Parallel Graph Coloring
System and Device Programming Project Report

Cito Gianni
Politecnico di Torino
s261725@studenti.polito.it

Fathollahi Hesam
Politecnico di Torino
s257696@studenti.polito.it

Niknahad Mahtab
Politecnico di Torino
s273284@studenti.polito.it

***Abstract*—In graph theory, graph coloring is a special case of graph labeling; in which particular labels, traditionally called "colors", are assigned to elements of a graph subject to certain constraints. The convention of using colors originates from coloring the countries of a map. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color. This version of the problem is called "vertex coloring". Graph coloring enjoys many practical applications as well as theoretical challenges. Many graph applications are based on vertex coloring and many graph properties are based on graph coloring. Moreover, vertex coloring is the most famous version of coloring since other coloring problems can be transformed into a vertex coloring instance.**

In this study, our goal is to analyze the performance of the famous Graph Coloring algorithms using multi-threading implemented on CPU and they are tested for the time and memory usage.

I. Introduction

A graph $G = (V, E)$ includes a set of vertices V and a set of edges E, where $E \subseteq V \times V$. Graph coloring $C : V \longrightarrow N$ is a function that assigns a color to each vertex that satisfies the property such that: $C(v) \neq C(u) \forall (u, v) \in E$.

In mathematical and computer representations, it is typical to use the first few positive or non-negative integers as the "colors". In general, one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are. The color representation can be optimized for memory space (e.g., using bit-fields in C++). The following pictures shows a few examples of graph coloring (left-hand side and center) and an example of application (right-hand side, i.e., map coloring).

The implementation starts from the Sequential Greedy algorithm then we will switch to the **One-distance**, **Luby's**, **Jones-Plassmann**, **Largest-Degree-First** and **Smallest-Degree-First** algorithms in order to make the comparison between these algorithms with respect to the time and memory usage. Finally, we will show the result in the last section.

II. Method

In this section, we are going to list how handled the different graph file formats and how we managed the graph in our algorithms.

Files are read by *read_graph* function which works as below function.

read_graph(char filename, int **xadj, int **adj,double **ewghts, int **vwghts,int* nov)*

It opens file by *fopen* method and read only permission. After file reading it's extension should be recognize because the file structures are different in *.gra* and *.graph* file and it will send to different method according to the type. In *.graph* in the first line the number of vertexes and edges are existed. We should read this files line by line with *fgets* function and insert in the buffer. Next start to read each vertex neighbours by reading buffer chunk chunk. Each row id represents the vertex number. The empty lines show vertexes without neighbor.

Vertexes will recorded in a array of integer. There also two other array for weighted vertexes and edge. Also another array for number of edges in every vertexes. In *.gra* file the story is the same with a partial difference in data structure where only the number of edges is exited in fist line and lines started with vertex number and end with that in reading line chunk by chunk these difference is considered.

Let's now see how our graph is going to be handled in our code. To do this, we chose to use a *vector of integers pointers* approach to handle the graph which means that we use the vertex number (an integer) to access and read all its neighbors (which are integer numbers as well and represent the vertices numbers to which edges are created).

In order to understand how the graph is populated in our code, let's come back to the reading phase. So, reading the file passed in input, both of *.graph* or *.gra* format, we start adding all the edges into our graph. Basically this means that every time we read a line, we know that the vertex considered is represented by the number of the line which we are reading, and then, all the numbers that are in that line are just the vertices towards which edges are created.

In terms of code, this operation means that every time we want to add an edge to our graph, the first parameter passed is the considered vertex, v, and the second parameter is the vertex towards which the edge is created, u. In practice, we are going to access to the neighbors of the vertex v and we are going to add to them the u vertex.

At the end of our reading phase, we will have a graph which will be a vector with vertices as index linked then to all the corresponding neighbors.

The approach chose in our project give us a really fast lookup for all the operations needed in the algorithms that we considered to address the problem exposed in this report. This means that we are going to be fast in look up operations where we need to get all neighbors of a specific vertex or for those operations in which we need to iterate or manipulate

over the graph rapidly.

III. Graph Coloring Algorithms

In this section, we are going to discuss about the algorithms that we use in our project.

We consider both serial and parallel algorithms in order to see where they can affect the speed of coloring a graph.

For all the parallel algorithm, we have handled this part exploiting all the possible threads available in the computer where the code is run. Our approach in all the algorithm that are parallelized was that of divide the number of vertices according to the number of thread selected in the *main* function. This basically allows the script to divide and manage more parallelized operation at the same time reducing consequently the computation time.

If from one side parallelization allows to speed up the computation, from the other side, some operations require to be synchronized, because they manipulate shared resources and need something to prevent other threads to perform actions on that. This could be overcome by means of a lock, which in our case we have handled using *mutexes*.

A. Sequential Greedy

Let's start analyzing this algorithm considering a symmetric graph *G = (V, E)* with vertex set *V*, with *|V| = n*, and edge set *E*. We say that the function $\sigma : V \rightarrow \{1...s\}$ is an *s*-coloring of *G*, if $\sigma(v) \neq \sigma(w)$ for all edges $(v, w) \in E$. The minimum possible value for *s* is known as the *chromatic number* of G, which we denote as $\chi(G)$.

The best polynomial time heuristic known can theoretically guarantee a coloring of only size $c(n/logn)\chi(G)$, where *c* is some constant.

Let's see now the pseudo-code of the algorithm.[1]

Algorithm 1 Sequential Greedy Coloring Heuristic

```
n ← |V|
Choose a random permutation p(1), ..., p(n) of numbers 1, ..., n
U ← V
for i = 1...n do in sequence
    v = p(i);
    S ← {colors of all colored neighbors of v};
    c(v) = smallest color not in S
    U ← U − {v_i}
end for
```

It is known that an optimal coloring can be obtained with a greedy heuristic if the vertices are visited in the correct order. The basic structure of the greedy heuristic is shown above in Algorithm 1.

Let's better understand the meaning of each symbol in the algorithm wrote above. *n* is the number of vertices in the *G* graph, U is the set of uncolored vertices, even though in this set is not really used, and c is the color assigned to a specific vertex.

The only aspect of the sequential heuristic that must be specified is the rule for choosing the vertex v_i. Many strategies for obtaining this vertex ordering have been proposed, for instance the *saturation degree ordering (SDO)*, suggested by Brélaz, or the *incidence degree ordering (IDO)*, introduced by Coleman and Moré, but the one used in our project is just a simple solution. We basically assigned the minimum color available between the node and its neighbors iterating over all the vertices of the graph, so starting coloring from the first one until the last one (v_i where $i = 1...n$).

B. Parallel Graph Coloring Algorithms

Parallel algorithms for graph coloring are based on the simple observation that any independent set of vertices can be colored in parallel, where an *independent set* is a set of vertices such that no two vertices are neighbors, which means that share a common edge.

Algorithm 2 Generic Parallel Graph Coloring

```
U ← V
while |U| > 0 do in parallel
    Choose an independent set I from U
    Color all vertices in I
    U ← U − I
end while
```

The generic procedure described above is used to color a graph in parallel. The strategy for coloring the vertices will depend on what is required from the coloring. If the aim is an *optimal* coloring usually the smallest available color is chosen, that is the smallest color not already being used by a neighboring vertex. If the aim is a *balanced* coloring, the least used available color might instead be chosen to balance the numb er of vertices of each color. [2]

All of the algorithms that we are used in the project are local in the graph, in the sense that only information from neighboring vertices is required.

C. Distance one

Graph is *G*=(V,E), a (1,k)-coloring *i*s a function from the vertices *V* to colors 1, 2, ...,k *t*hat all couple of vertices by distance 1 have different colors.

This algorithm for finding suitable color all vertexes should iterate over one-by-one that also will satisfy conditions distance-1 coloring. The color that should allocate for the vertex, is the smallest color that is not used by the vertex's neighbors(same as the first fit method for allocating colors). For finding the right color for vertexes(possible smallest color)the array is existed for keeping the data when all the neighbors of the vertex should be visited. After iteration over array smallest unused color (there is a flag for determining the color is already used or not) after *O*(N) *t*ime complexity (N is the number of neighbors of the vertex) could be found.

Furthermore, the algorithm is working by the greedy approach since the graph sizes could be big.

With the same approach in the sequential version, the parallel version is made that we should control and assign multi vertexes(because we work by multi threads) that is a race condition. Since it's possible to have conflict in coloring (neighbors with the same color) the colors should check-in recolored(if needed) until there not be any conflict.

D. Luby's Algorithm

Luby proposed a Monte Carlo method for constructing the independent set in parallel. The independent sets are found by assigning a *weight* to each vertex. The weights chosen by Luby were a random permutation of the integers $1, 2, ..., |U|$ multiplied by 2[3]. An independet set can be constructed in parallel by choosing all vertices whose weights are local maxima, i.e. vertices having a weight larger than any of their neighbors in the subgraph induced by U.

The parallel *Maximal Independent Set (MIS)* algorithm for graph coloring follows the basic method given in Algorithm 2, with a *maximal independent set* being constructed in parallel at each step using Luby's method. The coloring is done by giving each MIS a different color. [2]

Algorithm 3 Luby - MIS

$M \leftarrow \{\}$
$A \leftarrow V$

while $A \neq 0$ **do**
  for all $v \in A$ **do in parallel**
    chooses a value $r_v = \frac{1}{2d(v)}$ for each vertex
  end for

  $M' \leftarrow 0$
  for all $v \in A$ and $u \in N(v) \cap A$ **do in parallel**
    if $r_v > r_u$ **then**
      Add v in M'
    end if
  end for

  $M \leftarrow M \cup M'$
  $A \leftarrow A - (M' \cup N(M'))$
end while

return M

As we may note from the above graph two steps can be executed in parallel in order to speed up the computation. [4]

The input to the MIS algorithm is an undirected graph $G = (V, E)$. The output is a maximal independent set $I \subseteq V$. For all $W \subseteq V$, let $N(W) = \{w \subset V : \exists v \subset W, (v, w) \subset E\}$. Let $d(v)$ be the degree of vertex v with respect to the graph. It is easy to show that I is a *maximal independent set* in G at the termination of the algorithm.

Each execution of the body of the while loop can be implemented in $O(logn)$ time on a EREW P-RAM using $O(m)$ processors, where the expected number of random bits used is $O(n)$.

The expected number of executions of the while loop before termination of the algorithm is proven to be $O(logn)$. Thus, the expected running time of the entire algorithm on a EREW P-RAM is $O((logn)^2)$ using $O(m)$ processors, where the expected number of random bits used is $O(nlogn)$. [3]

So, basically the Maximal Independent Set (MIS) algorithm colors the graph by repeatedly finding the largest possible independent set of vertices in the graph. All vertices in the first such set are given the same color and removed from the graph. The algorithm then finds a new MIS and gives these a second color, and continues finding and coloring maximal independent sets until all vertices have been colored.

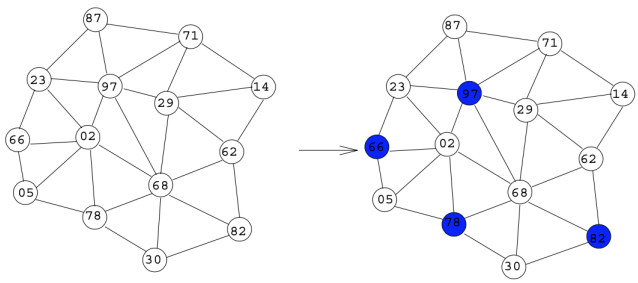
Furthermore, in order to implement correctly this kind of solution and be *thread-safe* we have also used a *mutex* in order to ensure that the addition operation of a vertex to the Maximal Independent Set M' is done in a safe e correct way.

E. Jones and Plassmann Algorithm

Jones and Plassmann recently described a parallel coloring algorithm that improves upon the parallel MIS algorithm and the upon the Luby's solution. They pointed out that it is not necessary to create a new random permutation of the vertices every time an independent set needs to be calculated. A single set of unique random weights can be constructed at the beginning and used throughout the coloring algorithm. This can easily be done by assigning random numbers to each of the vertices and using the unique vertex number to resolve a conflict in the unlikely event of neighboring vertices getting the same random number.

The Jones-Plassmann algorithm then proceeds very much like the MIS algorithm, except the fact that it does not find a maximal independent set at each step in *Algorithm 2*, since it just finds an independent set in parallel using Luby's method of choosing vertices whose weights are local maxima, and the other difference is that the vertices in the independent set of *Algorithm 2* are not assigned the same new color, as they are in the MIS algorithm. In this case, the vertices are colored individually using the smallest available color, i.e. the smallest color that has not already been assigned to a neighboring vertex. This procedure is repeated using the standard method shown in *Algorithm 2* until the entire graph is successfully colored.[2].

In order to better understand this implementation proposed by Jones and Plassmann, we can take a look the following image.



Algorithm 4 Jones-Plassmann

$A \leftarrow V$

while $|A| > 0$ **do**
  for all $v \in A$ **do in parallel**
    $I \leftarrow \{v$ such that $w(v) > w(u) \forall$ neighbors $u \in A\}$

    for all vertices $v' \in I$ **do in parallel**
      $S \leftarrow \{$colors of all neighbors of $v'\}$
      $c(v') =$ minimum color not in S
    end for
  end for

  $A \leftarrow A - I$
end while

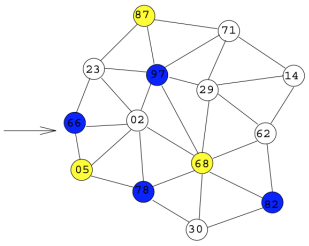


Figure 1. Jones and Plassmann example for graph coloring

As we can see from *Figure 1*, initially random numbers are assigned to each vertex. Then, each vertex looks at its neighbors and, if it has the highest number among them, it gets assigned by the lowest available color. Each uncolored vertex looks at its uncolored neighbors and gets colored with the lowest available color, only if it has the highest number, and so on until the set of uncolored vertices is empty.

Also in this case, to ensure a safe addition operation of a specific vertex in the independent set I we use a *mutex*.

F. Largest-Degree-First

The Largest-Degree-First algorithm can be parallelized using a very similar method to the Jones-Plassmann algorithm. The only difference is that instead of using random weights to create the independent sets, the weight is chosen to be the degree of the vertex in the induced subgraph. Random numbers are only used to resolve conflicts between neighboring vertices having the same degree. In this method, vertices are not colored in random order, but rather in order of decreasing degree, with those of largest degree being colored first.

This approach aims to use *fewer colors than the Jones-Plassmann algorithm*. A vertex with i colored neighbors will require at most color $i+1$. The Largest-Degree-First algorithm aims to keep the maximum value of i as small as possible throughout the computation, so that there is a better chance of using only a small number of colors.[2]

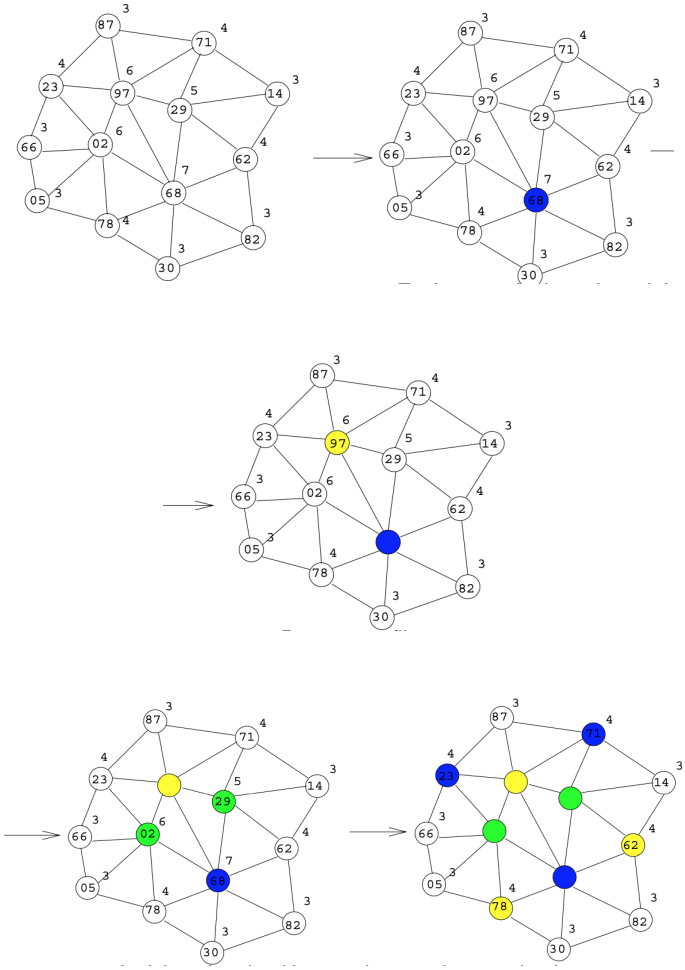


Figure 2. Largest-Degree-First example for graph coloring

As we may see from the above graphic example, we randomly assign numbers with the corresponding degree to each vertex. Then, each vertex looks at its neighbors and if it has the highest degree is then assigned a color. In case of *degree conflicts*, they are resolved by the highest random number which is assigned to that specific vertex. From this point on, the algorithm continues coloring all the remaining vertices which belongs to the uncolored set.

The following is a possible pseudo-code for the Largest-Degree-First algorithm.

Even in this case, the addition to the independent set I is made safe through the use of a *mutex*.

G. Smallest-Degree-Last

The Smallest-Degree-Last algorithm tries to improve upon the Largest-Degree-First algorithm by using a more sophisticated system of weights. In order to achieve this, the algorithm operates in two phases, a weighting phase and a coloring phase.

The weighting phase begins by finding all vertices with degree equal to the smallest degree d presently in the graph. These are assigned with the current weight and removed from the graph, thus changing the degree of their neighbors. The

Algorithm 5 Largest-Degree-First

$A \leftarrow V$

while $|A| > 0$ **do**
  for all $v \in A$ **do in parallel**
    $I \leftarrow \{v$ such that $w(v) > w(u)$ or $(w(v) == w(u)$ and $r(v) > r(u)) \ \forall$ neighbors $u \in A\}$

    for all vertices $v' \in I$ **do in parallel**
      $S \leftarrow \{$colors of all neighbors of $v'\}$
      $c(v') =$ minimum color not in S
    end for
  end for

  $A \leftarrow A - I$
end while

algorithm repeatedly removes vertices of degree d, assigning successively larger weights at each iteration. When there are no vertices of degree d left, the algorithm looks for vertices of degree $d + 1$. This continues until all vertices have been assigned a weight.[2]

Algorithm 6 Smallest-Degree-Last (Weighting phase)

$k \leftarrow 1$
$i \leftarrow 1$
$U \leftarrow V$

while $|U| > 0$ **do**
  while $\exists$ vertices $v \in U$ with $d^U(v) \leq k$ **do in parallel**
    $S \leftarrow \{$all vertices v with $d^U(v) \leq k\}$

    for all vertices $v \in S$ **do**
      $w(v) \leftarrow i$
    end for

    $U \leftarrow U - S$
    $i \leftarrow i + 1$
  end while

  $k \leftarrow k + 1$
end while

As we can see, this weighting process can be formalized by the above parallel algorithm, which assigns a weight $w(v)$ to each vertex v. Let U be the set of unweighted vertices and $d^U(v)$ be the number of vertices in U neighboring v, i.e. the degree of the vertex in the subgraph induced by U. Clearly the degree of a vertex will decrease as its neighbors are removed from consideration. Note that the following code represents only half the coloring algorithm; once the weights have been assigned, coloring proceeds as in the Jones-Plassmann and Largest-Degree-First algorithms.[2]

We can notice that what we said is described in the algorithm below, which is the same shown previously in previous chapters.

Algorithm 7 Smallest-Degree-Last

$A \leftarrow V$

while $|A| > 0$ **do**
  for all $v \in A$ **do in parallel**
    $I \leftarrow \{v$ such that $w(v) > w(u)$ or $(w(v) == w(u)$ and $r(v) > r(u)) \ \forall$ neighbors $u \in A\}$

    for all vertices $v' \in I$ **do in parallel**
      $S \leftarrow \{$colors of all neighbors of $v'\}$
      $c(v') =$ minimum color not in S
    end for
  end for

  $A \leftarrow A - I$
end while

To better understand the algorithm we may take a look to a simple example of the weighting algorithm shown here below.

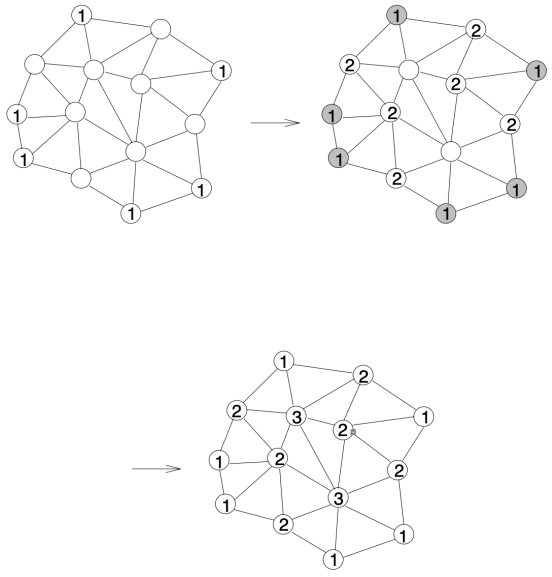
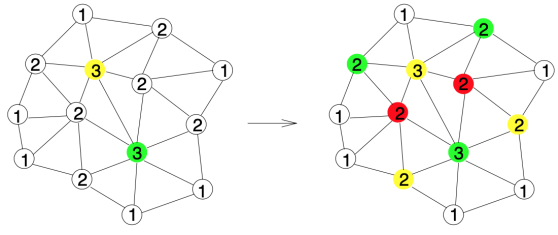


Figure 3. Weighting of the vertices during the Smallest-Degree-Last Algorithm



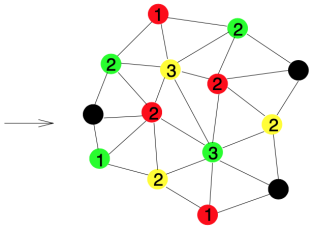


Figure 4. Coloring of the vertices during the Smallest-Degree-Last Algorithm

After the values of $w(v)$ are assigned, the coloring proceeds by starting at the highest value of $w(v)$ and working backwards. This coloring procedure works using the weights assigned by the first stage in the same way that the Largest-Degree-First algorithm uses the degree of the vertices. In other words the coloring phase has each vertex look around at its uncolored neighbors and when it discovers it has the highest weight (conflicts, once again, are resolved by a random number), it colors itself using the lowest available color in its neighborhood. [2]

From a point of view of synchronization for parallel implementation, in this algorithm we didn't use in the weighting phase a *mutex* since we just need to update a specific weight for a specific vertex. Whereas, to add then a vertex to the independent set that needs to be colored, we need to be safe in the addition and for this reason we use a *mutex* to ensure the correct operation.

IV. Results

We used three devices with the following specifications to test the performance and consuming time of the program,

OS	RAM	CPU	Device number
Mac OS	16GB 1600MHz	2,8GHz Intel i7	1
Ubuntu 64bits	6GB	2,5GHz Intel i7	2
Windows 10	8GB	2,5GHz Intel i5	3

In the tables on next pages, results are shown.

V. Conclusion

According to the performance of each algorithms, Luby has the slowest coloring time in comparison with the other parallel algorithms. Distance - one algorithm in the graphs with high number of the nodes has better performance in the parallel mode with 2 threads. For the Jones-Plassman, in most cases, it had the better performance when we use 4 threads, the performance of Largest Degree First and Smallest Degree Last are close to each other but the Largest Degree had a better result in our experiment. Finally in our experiment the Distance one has the best performance among other algorithms.

Considering instead sequential algorithms, sequential Distance - One has much way better performance compared to the sequential greedy algorithm since the difference in milliseconds it can be easily seen already from the first graph.

References

[1] A PARALLEL GRAPH COLORING HEURISTIC
Mark T. Jones, Paul E. Plassmann, 1992.
https://www.researchgate.net/publication/2768023_A_Parallel_Graph_Coloring_Heuristic

[2] A Comparison of Parallel Graph Coloring Algorithms
J. R. Allwright, Rajesh Bordawekar, P. D. Coddington, Kıvanç Dinçer, 1995.
https://www.researchgate.net/publication/2296563_A_Comparison_of_Parallel_Graph_Coloring_Algorithms

[3] A Simple Parallel Algorithm for the Maximal Independent Set Problem
M. Luby, 1986.
http://compalg.inf.elte.hu/~tony/Oktatas/Osztott-algoritmusok/mis_1986_luby.pdf

[4] Parallel maximal independent set algorithms for graphs and hypergraphs
Jessica Shi, Yiqiu Wang, Zeyuan Shang, 2018
http://web.mit.edu/jeshi/www/public/papers/parallel_MIS_survey.pdf

Table I
PERFORMANCE TEST FOR SEQUENTIAL DISTANCE ONE ALGORITHM

input	coloring time (ms)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_15_s0	2	13	25636	7660	2
rgg_n_2_16_s0	4	15	30156	12168	2
rgg_n_2_17_s0	23	16	39856	21976	2
rgg_n_2_18_s0	41	17	60324	42380	2
rgg_n_2_19_s0	72	19	103252	85322	2
rgg_n_2_20_s0	235	18	193260	175464	2
rgg_n_2_21_s0	190	21	381536	363668	2
rgg_n_2_22_s0	404	21	774000	756064	2
rgg_n_2_23_s0	926	23	1591732	1573800	2
rgg_n_2_24_s0	1655	23	3292144	3274136	2
cit-Patents.scc.gra	229	12	324856	234204	2
uniprotenc_22m.scc	8	2	54964	37896	2
uniprotenc_100m.scc	186	2	366308	349204	2
./go_uniprot.gra	193	6	496188	479000	2

Table II
PERFORMANCE TEST FOR SEQUENTIAL GREEDY ALGORITHM

input	coloring time (ms)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_15_s0	2020	13	26384	6744	1
rgg_n_2_16_s0	7737	15	17468	14140	1
rgg_n_2_17_s0	31221	16	36000	29796	1
rgg_n_2_18_s0	125690	17	74132	62528	1
rgg_n_2_19_s0	515547	19	145224	131156	1
rgg_n_2_20_s0	2071058	18	287460	274592	1

Table III
PERFORMANCE TEST FOR PARALLEL DISTANCE ONE ALGORITHM

input	number of threads	coloring time (ms)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_15_s0	2	3	13	99368	7660	2
rgg_n_2_15_s0	4	3	13	246832	7660	2
rgg_n_2_15_s0	8	6	14	541760	7660	2
rgg_n_2_16_s0	2	4	15	104024	12496	2
rgg_n_2_16_s0	4	13	15	251620	12496	2
rgg_n_2_16_s0	8	16	15	546548	12496	2
rgg_n_2_17_s0	2	17	16	113588	22120	2
rgg_n_2_17_s0	4	20	16	261512	22380	2
rgg_n_2_17_s0	8	20	16	5564440	22380	2
rgg_n_2_18_s0	2	37	17	134056	42608	2
rgg_n_2_18_s0	4	27	17	282360	43132	2
rgg_n_2_18_s0	8	26	17	577288	43396	2
rgg_n_2_19_s0	2	49	19	176984	85520	2
rgg_n_2_19_s0	4	52	19	326056	86836	2
rgg_n_2_19_s0	8	54	19	620984	87100	2
rgg_n_2_20_s0	2	203	18	266992	175640	2
rgg_n_2_20_s0	4	194	18	417600	178540	2
rgg_n_2_20_s0	8	196	18	712528	178540	2
rgg_n_2_21_s0	2	172	21	455268	363884	2
rgg_n_2_21_s0	4	231	21	608948	369952	2
rgg_n_2_21_s0	8	230	20	903876	369952	2
rgg_n_2_22_s0	2	362	21	847732	756160	2
rgg_n_2_22_s0	4	447	21	1007556	768300	2
rgg_n_2_22_s0	8	468	21	1302484	768300	2
rgg_n_2_23_s0	2	802	23	1665464	1574032	2
rgg_n_2_23_s0	4	1056	23	1837576	1598580	2
rgg_n_2_23_s0	8	1353	23	2132504	1598580	2
rgg_n_2_24_s0	2	1512	23	3365876	3274376	2
rgg_n_2_24_s0	4	2052	23	3529796	3290722	2
rgg_n_2_24_s0	8	2002	23	3824724	3290808	2
cit-Patents.scc.gra	2	455	12	324856	234204	2
cit-Patents.scc.gra	4	453	12	472320	234344	2
cit-Patents.scc.gra	8	621	12	767248	234300	2
uniprotenc_22m.scc	2	32	2	128696	38064	2
uniprotenc_22m.scc	4	33	2	276160	38080	2
uniprotenc_22m.scc	8	32	2	571088	38008	2
uniprotenc_100m.scc	2	186	2	440040	349528	2
uniprotenc_100m.scc	4	266	2	587504	349352	2
uniprotenc_100m.scc	8	244	2	882432	349480	2
./go_uniprot.gra	2	637	6	569920	479308	2
./go_uniprot.gra	4	629	6	717384	479248	2
./go_uniprot.gra	8	642	6	1012312	479232	2

Table IV
PERFORMANCE TEST FOR PARALLEL JONES-PLASSMANN ALGORITHM

input	number of threads	coloring time (ms)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_16_s0	1	173	25	12660	15765	3
rgg_n_2_16_s0	2	152	27	12656	15765	3
rgg_n_2_16_s0	4	172	27	12914	15749	3
rgg_n_2_16_s0	8	154	25	13062	15855	3
rgg_n_2_17_s0	1	400	29	36601	38952	3
rgg_n_2_17_s0	2	372	31	25464	28143	3
rgg_n_2_17_s0	4	253	28	25169	27959	3
rgg_n_2_17_s0	8	323	28	36941	39354	3
rgg_n_2_18_s0	1	747	32	83984	85143	3
rgg_n_2_18_s0	2	518	30	49868	51916	3
rgg_n_2_18_s0	4	506	30	49770	51884	3
rgg_n_2_18_s0	8	1113	30	84127	85299	3
rgg_n_2_19_s0	1	1144	32	179023	177627	3
rgg_n_2_19_s0	2	1122	33	99164	99827	3
rgg_n_2_19_s0	4	1101	32	98996	99852	3
rgg_n_2_19_s0	8	1128	31	180326	178925	3
rgg_n_2_20_s0	1	3306	36	374525	367144	3
rgg_n_2_20_s0	2	2227	34	199639	197664	3
rgg_n_2_20_s0	4	2237	36	200601	198696	3
rgg_n_2_20_s0	8	2265	33	377716	370917	3

Table V
PERFORMANCE TEST FOR PARALLEL SMALLEST-DEGREE-LAST ALGORITHM

input	number of threads	coloring time (seconds)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_17_s0	1	2.29	787	67424	69332	3
rgg_n_2_17_s0	2	1.14	323	67391	69382	3
rgg_n_2_17_s0	4	5.50	420	67665	69672	3
rgg_n_2_17_s0	8	3.39	305	68055	70049	3
rgg_n_2_18_s0	1	3.32	835	136478	136892	3
rgg_n_2_18_s0	2	2.28	1261	136613	137216	3
rgg_n_2_18_s0	4	3.30	623	137408	138018	3
rgg_n_2_18_s0	8	7.77	2112	138014	138670	3
rgg_n_2_19_s0	1	7.70	1641	277917	275595	3
rgg_n_2_19_s0	2	5.57	3416	278335	276021	3
rgg_n_2_19_s0	4	6.60	1908	278863	276774	3
rgg_n_2_19_s0	8	7.70	2254	281194	279040	3
rgg_n_2_20_s0	1	14.14	3773	566755	558891	3
rgg_n_2_20_s0	2	12.12	5408	567705	559521	3
rgg_n_2_20_s0	4	12.12	6169	568156	560414	3
rgg_n_2_20_s0	8	15.15	6706	570875	563146	3

Table VI
PERFORMANCE TEST FOR PARALLEL LARGEST DEGREE FIRST ALGORITHM

input	number of threads	coloring time (seconds)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_17_s0	1	0.93	136	45080	47673	3
rgg_n_2_17_s0	2	1.12	151	45252	47865	3
rgg_n_2_17_s0	4	0.53	107	45404	47988	3
rgg_n_2_17_s0	8	0.89	171	45170	47771	3
rgg_n_2_18_s0	1	1.14	232	90247	92114	3
rgg_n_2_18_s0	2	1.12	138	90591	92524	3
rgg_n_2_18_s0	4	1.13	204	90030	91926	3
rgg_n_2_18_s0	8	1.14	290	90210	92184	3
rgg_n_2_19_s0	1	3.33	588	183144	183652	3
rgg_n_2_19_s0	2	2.25	574	183136	183664	3
rgg_n_2_19_s0	4	3.31	771	183160	183664	3
rgg_n_2_19_s0	8	3.34	826	183455	183902	3
rgg_n_2_20_s0	1	6.62	2181	374448	372031	3
rgg_n_2_20_s0	2	4.46	1430	374427	372019	3
rgg_n_2_20_s0	4	4.48	2268	374476	372154	3
rgg_n_2_20_s0	8	8.88	1539	374951	372600	3
rgg_n_2_22_s0	1	31.31	7787	1582620	1560551	3
rgg_n_2_22_s0	2	24.24	7049	1583484	1563021	3
rgg_n_2_22_s0	4	23.23	7149	1585061	1564745	3
rgg_n_2_22_s0	8	27.27	8393	1587355	1566949	3

Table VII
PERFORMANCE TEST FOR LUBY ALGORITHM

input	number of Threads	coloring time (seconds)	number of colors	process virtual memory (Kb)	process RAM memory (Kb)	device number
rgg_n_2_16_s0	1	198	97	564207	552730	3
rgg_n_2_16_s0	2	137	97	564449	553033	3
rgg_n_2_16_s0	4	137	97	564252	552964	3
rgg_n_2_16_s0	8	203	97	564248	552861	3
rgg_n_2_17_s0	1	780	112	1287766	418394	3
rgg_n_2_17_s0	2	600	112	1287245	1033285	3
rgg_n_2_17_s0	4	720	112	1287045	1256693	3
rgg_n_2_17_s0	8	840	112	1287954	1189830	3