \setminus \{v\} \\
\quad \quad X := X \cup \{v\}
\]

In the first call to the algorithm, $R$ and $X$ are empty; The current maximum clique is yet to be found and there are no vertices to exclude. $P$ contains all the vertices of the graph. For every vertex in $P$, it will select it as a candidate $v$, and place it in $R$. It then only retains those vertices in $P$ which are the neighbors of the $v$. This is because a vertex not connected to $v$ cannot be part of the proposed maximum clique in $R$. After a recursive call is made with the new sets, $v$ is removed from $P$ and it added to the excluded set $v$. When there are no more candidates remaining and there no more vertices in $X$, $R$ is reported to be the maximum clique.

Design and Operation of the Parallel Program

The Bron-Kerbosch works on by computing the sets $R$, $P$ and $X$ and checking for the cliques while computing on these sets. We could visualize the recursion tree as given below, where every thread explores through a subtree of the recursive tree searching for the cliques and reports the maximum at the end after the reducing the individual results.
We send these configurations to each core or the worker to start the processing of the algorithm. Each of the core then computes on the configuration it has been provided with and reports back the maximal clique it has found in recursive call with the algorithm.

To explore the program’s speedup, we had constructed the algorithm to work on both, a multicore and cluster platform. Since there generally would be more configurations than number of threads, every thread is instructed to explore a set of configurations. In our first implementation, the configurations were constructed as a preprocessing step. The multicore program had these configurations stored as a common datastore for all the threads. The nodes of a cluster received these configurations in the form of tuples sent by the master node. Improvements in the program structure eliminated the time taken to transport the tuples; a procedure was written to compute a configuration based on the selected vertex.

We started with the configuration creation in the shared memory, so the multicore program was our first preference to work with. But since then we eliminated the need to create the configurations in
sequential manner before the execution of the parallel part, we created a multi-cluster program to have good flexibility with adding more nodes and architecture if needed.

As we explained in presentation - 4, our initial parallel multicore and cluster implementations ran accurately and got the right results but failed to give good results when it came to scaling. So we decided to implement a variant, which we termed version 2, for both multicore and cluster. In version 1, we explored one level deep for each vertex in the graph. It became apparent that this was leading to worse speed up that expected, since the tree produced during the algorithm’s execution was always deeper for certain nodes. In order to remedy this (at least partially), version 2 went two levels deep into the graph. As can be seen from the results below, this end up producing significantly better strong scaling results. Unfortunately, weak scaling remained the same.

Developer's Manual

1. You will require Java version 1.8 or higher to compile the source code.
2. Download the pj2 library from https://www.cs.rit.edu/~ark/pj2.shtml
3. Set the classpath to pj2 library by using command:
   export CLASSPATH=.:<PATH_TO_PJ2_JAR>
4. Change the directory to the src from the project folder
5. Compile all the files using Java compiler:
   javac *.java
6. This step is to execute the program in cluster only: create a jar file with the generated classes. To create the jar file, execute:
   jar cf prj.jar *.class
7. You are good to go!

User's Manual

After downloading the project folder you will have all the graph resources in the 'res' folder.
Steps for cluster execution:

1. Run with:
   java pj2 jar=pro.jar debug=makespan workers=<$W>$ <FUNCTION> <RES>
   <$W>$ = number of workers you want to create (1 and above)
   <FUNCTION> = MaximumCliqueClusterMassivelyParallel (version 1)
   = MaximumCliqueClusterMassivelyParallel2 (version 2)
   <RES> = one of the test file from the RES folder with the complete address to the directory
"CompleteGraph(<K>)" <K> is the number of connected vertices you want to create in the graph.

Steps for multi-core execution:

1. Run with:
   ```
   java pj2 debug=makespan cores=<C> <FUNCTION> <RES>
   ```

   <C> = number of workers you want to create (1 and above)

   <FUNCTION> = MaximumCliqueMulticoreParallel (version 1)
     = MaximumCliqueMulticoreParallel2 (version 2)

   <RES> = one of the test file from the RES folder with the complete address to the Directory
   = "CompleteGraph(<K>)" <K> is the number of connected vertices you want to create in the graph.
Strong Scaling Performance Data

As can be seen from the graphs below, the efficiency drops off quite rapidly and there is hardly any improvement in run time even after increasing the number of cores.

Multicore (V1)
Cluster (V1)

Similarly, as can be seen from the graphs below, the efficiency drops off quite rapidly here as well, and there is hardly any significant improvement in run time even after increasing the number of ndos. But the performance seems to be more uniform across different data sets.
Multicore (V2)

As can be clearly seen, the efficiency and speedups are significantly better here compared to V1. But the overall run times have gone up. There seems to have been a trade off of run time vs speed up.
Cluster (V2)

Similar to the multicore graphs above, cluster v2 is a significant improvement over cluster v1 in terms of efficiency and speed ups achieved. The cost again seems to be present in the form of run times.
Possible Reasons For Non-Ideal Strong Scaling

The algorithm designed works using three core sets of vertices. These vertices are stored in the program using HashSet objects which are then encapsulated by a Configuration object used at a particular level. For every recursive call, the HashSets are cloned into a new Configuration object. When running with limited memory, Java will invoke the garbage collector to remove objects in memory marked for deletion. Operating on a graph large enough will generate a huge amount of objects and invoke the garbage collector quite frequently. Looking at our test results, we largely suspect this to be the reason for our speedup, or rather the lack of it.

Depending on the structure of the graph, certain configurations would have a dense subtree compared to others. Our algorithm distributes configuration vertices to threads without taking this into account. Hence, with no appropriate load balancing algorithm, some cores remain idle after execution, waiting for the cores with dense configurations to finish. A better distribution of configurations can definitely yield a better runtime with an acceptable speedup.
Weak Scaling Performance Data

The graphs shown below deal with weak scaling performance for our Parallel Cluster v2 algorithm. As can be seen from the graphs below, weak scaling has remained bad even after we switched from v1. The graphs are similar for multicore v1 and v2 as well. Our algorithms does not seem to weakly scale.
Possible Reasons For Non-Ideal Weak Scaling

Since our algorithms do not take into account the degeneracy of the graph being passed in, the division of work among the cores is still far from ideal. There is no way for our algorithm to optimize or change its behaviour based on any degeneracy that can be exploited. Also, the nature of the data being used means that the degeneracy is nearly the same for all the vertices. So, it can be concluded that dense graphs are a bad proposition for our algorithms. It can show better behaviour with sparse graphs that vary widely in terms of degeneracy. It should be noted that the computational cost changes wildly with changes in the graph being used. And no matter what, the computational tree of execution is always going to be skewed towards the nodes or workers that are unfortunate enough to deal with those vertices that both ‘earlier’ on in the graph and happen to be well connected to other vertices in the graph. These are the workers and nodes that end up doing the processing from the start of the algorithm to the end, which is always undesirable.
Possible Future Work

Optimizations for the Bron-Kerbosch could definitely be looked at. Both the pivot based approach and the vertex ordering approach that places an onus on degeneracy could be more reliable candidates. Based on our findings, we believe that just applying these approaches naively may not lead to immediate improvements. The entire structure of the algorithm may need to be looked at in order to parallelize and scale it properly. Pre-processing the nodes in the graph to determine the degeneracies before hand could potentially lead to a payoff in terms of speed if the proper bounds are aggressively applied. This was an approach we decided to forego early on believing that it could add a lot of sequential dependency to our algorithm. In hindsight, maybe it could have been worth it. Map-reduce and GPU methods could be considered. Also, other methods outside of augmentations to the Bron-Kerbosch algorithm could be looked at. Variations of search algorithms such as genetic algorithms and local search could be attempted. Also, perhaps bit sets could be used instead of other structures to store the graphs and efficiently do set operations when required. Lastly, heuristic solutions could help reduce the time. They can be used as a stage in the overall algorithm so that a second exhaustive search can disregard entire neighbourhoods in the graph.

Takeaways From The Project

There are several useful and fulfilling things we can take away from this project. Firstly, we definitely improved our research skills and ability to pore over and absorb vast amounts of theoretical material from research papers. We felt that the paper analysis portion greatly enhanced our confidence when it came to going through papers and effectively distilling their content. Secondly, we learned how to approach parallelization of graph problems in general and the maximum clique problem specifically. We definitely improved our ability to detect sequential dependencies in algorithms and ways to effectively reduce those in our design. Finally, we also learned how to effectively combine different approaches that we found in various papers to create a combination of them.
Member Contributions

Vishal: Worked on the sequential implementation, cluster v1, Bron-Kerbosch v2, generated the graphs and tables.
Srinath: Worked on multicore v1, cluster v2, collected results for v1 program runs.
Akshay: Worked on multicore v2, Bron-Kerbosch v1, collected results for v2 program runs.
Every body contributed to the presentations and the final report. We each usually took up different sections that needed to be done and combined them towards the end. Overall, we feel that the workload was quite evenly distributed among us throughout the research investigation.

References