BTP - Project Report

Distributed Maximal Clique Computation

Maximal cliques are important substructures in graph analysis. Most of the proposed algorithms for computing maximal are sequential algorithms that cannot scale due to the high complexity of the problem. Most existing parallel algorithms for computing maximal cliques suffer from skewed workload. In this project, we took inspiration from the algorithm proposed by Yanyan Xu, James Cheng, Ada Wai-Chee Fu from Department of CSE, The Chinese University of Hong Kong and Yingyi Bu from Department of CS, University of California from 2014 IEEE International Congress on Big Data named same. This a distributed algorithm built on a share-nothing architecture for computing the set of all maximal cliques in a given undirected unweighted graph. This algorithm effectively address the problem of skewed workload distribution due to high-degree vertices, which also leads to drastically reduced worst-case time complexity for computing maximal cliques in common real-world graphs.

Clique : Consider an undirected unweighted graph G = (V, E). C, a subset of vertex set V, is called a clique if every vertex in C is connected to every other vertex in C by an edge in G.

Maximal Clique : The clique C defined above is called a maximal clique if any proper superset of C is not a clique.

The problem of **Maximal Clique Enumeration** is to compute the set of all maximal cliques in G.

A graph can have numerous maximal cliques. The count may increase drastically with increase in number of vertices and the edge density of the graph.

We study the problem of computing maximal cliques in a simple **undirected graph, G = (V,E)**, where **V** is the set of vertices and **E** is the set of edges of G. We keep G in its *adjacency list* representation. Each vertex $v \in V$ is assigned a unique vertex ID, denoted by **ID(v)**, where the vertex ID ranges from 1 to $|V|$. Given any two vertices u and v, we use *ID(u) < ID(v)* or equivalently *ID(v) > ID(u)* to denote that u is ordered before v according to the order of their IDs. In the adjacency list representation of a graph, vertices are ordered in ascending order of their IDs.

We define the set of adjacent vertices of a vertex $v \in V$ as **adj (v) = {u : (u, v) ∈ E}**. We further define **adj (<v) = {u : u ∈ adj (v), ID(u) < ID(v)}** and **adj (> v) = {u : u ∈ adj (v), ID(u) > ID(v)}**. A set of vertices, C, where $C \subseteq V$, is a **clique** in G if every $v \in C$ is adjacent to all other vertices in C, i.e., $v \in adj(u)$ for all $u \in (C \setminus \{v\})$. If there is no $C' \supset C$ such that C' is a clique in G, then C is a **maximal clique**. We use **M(G)** to denote the set of maximal cliques in G. We also use $\mathbf{M_v}$ to denote the set of maximal cliques starting with v, i.e., $\mathbf{M_v = \{C : C \in M(G), v = argmin_{u \in C} ID(u)\}}$, where "$\mathbf{v = argmin_{u \in C} ID(u)}$" means "$v \in C$ such that **ID(v) =min{ID(u) : u ∈ C}**".

The algorithm consists of two phases: **data distribution** and **maximal clique enumeration (MCE)**, The data distribution phase is shown in Lines 1-6 of Algorithm 1. Given a simple undirected graph G = (V,E), the algorithm divides the task of MCE into many sub-tasks to be computed in parallel. The data necessary for MCE at each worker machine is to be distributed.

Algorithm 1: Parallel MCE

1 **Data distribution:**
  Input : $\langle v; adj(v)\rangle$ for each $v \in V$
2 **begin**
3 **foreach** *vertex* $v \in V$ **do**
4 output $\langle ID(v); (v, adj(v))\rangle$;
5 **foreach** *vertex* $u \in adj(< v)$ **do**
6 output $\langle ID(u); (v, adj(v))\rangle$;
7 **Maximal clique enumeration (MCE):**
  Input : $\langle ID(v); (v, adj(v), \{(u, adj(u)) : u \in adj(> v)\}\rangle$
  for each $v \in V$
8 **begin**
9 **foreach** $u \in adj(> v)$ **do**
10 $ADJ_{>v}[u] \leftarrow adj(u) \cap adj(> v)$;
11 $ADJ_v[u] \leftarrow adj(u) \cap adj(v)$;
12 LocalMCE$(\{v\}, adj(> v), adj(< v), ADJ_{>v}, ADJ_v)$;

Algorithm 2: LocalMCE$(C, cand, prev, ADJ_{>v}, ADJ_v)$

1 **if** $cand = \emptyset$ *and* $prev = \emptyset$ **then**
2 output C as a maximal clique;
3 **else if** $cand \neq \emptyset$ **then**
4 let u_p be the vertex in $cand$ that maximizes $|cand \cap ADJ_{>v}[u_p]|$;
5 $U \leftarrow cand \setminus ADJ_{>v}[u_p]$;
6 sort U in descending order of $|ADJ_{>v}[u]|$ for all $u \in U$;
7 **foreach** $u \in U$ **do**
8 $cand \leftarrow cand \setminus \{u\}$;
9 $cand' \leftarrow cand \cap ADJ_{>v}[u]$;
10 **foreach** $w \in cand'$ **do**
11 $ADJ'_{>v}[w] \leftarrow ADJ_{>v}[w] \cap cand'$;
12 $ADJ'_v[w] \leftarrow ADJ_v[w] \cap prev$;
13 LocalMCE$(C \cup \{u\}, cand', prev \cap ADJ_v[u], ADJ'_{>v}, ADJ'_v)$;
14 $prev \leftarrow prev \cup \{u\}$;

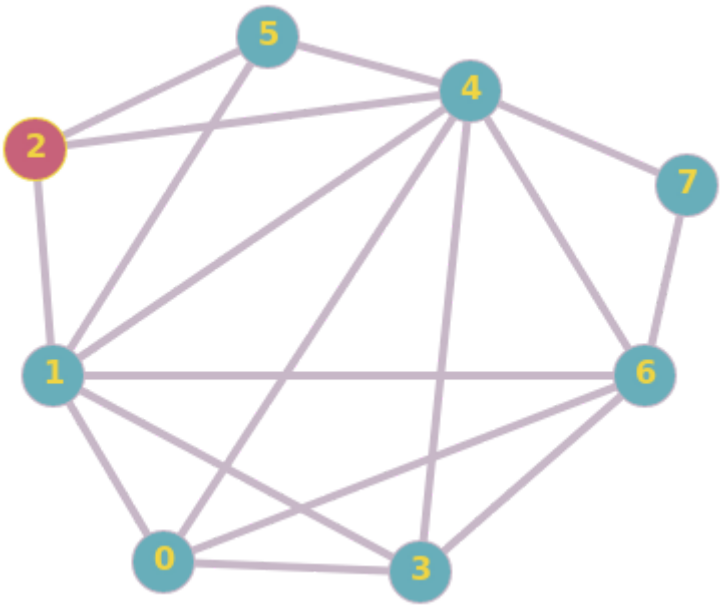
Note that for each C ∈ M_v , **(C \ {v}) ⊆ adj(> v).** Thus, to enumerate the maximal cliques in M_v , we only need **adj(u) ∩ adj(> v),** denoted by **$ADJ_{>v}$[u]**, for each u ∈ adj(> v). However, to check maximality of the cliques, we also need **adj(u) ∩ adj(v)**, denoted by **ADJ_v[u]**, for each u ∈ adj(>v).

We first explain some notations used in Algorithms 2. We use C to denote the clique currently being enumerated, **cand** to denote the set of candidate vertices that can be used to expand or form a clique, and **prev** to denote a set of vertices that are in some other maximal cliques (either enumerated previously by the same worker or enumerated by another worker) so that C is maximal only if prev = ∅. We also use $ADJ_{>v}$ and ADJ_v to denote the sets **{$ADJ_{>v}$[u] : u ∈ adj(> v)}** and **{ADJ_v[u] : u ∈ adj(> v)}**, respectively.

The LocalMCE algorithm starts from a set C initially consisting of a single vertex, and repeats the process “find a candidate vertex u ∈ cand that is a common neighbor of all vertices in the current C and then add u to C” until there exists no common neighbor of the current C, in which case cand = ∅, and C is returned as a maximal clique if prev = ∅. When we grow the current clique C to C = (C ∪ {u}), we refine cand by intersecting it with $ADJ_{>v}$[u] because any candidate vertex that can grow C must be in $ADJ_{>v}$[u]. We also refine prev by intersecting it with ADJ_v[u] because if another maximal clique C exists such that C cannot be grown into a maximal clique in the end, then (C \ C) must be a subset of ADJv[u]. For the same reasons, we also refine $ADJ_{>v}$[w] and ADJ_v[w] for each new candidate vertex w ∈ cand, by intersecting them with cand and prev, respectively. Then, LocalMCE is invoked recursively to further grow C.

Sample Graphs

Graph with 8 vertices and 17 edges

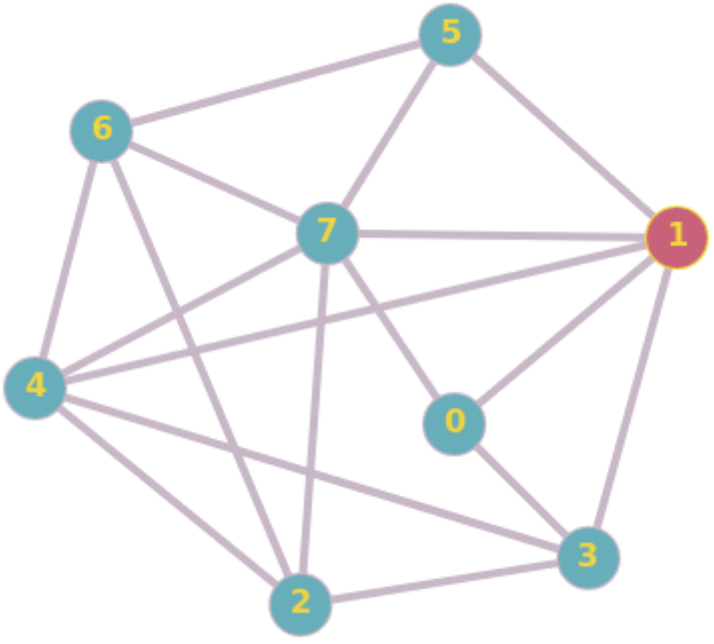


Graph in adjacency list form:

8
4 1 3 4 6
6 0 2 3 4 5 6
3 1 4 5
4 0 1 4 6
7 0 1 2 3 5 6 7
3 1 2 4
5 0 1 3 4 7
2 4 6

Output:
7 4 6
2 1 4 5
0 1 3 4 6

Graph with 8 vertices and 17 edges



Graph in adjacency list form:

8
3 1 3 7
5 0 3 4 5 7
4 3 4 6 7
4 0 1 2 4
5 1 2 3 6 7
3 1 6 7
4 2 4 5 7
6 0 1 2 4 5 6

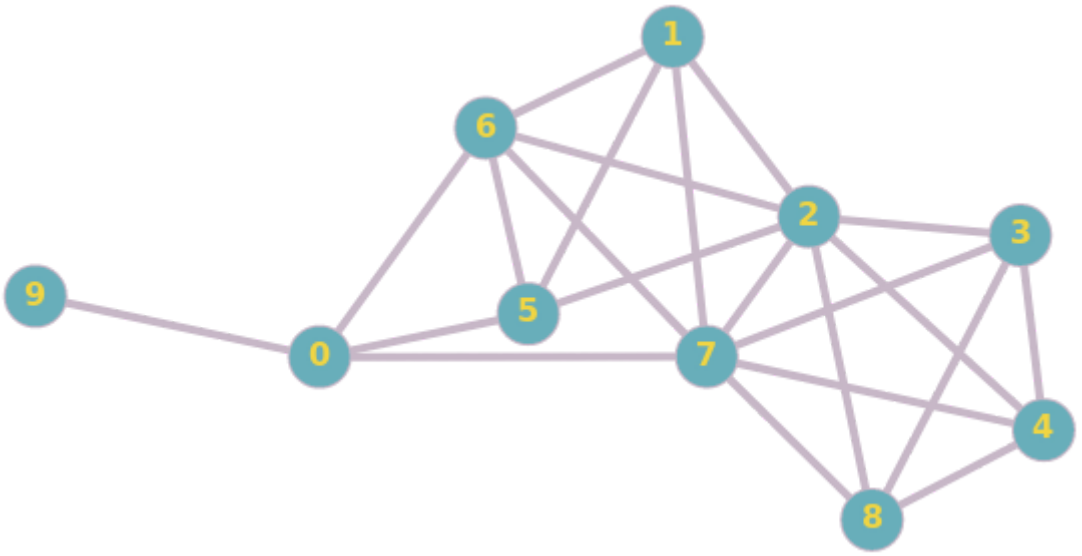
Output:

0 1 7
5 7 1
2 4 3

0 1 3
5 7 6
2 4 6 7

3 1 4 1 4 7

Graph with 10 vertices and 22 edges

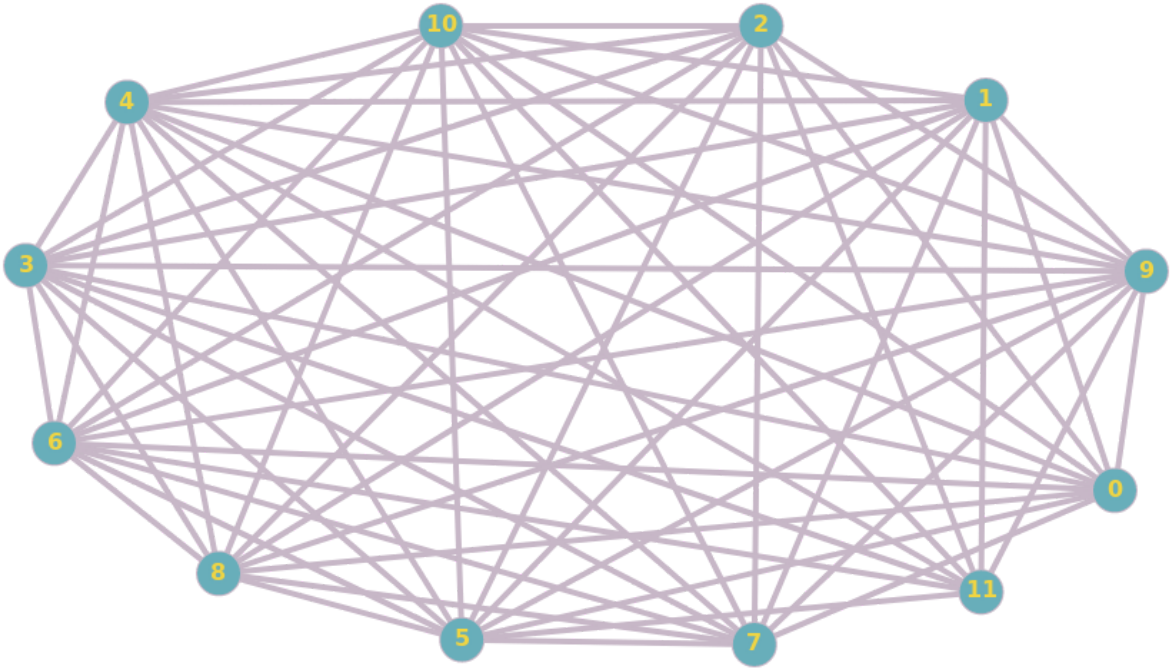


Graph in adjacency list form:

10
4 5 6 7 9
4 2 5 6 7
7 1 3 4 5 6 7 8
4 2 4 7 8
4 2 3 7 8
4 0 1 2 6
5 0 1 2 5 7
7 0 1 2 3 4 6 8
4 2 3 4 7
1 0

Output:
0 6 7
0 6 5
1 2 6 7
1 2 6 5
3 2 4 7 8

Graph with 12 vertices and 62 edges



Graph in Adjacency list form:

12
10 1 2 3 4 5 6 7 8 9 10
10 0 3 4 5 6 7 8 9 10 11
10 0 3 4 5 6 7 8 9 10 11
11 0 1 2 4 5 6 7 8 9 10 11
11 0 1 2 3 5 6 7 8 9 10 11
11 0 1 2 3 4 6 7 8 9 10 11
11 0 1 2 3 4 5 7 8 9 10 11
10 0 1 2 3 4 5 6 8 9 10
10 0 1 2 3 4 5 6 7 9 10
11 0 1 2 3 4 5 6 7 8 10 11
11 0 1 2 3 4 5 6 7 8 9 11
8 1 2 3 4 5 6 9 10

Output:
11 3 4 5 6 9 10 1
11 3 4 5 6 9 10 2
0 3 4 5 6 7 8 9 10 1
0 3 4 5 6 7 8 9 10 2

Implementation:

We used MPI(C) for the implementation of the above algorithm with master-slave architecture.
As per the algorithm we need to compute the cliques for each vertex ‘v’ wherein ‘v’ is the vertex with the least degree.
So, if we through master send the vertex ‘v’ and the data required to find corresponding cliques to a slave process, it does the computations and enumerates the corresponding cliques. So, we can run multiple processes simultaneously. There is no communication between the slaves and they do not share any common memory justifying the distributed nature of the algorithm.
The master process in our mpi code **btp1.c** sends data of each of the vertices to one of the processes in every round starting from the first process. Then it waits until the the first process completes to send the data of next vertex to it. The data is sent in order i.e in each round first process should receive data before data is sent to the second process and so on.
As per the algorithm, the vertex with higher degree is most likely to have enumerated many of the cliques in which it is member when a vertex with lower degree is used for enumeration. In this case, when a vertex(i.e the process performing its computations) takes much more time to complete its computations than other vertices, then all other processes will be completed and have to wait until this process completes in order to receive data for their next computations.
So, we wanted to change the code (**btp2.c**) so that such scenario does not occur. For this, we made the following changes. Initially, the master sends the data of ID of the vertices to all the slave processes. Then slave processes send a message that they are ready for computation and the master sends the data to the processes in FCFS basis. After performing its computations, the slave once again sends a message to the master that it is ready for further computations and the master sends the data of next vertex and so on. Hence no process has to wait and resources can be utilised more efficiently.

Time complexity:

The time complexity of the algorithm is $O(h.3^{h/3})$ where h is the maximum value of h such that there are at least h vertices with degree at least h.
Consider $|adj(> v)| \leq h$ for all $v \in V$.
Suppose on the contrary that there exists a vertex $v \in V$ such that $|adj(> v)| > h$. Since $|adj(> v)| > h$, there are at least (h+1) vertices that are ordered after v, i.e., they have degree at least as large as v. Since $|adj(v)| \geq |adj(> v)| > h$, v has degree at least (h+ 1) and hence each $u \in adj(> v)$ has degree at least (h + 1). This means that there are at least (h + 1) vertices that have degree at least (h + 1), which contradicts to the fact that h is the maximum value of h such that there are h vertices with degree at least h. Thus, $|adj(> v)| \leq h$ for any $v \in V$.

When we are computing a maximal clique starting with a vertex v, the possible vertices in the clique are u, u ∈ adj(>v). Hence h is a bound to the number of potential vertices that are to checked for finding a clique.
I

An overview of ADA cluster of IIIT-H : Ada cluster consists of forty Boston SYS-7048GR-TR nodes equipped with dual Intel Xeon E5-2640 v4 processors, providing 40 virtual cores per node, 128 GB of 2400MT/s DDR4 ECC RAM and four Nvidia GeForce GTX 1080 Ti GPUs, providing 14336 CUDA cores, and 44 GB of GDDR5X VRAM. The nodes are connected to each other via a Gigabit Ethernet network. All compute nodes have a 1.8 TB local scratch and a 960 GB local SSD scratch. The compute nodes are running Ubuntu 16.04 LTS. SLURM software is used as job scheduler and resource manager. The aggregate theoretical peak performance of Ada is 30.72 TFLOPS (CPU) + 1760 TFLOPS (FP32 GPU).

Some implementation results:

For a graph of 80 vertices:

no. of edges = 2854
maxDegree = 78
h = 69
Using code btp1.c:

80 vertices 40 cores on ada (1 master, 39 slaves compute simultaneously)

real 2m53.816s
user 7m33.192s
sys 2m49.356s

80 vertices 10 cores on ada (1 master, 9 slaves compute simultaneously)

real 2m45.640s
user 15m8.744s
sys 3m14.180s

80 vertices 4 cores on ada (1 master, 3 slaves compute simultaneously)

real 5m40.206s
user 17m9.544s
sys 3m51.856s

Using code btp2.c:

80 vertices 40 cores on ada (1 master, 39 slaves compute simultaneously)

real 1m54.980s
user 18m1.712s
sys 3m11.320s

80 vertices 10 cores on ada (1 master, 9 slaves compute simultaneously)

real 2m28.948s
user 15m37.288s
sys 3m5.568s

80 vertices 4 cores on ada (1 master, 3 slaves compute simultaneously)

real 5m13.436s
user 15m39.072s
sys 3m36.768s

The number of maximal cliques was found to be 19 million cliques which indicates the complexity of the problem. Size of output file was 1.2 GB.

For a graph of 100 vertices:

no. of edges = 4442
maxDegree = 95
h = 85

Using code btp1.c:

100 vertices 40 cores on ada (1 master, 39 slaves compute simultaneously)

real 47m17.951s
user 714m15.088s
sys 230m7.320s

100 vertices 10 cores on ada (1 master, 9 slaves compute simultaneously)

real 57m43.419s
user 556m21.668s
sys 68m34.596s

Using code btp2.c:

100 vertices 40 cores on ada (1 master, 39 slaves compute simultaneously)

real 48m16.620s
user 501m50.128s
sys 271m16.136s

The above two graphs are too dense and taken from standard test datasets.
There will be few vertices (say k) enumeration whose corresponding maximal cliques will take a lot of time. So, if the number of process is increased up to ‘k’, the time taken will decrease but after that there will not be much effect on time for the above graphs.
The time reduced when we increased the number of slaves from 3 to 9 but not much improvement is found when we increase the number of slaves from 9 to 39 for the graph 80 vertices h = 69.
For the above two graphs, most of ‘h’ vertices are interconnected and hence time complexity is high.
For the below graph ‘F’ of facebook with 1400 vertices and h = 68, the time taken is lesser compared to graph ‘G’ 80 vertices h = 69 because the ‘h’ vertices of ‘F’ are not interconnected so densely as the ‘h’ vertices of G. If the ‘h’ vertices of ‘F’ are like two clusters with very less interconnections between these clusters, effective ‘h’ will only be ‘h/2’.
It is to be noted that ‘h’ value indicates only the upper bound which is the worst case scenario where most vertices in ‘h’ are connected to each other.

***** For a real world graph - facebook with 1400 vertices:

No. of vertices = 1400
No. of edges = 16156
Max degree = 533
h = 68

Using code btp1.c:

no, of CPU dedicated= 2, no. of processes = 2

real 7m21.642s
user 8m24.164s
sys 6m18.456s

no, of CPU dedicated= 3, no. of processes = 3

real 4m23.044s
user 7m24.552s
sys 5m44.324s

no, of CPU dedicated= 10, no. of processes = 10

real 1m46.509s
user 11m43.644s
sys 6m0.768s

no. of CPU dedicated= 40, no. of processes = 40

real 0m56.087s
user 27m10.136s
sys 9m52.324s

Using code btp2.c:

no, of CPU dedicated= 2, no. of processes = 2

real 5m11.827s
user 6m6.336s
sys 4m18.024s

no, of CPU dedicated= 3, no. of processes = 3

real 1m58.871s
user 2m46.260s
sys 3m11.308s

no, of CPU dedicated= 4, no. of processes = 4

real 1m18.416s
user 2m7.144s
sys 3m6.988s

no, of CPU dedicated= 10, no. of processes = 10

real 0m24.635s
user 1m25.452s
sys 2m40.440s

no, of CPU dedicated= 40, no. of processes = 40

real 0m7.331s
user 1m36.092s
sys 1m41.524s

For this graph, the vertices are not connected like a cluster as in the first two graphs (80, 100 vertices) . The computational complexity does not depend only on a few vertices. Hence the improvement in results with increase in the number of processes. Also, the code btp2.c shows improvement from btp1.c in scenarios where there are many processors and they have to wait for other processor to finish computations as per btp1.c (explained before) which does not occur in btp2.c for any number of processors.
The code btp2.c ensures all the available resources are used efficiently.

For a real world graph - facebook with 2000 vertices:

No. of vertices = 2000
no. of edges = 37645
max degree = 1045
h = 120

no. of CPU dedicated= 40, no. of processes = 40

real 25m11.432s
user 214m38.776s
sys 111m14.504s

Note: An interesting observation was found when I put a print statement between two successive receive statements for some analysis and it took about 30 seconds for facebook_1400 graph on 40 processors, while it took only 6-7 seconds when the print statement was removed. It might be due to the switch between data transfer modes frequently and due to buffering of stdout.