Maximum Clique Problem

Dler Ahmad
dha3142@rit.edu

Yogesh Jagadeesan
yj6026@rit.edu

1. INTRODUCTION

Graph is a very common approach to represent computational problems. A graph consists a set of vertices V connected through a set of edges E. For instance, cities and the distances among them can be represented via a weighted graph where a vertex demonstrates a city and an edge indicates the distance between each pair of cities. A graph can be either *partially-connected* or *fully-connected*. A partially-connected graph have some pairs of vertices which no edge connects them. In our example, this may be the case where there is no path from a city to another one. On the other hand, in a fully-connected graph (also called complete graph), all pairs of vertices are connected via an edge.

Moreover, a graph can have sub-graphs that are either partially or fully connected. A sub-graph of a graph that is complete is called clique. Figure 1 illustrates a clique within a graph. An example for a clique application can be a graph representing a social network in which the graph's vertices represent people, and the graph's edges represent mutual acquaintance. A clique within this graph would indicate a subset of people who all know each other.

A graph may contain many cliques, which may vary in size. The largest clique within a particular graph is called *maximum clique* of that graph. In our social network example, the maximum clique of the graph would represent the largest subset of the network population in which all people in that subset know each other. Along with its application in social networks, maximum clique problem has many other applications in data analysis, computer vision and bioinformatics.

In this paper, we will investigate some of the suggested approaches to find the maximum clique in a graph. We will introduce a sequential, a multicore parallel and a cluster parallel program to solve maximum clique problem. We will discuss how to compile and run each version of the program. Finally, we will analyze the result and the performance of each version in both strong scaling and weak scaling performance analysis methods.

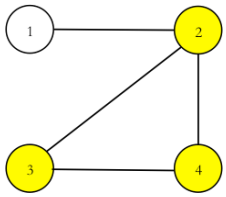


Figure 1. A clique in graph

2. PROBLEM

The maximum clique problem is an NP-complete problem [1]. Let G=(V,E) be an undirected graph where V={1,2,...,n} is the set of vertices in G, n=|V| is the size of the set V and E ⊆ V ×V is the set of edges in G. A graph G is complete if all its vertices are connected. i.e. every i, j ∈ V such that i ≠j , (i,j) ∈ E. A clique C is a subset of V, such that graph G(C) is complete. The cardinality of C is the number of vertices in the C. The maximum clique problem is the problem of finding clique C with maximum cardinality in graph G.

3. RELATED WORK

Different approaches have been introduced to solve maximum clique problem. While some of them may share similar ideas, others are distinctive. We now study three of these approaches in detail.

3.1 K-opt Local Search

One way to find the maximum clique in a graph is to perform a local search on a candidate graph. That is, given graph G, an initial vertex v from the set of vertices V in the G is picked. This v is considered the minimum clique CC (Current Clique) in G. Then all the neighbor vertices of v will be examined for being added to CC. If a neighbor vertex is connected to all vertices in CC, then it will be added. This procedure is continued until no other vertex can be added to the CC.

The problem with local search is that the solution may not always be optimal. Depending on the initial CC, the local search may find a clique that is not maximal. Consider the graph shown in figure 2. If the local search picks edge A as the initial CC, after adding vertex 3, there will be no neighbor vertex that is connected to all vertices in CC. Thus, the CC can not be expanded any more and the final clique of size 3 will be reported as the solution. However, if the local search picks edge B as the initial CC, it can be observed that a solution of size 4 will be reported. This observation proves that local search may return non-optimal solution occasionally.

The research paper Katayama et al [2], tries to remedy the local search non-optimality by introducing a local search based algorithm called k-opt local search. It uses a special vertex-dropping mechanism to enlarge the clique whenever the CC cannot be expanded any more. To do that, the algorithm keeps a list of vertices OM containing vertices that miss one edge to CC. i.e. the vertices that are connected to |CC|-1 vertices of CC. Whenever CC cannot be expanded any more, the algorithm randomly picks a vertex v in OM and drops the vertex in CC that is not connected to v. Then the algorithm will perform another local search with the new CC. If expanding the CC is still not possible, the dropping procedure will be repeated. The algorithm will return CC as the solution when OM={∅}.

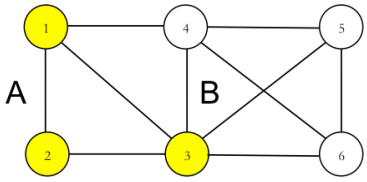


Figure 2. local search solution

3.2 Iterated K-opt Local Search

Surprisingly, the k-opt local search may not always return optimal solution either. Consider the graph shown in the figure 3. This graph is similar to graph in figure 2 with an added sub-graph. If we run k-opt local search algorithm on this graph with

same initial CC as previous section, after adding vertex 6, the k-opt local search cannot expand the CC any more and it will return a clique of size 4 as the maximum clique. However, it can be observed that a larger clique exists in the graph.

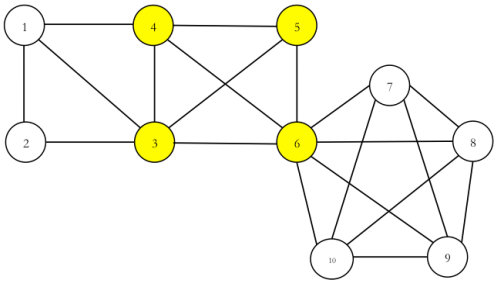


Figure 3. K-opt local search solution

The paper Katayama et al [3], tackles the non-optimality problem of k-opt local search, by introducing its iterative version called iterated k-opt local search (IKLS). IKLS tries to enlarge the CC by running k-opt local search multiple times. After first run, the result is saved as Best-Clique. Then, IKLS adds and drops some vertices using a technique called Lowest-Edges-Connectivity-based Kick (LEC-Kick for short). LEC-Kick procedure is done in three steps. First, one vertex with lowest connectivity to the CC is picked. i.e. a vertex v in V that is connected to least number of vertices in CC. If there is more than one vertex with same low-connectivity degree, one will be picked randomly. Second, the chosen vertex v is added to CC. Third, all vertices in CC, which are not connected to v, will be dropped. After LEC-Kick is done, IKLS will run another k-opt local search on the newly formed initial CC. If a larger clique is found that is larger than Best-Clique, it will be saved as Best-Clique; otherwise another LEC-Kick followed by another k-opt local search is performed. IKLS will return the solution only after termination condition is satisfied. The termination condition can either be a clique of a desired size or the number of times IKLS runs k-opt local search.

In addition to the LEC-KICK, IKLS occasionally performs another technique called *RESTART*, to diversify the clique search space. The RESTART process is performed when no larger clique is found after |Best-Clique| iterations where |Best-Clique| is the size of best clique found so far. In that case, a single vertex in Best-Clique is randomly selected as an initial CC and the IKLS is started over.

3.3 Adoptive Diversification Strategy

As we have seen, heuristic approaches to solve maximum clique problem can lead to less than optimum solution. Too much focus on search intensification (exploring the search area in depth-wise fashion) can lead to getting stuck at a local optimum. On the other hand, extensive diversification (exploring different search area by jumping to a rather far node from current search location) may not guarantee to explore a deep enough search space to find a global optimum solution. It seems a balance is required between intensified and diversified searches to increase the chance of discovering a global optimal solution.

Research paper Benlic et al [4], targets this issue by introducing *Perturbation Moves* to enlarge the clique. Perturbation Moves are moves that would get the search out of a current search space and redirect it to a potentially new search space. The authors suggest two distinct Perturbation Moves named *Directed Perturbation* and *Random Perturbation*. Directed Perturbation moves are based on a list called *Tabu list* containing prohibited moves. Tabu list is updated in each iteration and contains vertices which are believed will not help in obtaining a larger clique. On the other hand, Random Perturbations are moves that take the search out of the current search space in order to explore a potentially new search space. In order to achieve a balance between directed and random perturbation, following probabilistic approach is applied. The algorithm keeps track of the number of non-improving visited neighbors ω. For smaller values of ω, directed perturbation is applied more often. However, as ω increases, directed perturbation moves are progressively decreased and the number of random perturbation moves is increased.

The results shows that adoptive diversification strategy combined with iterated local search, outperforms the standard iterated local search based on random moves on almost all the test cases and led to a larger solution.

4. DESIGN

We have developed three versions for our solution for maximum clique problem: A sequential version, which runs on a single machine without any parallelism, a multicore parallel version, which runs on a single machine and takes advantage of all its cores in parallel, and a cluster parallel version, which runs on a cluster with multiple nodes. We now look at each of these versions in detail.

4.1 Sequential Version

MaxCliqueSeq program is the sequential version of our solution for maximum clique problem. This version runs on a single machine and has no parallelism in it. The program starts out by validating the command line arguments. The arguments are *filename,* specifying the file that contains the input graph information, *seed,* a random number generator seed, and *numberOfIterations,* the number of random restarts on the graph. After validation process, the program variables are initialized and the specified graph file is read. There are two complementary classes in our solution. Class *Graph*, which consists required fields and methods for a graph object and class *Clique* that holds fields and methods for a clique object. A graph object can be constructed by passing the file in which the graph information is stored. These files needs to be in the following format:

<NumberOfVertices><space><NumberOfEdges>
<vertex><space><vertex>
<vertex><space><vertex>
<vertex><space><vertex>
…

The first line of the file specifies the number of vertices and the number of the edges in the graph. From the second line on, each line specifies pairs of vertices, which are connected via an edge. For instance, a line with <0><space><2> means vertex 0 is connected to vertex 2. Meanwhile, since all graphs used in our investigation are undirected graphs, this line also indicates that there is an edge, connecting vertex 2 to vertex 0.

After the graph is constructed, the primary part of the program starts. The program attempts *numberOfIterations* times to find a maximum clique in the graph. Each attempt, starts by picking a random vertex from the graph, which forms initial CC (current clique). Then the program checks all the neighboring vertices of picked vertex to see if they can be added to the CC. If a vertex has equal degree as the |CC| and has an edge to every vertex in CC, then it is considered a candidate vertex to be added to CC. However, of all neighboring candidates of the initial CC, only one with highest degree will be eventually added. If multiple candidates have equal degree, one will be picked randomly. At this point, the procedure is repeated for the

neighboring vertices of the newly added vertex. The iteration will finish when no more candidates can be found.

At the end of each iteration, the newly discovered clique will be compared with the so far best-found clique. If it is larger, it will be stored as the best-found clique. Finally, after all iterations are done, the result, which includes the vertices and the size of the best clique, will be printed.

4.2 Multicore Parallel Version

MaxCliqueSmp program is multicore parallel version of our solution for maximum clique problem. This version runs on a single machine and takes advantage of all its cores in parallel. Similar to sequential version, the multicore parallel version starts out by validating command line arguments. The arguments are *filename,* specifying the file that contains the input graph information, *seed,* a random number generator seed, and *numberOfIterations,* the number of random restarts on the graph. After validation process, the program variables are initialized and the graph object is constructed. Now its time for the program to attempts *numberOfIterations* times to find a maximum clique in the graph. This is the part where the multicore parallel version differs from sequential version. Since there are no sequential dependencies between the attempts, each can be performed on a single core of the host machine. To do that, instead of a regular for loop, a parallel for loop is used to perform the iterations. The iterations are divided among the cores. Each core will do a multiple random restart and would find a potential maximum clique. At the end, the maximum cliques found by each core are reduced to find the largest clique among them. Finally, the result, which includes the vertices and the size of the best clique, will be printed.

4.3 Cluster Version

MaxCliqueClu program is cluster parallel version of our solution to maximum clique problem. This version runs on a cluster with multiple nodes. The cluster parallel version follows the Master/Worker paradigm for cluster programming in which the master will distribute the computations among the workers. Also, the nodes in the cluster, communicate with each other through tuple space. All the nodes in the cluster can access objects in tuple space.

The cluster parallel program has two static inner classes, *WorkerTask* and *ReduceTask*. The WorkerTask class, defines the task of each worker which is to perform one attempt to find a maximum clique in the graph. The ReduceTask class will reduce preliminary result of all nodes to a final result and prints it. In the main program, first the command line arguments are validated. The arguments are identical to previous versions. Then after initializing the program variables, the graph object is constructed and is put into the tuple space so that it is accessible by all nodes in the cluster. Then using a *masterFor* loop, the iterations are distributed among the workers. Each worker will be assigned one iteration at a time. If a worker is done with one iteration, it will be assigned another one. This procedure continues until all iterations are performed. Then each worker puts its clique into tuple space and ReducerTask will reduce preliminary cliques of all workers and finds the largest one among them. Ultimately, the final result, which is the largest clique found by all workers, will be printed.

5. USE

5.1 Compile The Programs

To compile the sequential and multicore parallel versions, you need to login into one of CS multicore parallel computers and set your java class path to include the PJ2 library and your command path to include the JDK 1.7 installation. You can use following commands in bash shell to set up the class path and command path:

```
$ export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
$ export PATH=.:/usr/local/dcs/versions/jdk1.7.0_11_x64/bin:$PATH
```

Then, to compile the sequential program, use following command in the directory where *MaxCliqueSeq.java* is located:

```
$ javac MaxCliqueSeq.java
```

To compile the multicore parallel program, use following command in the directory where *MaxCliqueSmp.java* is located:

```
$ javac MaxCliqueSmp.java
```

To compile the cluster parallel program, you need to log into CS cluster parallel computer, *tardis.* To set up the class path and command path on tardis, use the same commands as above. Then to compile the program, use following command in the directory where *MaxCliqueClu.java* is located:

```
$ javac MaxCliqueClu.java
```

In addition to the program files, you also need to compile the *Graph.java* and *Clique.java* located in the same directory of the program file. These files can be compiled using following command:

```
$ javac Graph.java Clique.java
```

5.2 Run The Programs

Before running the programs, you need to make sure that the each program class file and Graph.class and Clique.class are located in the same directory.

The sequential program can be run by typing following command in the directory where MaxCliqueSeq.class is located:

```
$ java pj2 MaxCliqueSeq <filename> <seed> <numberOfIterations>
```

<filename> is the path to the file containing the graph information, <seed> is a seed number of type long for random number generator, and <numberOfIterations> is a number of type int for the number of random restarts in the graph.

The multicore parallel program can be run by typing following command in the directory where MaxCliqueSmp.class is located:

```
$ java pj2 threads=<numOfThreads> MaxCliqueSmp <filename> <seed> <numberOfIterations>
```

<numOfThreads> is an optional parameter for the number of threads in the parallel thread team, and the rest of parameters are identical to sequential program parameters.

The cluster parallel program can be run by typing following command in the directory where MaxCliqueClu.class is located:

```
$ java pj2 threads=<numOfThreads> workers=<numOfWorkers> jar=<nameOfJar> MaxCliqueClu <filename> <seed> <numberOfIterations>
```

<numOfThreads> is an optional parameter for the number of threads in the parallel thread team, <numOfWorkers> is an optional parameter number of workers to be involved in the computation, and <nameOfJar> is a jar file that contains all program classes. The rest of parameters are identical to sequential program parameters.

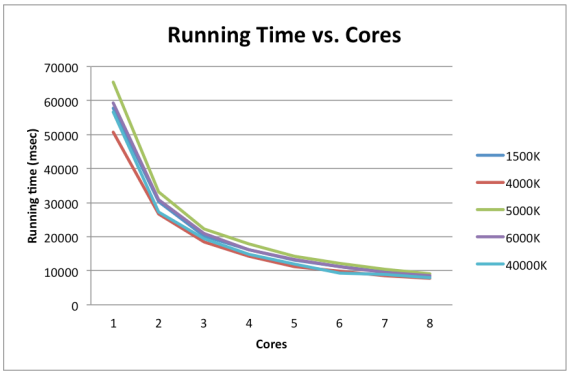
6. PERFORMANCE

We have measured the multicore parallel program efficiency with different problem size. The problem size for our solution would be the number of attempts to find a clique in the graph. This is the same number which user will specify when running the program. We now look at the program's strong scaling and weak scaling performance calculation.

6.1 Strong Scaling

For each instance of the strong scaling performance measurement, we increased the number of the threads that execute the program but we kept the problem size untouched. Figure 4 shows the result of our strong scaling performance calculation. As can be observed, increasing the number of threads has caused significant speed up in running time. In fact, in almost all cases, the efficiency of the program is very close to the ideal efficiency. However, the sequential portion of the program, which deals with reading the graph data from the file, causes the speed up and efficiency to drop from ideal ones.

V , E	N	K	T	Speedup	Eff
400,36990	1500000	Seq	60861		
		1	57734	1.054	1.054
		2	30325	2.001	1
		3	20155	3.02	1.006
		4	16128	3.774	0.943
		5	13075	4.655	0.931
		6	11214	5.427	0.904
		7	9615	6.33	0.904
		8	8415	7.232	0.904
800,46199	4000000	Seq	59097		
		1	50805	1.163	1.163
		2	26669	2.216	1.108
		3	18405	3.211	1.07
		4	14239	4.15	1.04
		5	11294	5.23	1.05
		6	9816	6.02	1.003
		7	8476	6.972	0.996
		8	7724	7.65	0.956
100,3109	5000000	Seq	65569		
		1	65502	1.001	1.001
		2	33185	1.976	0.988
		3	22326	2.937	0.979
		4	17981	3.646	0.912
		5	14256	4.599	0.92
		6	12135	5.403	0.9
		7	10361	6.328	0.904
		8	9086	7.216	0.902
200,7224	6000000	Seq	62745		
		1	59329	1.057	1.057
		2	30789	2.038	1.019
		3	20911	3	1
		4	16118	3.893	0.973
		5	13228	4.743	0.949
		6	11300	5.553	0.925
		7	9549	6.571	0.939
		8	8583	7.31	0.914
50, 399	40000000	Seq	62601		
		1	56670	1.104	1.104
		2	27256	2.296	1.148
		3	19341	3.237	1.079
		4	14812	4.226	1.056
		5	11958	5.235	1.047
		6	9320	6.717	1.195
		7	9019	6.941	0.991
		8	7882	7.942	0.993



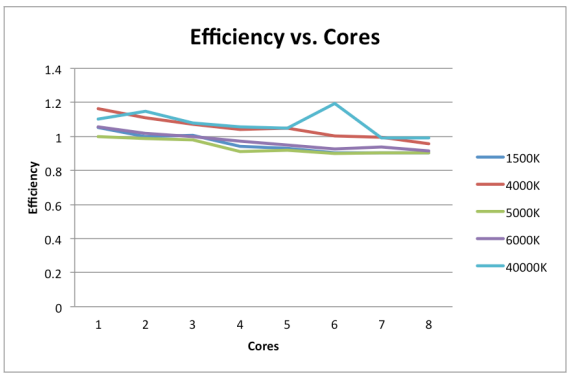


Figure 4. MaxCliqueSmp strong scaling performance metrics

6.2 Weak Scaling

For each instance of weak scaling performance measurement, as we increased the number of threads that execute the program, we also increased the problem size proportionately. Figure 5 shows the weak scaling performance metrics. It can be observed that, although we increase the problem size in every instance of measurement, the running times are more or less the same. This proves that, the computations are well-distributed among the executing threads, which eventually leads to a same running time. However, as seen for strong scaling, the sequential portion of the program causes the speed up and efficiency to drop from ideal ones.

7. FUTURE WORK

In this project, we worked with graphs that are specifically made for our purpose to see the performance of the program. However, we believe that more realistic graph problems can be solved using our program. For instance, we can apply the program on a graph that represents relationships in a social network and find out the largest subset of the network population who know each other. The solution will provide the opportunity for the users to find people who they know in real life.

Also, we can experiment our program on GPU accelerated machines. In that case, we would need to design another version of the program that takes advantage of large number of cores in GPU and possibly achieve a better performance.

8. PROJECT OUTCOME

While doing this project, we learned that a heuristic approach to solve maximum clique problem can lead to a less than optimal solution. However, for a large number of attempts to find a clique in the graph, we might be able to produce a result close to optimal solution. This is due to the fact that starting to grow a clique from one initial vertex, may possibly lead to a larger clique than starting from another vertex. Thus, any more attempts will increase the chance to find a larger solution.

We also learned that, depending on the solution approach, the parallelism of the program might be more or less challenging. In our approach, since every attempt of finding a clique in a graph is independent of another, parallelization of the program is straightforward. The program is a loosely coupled parallel program in which each attempt is independently run on single core and ultimately the intermediary results are reduced to form a single final result. But for an approach like split partitioning, where the graph is divided into chunk of sub graphs, parallelism would not be easily possible and it might not scale well.

9. CONTRIBUTIONS

Dler spent a great deal of time in the kind of problems that could be parallelized. He went through a lot of research papers for the same and came up with multifarious problems that could be solved in parallel. The idea that a max clique problem could be possibly parallelized was his. Following this, he spent a great deal of time in analyzing the research papers and identifying possible methods that could be applied to our sequential and parallel design of our program thus contributing a great deal to the design as well. A good deal of the presentations and report were his work as he constantly worked

towards making them as presentable and understandable as possible. He also implemented the sequential version of the program and ensured good code quality.

V , E	N	K	T	Speedup	Eff
400 ,36990	1500000	Seq	60100		
	1500000	1	57734	1.041	1.041
	3000000	2	60640	1.982	0.991
	4500000	3	59119	3.05	1.016
	6000000	4	62173	3.867	0.967
	7500000	5	62617	4.8	0.96
	9000000	6	66122	5.453	0.909
	10500000	7	66404	6.335	0.905
	12000000	8	66695	7.209	0.901
800 ,46199	4000000	Seq	59097		
	4000000	1	50805	1.163	1.163
	8000000	2	52710	2.242	1.121
	12000000	3	53236	3.33	1.11
	16000000	4	56972	4.149	1.037
	20000000	5	56225	5.255	1.051
	24000000	6	59270	5.982	0.997
	28000000	7	59172	6.99	0.998
	32000000	8	63339	7.464	0.933
100,3109	5000000	Seq	65569		
	5000000	1	65502	1.001	1.001
	10000000	2	66280	1.978	0.989
	15000000	3	66780	2.946	0.982
	20000000	4	69091	3.796	0.95
	25000000	5	70706	4.637	0.927
	30000000	6	72523	5.427	0.904
	35000000	7	71807	6.392	0.913
	40000000	8	71592	7.327	0.916
200,7224	6000000	Seq	62745		
	6000000	1	59329	1.057	1.057
	12000000	2	61242	2.05	1.024
	18000000	3	61412	3.065	1.022
	24000000	4	66892	3.752	0.94
	30000000	5	66694	4.704	0.941
	36000000	6	68362	5.507	0.918
	42000000	7	68799	6.384	0.912
	48000000	8	69224	7.251	0.906
50,399	40000000	Seq	62601		
	40000000	1	56670	1.104	1.104
	80000000	2	54125	2.312	1.157
	120000000	3	54551	3.443	1.147
	160000000	4	57653	4.343	1.086
	200000000	5	54011	5.795	1.159
	240000000	6	55377	6.783	1.13
	280000000	7	62446	7.012	1.002
	320000000	8	62501	8.013	1.001

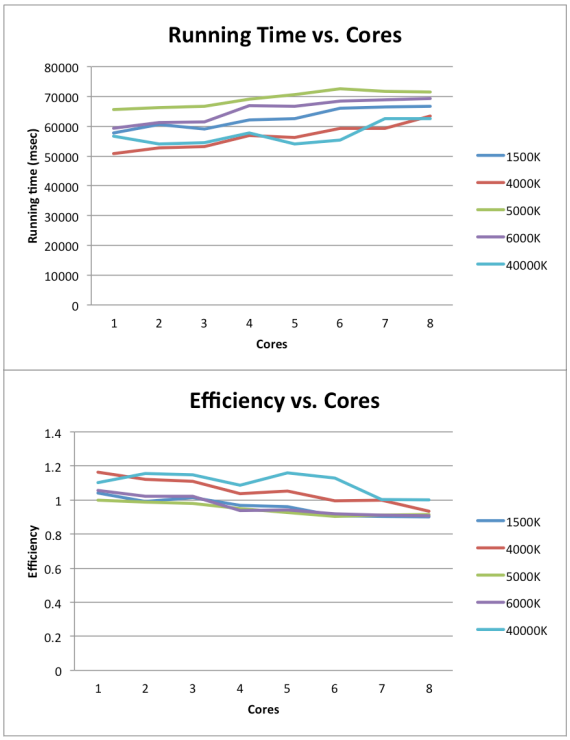


Figure 4. MaxCliqueSmp weak scaling performance metrics

Yogesh identified a couple of other problems that could be parallelized and initially came up with the idea of a parallel Sudoku solver. But then, it was identified that the max clique problem had more potential to be parallelized. He delved into the research papers as much as Dler did and nailed down the methods the strategies that could be borrowed from the papers and include them in our design. In addition to contributing to the presentations and the final report, he also implemented the parallel and the cluster versions of the program following the design which both contributed and agreed upon. He compared the running times of the sequential and the parallel versions and calculated strong and weak scaling efficiencies for five different problem sizes.

10. REFERENCES

[1] M.R. Garey and D.S Johnson, Computers and Intractability – A Guide to the Theory of NP-Completeness. New York. W.H. Freeman and Company, 1979.

[2] Kengo Katayama , Akihiro Hamamoto , Hiroyuki Narihisa, *Solving the maximum clique problem by k-opt local search*, Proceedings of the 2004 ACM symposium on Applied computing, March 14-17, 2004, Page Numbers – 1021-1025.

[3] Kengo Katayama , Masashi Sadamatsu , Hiroyuki Narihisa, *Iterated k-opt local search for the maximum clique problem*, Proceedings of the 7th European conference on Evolutionary computation in combinatorial optimization, p.84-95, April 11-13, 2007, Valencia, Spain

[4] Una Benlic, Jin-Kao Hao, *A study of adaptive perturbation strategy for iterated local search,* EvoCOP'13 Proceedings of the 13th European conference on Evolutionary Computation in Combinatorial Optimization Pages