Massively Parallel Stochastic Local Search for the Graph Coloring Problem

Junan Zhao
Rochester Institute of Technology
jz2723@rit.edu

Prof. Alan Kaminsky (Advisor)
Rochester Institute of Technology
ark@cs.rit.edu

Abstract

The graph coloring problem is a NP combinatorial optimization problem. Given an undirected graph, it aims to find the minimum number of colors to cover the graph. There is no exact algorithm which could solve the problem in polynomial time. However, some inexact algorithms proposed in past decades are able to get approximate answers in reasonable runtime. In this paper, a new parallel heuristic search algorithm called Massively Parallel Stochastic Local Search (MPSLS) is introduced. Compared to exact algorithms, it shows great superiority in much less runtime with finding the same optimum answer. The MPSLS algorithm is also able to get approximation solutions closer to the optimum answer for large size graphs than other inexact algorithms.

1. Introduction

The graph coloring problem is a classical NP-complete computational problem in computer science industry. Given an undirected graph, G = (V, E) where V represent the vertices set and E as the set of edges of the graph, the problem is asking the minimum number of colors needed to color all vertices so that no two adjacent vertices (vertices shared an edge) have the same color. There are other variants for the problem: Given a graph, is k colors enough to complete the coloring without any clash? Or given a graph and a fixing number of colors, color the graph vertices as many as possible without any clash.

As a NP-Complete problem, there is no polynomial time solution for it. Although there are exact algorithm solutions (e.g. exhaustive search), but their runtime would grows exponentially as the graph size getting large and thus become unpractical for large graph. Here, a massively parallel algorithm will be introduced in this paper to the graph coloring problem.

Generally, a massively parallel program would take use of a large number of processors, with threads running on each processor without interacting with each other. This approach is very suitable for solving problem with large amount of computation. Compared to a sequential program running with the same algorithm, a parallel version program would usually either dramatically shorten the required runtime or apparently improve the quality of the program output answer.

On the other hand, the graph coloring problem is also a combinatorial optimization problem. A combinatorial optimization problem typically has three basic attributes: configuration, constraints, and optimization. In such a problem, the goal is to find an optimal solution with some combination of items under certain constraints. Specifically, in the graph coloring problem, the configuration is the color assigned to each vertex and the number of colors used. The constraint is no two adjacent vertices could be assigned with the same color. The optimization goal is obviously to minimize the number of colors used in the graph without any conflict.

Instead of conducting an exhaustive search, a heuristic approach was adapted. Rather than looking at every possible candidate solution, a heuristic search program looks at only a selected number of candidate solutions. The candidates are chosen by some rule, or heuristic. The heuristic attempts to select candidates that have a higher likelihood of solving the problem. Of the selected candidates, the program chooses the one that yields the best solution.

A special case of heuristic search called stochastic local search is utilized in the proposed algorithm. A stochastic local search usually starts with a random configuration, and it keeps changing the configuration in a small step every time, in order to gradually approaching the optimum solution. After tried a huge number of configurations, the best one found so far will be reported as the final solution.

In this paper, a Massively Parallel Stochastic Local Search program (MPSLS) is proposed as a solution to the graph coloring problem. With a huge amount of stochastic local searches running in parallel, the reported best solution quality could be remarkably improved.

2. Related Work

To solve the problem, the most intuitionistic and naive approach is the exhaustive search. Basically, it is to try every possible color for every vertex and see whether any configuration works. Say if there are k colors and n vertices, the algorithm would list all k^n configurations and check any of them is valid. The initial value of k could be the number of vertices. Then gradually reduce k by 1 every time and see when the program couldn't find a solution, at that time, the k + 1 is the answer. However, this approach would only works for small problem size (number of vertices and edges). As the problem size goes up, the runtime of the exhaustive search climbs exponentially.

Besides exhaustive algorithm, there is another exact approach—backtracking. The algorithm starts with k=n (k is the number of colors and n is the number of vertices). For a given order of vertices, the process will keep coloring vertices until it met a vertex without any feasible color. Then it would step backward through the colored vertices in the reverse order and identify points where different color assignments to vertices can be made. Then the forward step would resume. In this way, if a feasible solution is found, the number of colors, k, will be subtract 1 and the whole process iterate again from the root vertex. The algorithm terminates when an iteration backtrack to the root vertex, or a stop criteria reached.

The above algorithms are identified as exact algorithms, which means what they found are the real optimum answers. Another category of algorithms solving NP problem is named inexact algorithm. It's possible that inexact algorithms ended with suboptimal solutions (that's why they're called inexact), but they have the advantage of short runtime. There are various inexact algorithms proposed over decades to solve the graph coloring problem. Many of them are based on the greedy algorithm. The algorithm operates by taking vertices one by one according to some ordering (possibly arbitrary) and assigns each vertex its first available color. As mentioned, it's possible it finally got a suboptimal solution. Only with some specific orders of vertices could the algorithm find the optimum solution.

Another very similar algorithm DSATUR (Br´elaz, 1979) almost has the same behavior but with the difference of deciding which vertex to color next dynamically, compared to the order is fixed before any coloring takes place in the greedy algorithm. In DSATUR, an uncolored vertex with the maximal saturation degree, which is how many adjacent neighbor vertices it has, will be chosen to be colored next.

The iterative greedy algorithm (Culberson, Luo, 1996) is another example of inexact algorithm. It starts with using DSATUR to get an initial solution. Then, at each iteration, the current solution $S = \{S_1, ..., S_n\}$ (S_i represents a color set and n is the number of colors currently being used) is taken and its color classes are reordered to form a new permutation of the vertices. This permutation is then used with greedy to produce a new feasible solution before the process repeats indefinitely.

One more example similar to the greedy algorithm is the Recursive Largest First (RLF) algorithm (Leighton, 1979). As opposed to one vertex at a time, RLF works as coloring a graph one color a time. For each step, the process identifies an independent set (vertices whose color doesn't affect each other) and assigns a color to the set. Then the independent set is removed from the graph. This process repeats until there is no vertex left.

Some other approaches starts with complete the graph with improper coloring, which is called searching in spaces of complete, improper k-coloring. Usually, they starts with a fixed number of colors, k, and assign each vertex a color using heuristics or randomness. If the process met a vertex couldn't been colored without clash. The vertex will be colored anyway with a random color. Then strategies will be taken to reduce the number of clashed coloring until a feasible solution is found. Examples include stimulated annealing algorithm (Kirkpatrick, 1983), which is a stochastic local search algorithm in complete, improper k-coloring space, with a mechanism to go random descent with a special parameter called "temperature". The random descent mechanism helps the algorithm to avoid trapping into local optima.

There are many parallel algorithms proposed as well. In 1993, Jones and Plassmann described an algorithm which colors sequence of independent set in parallel (Jones, Plassmann, 1993). In essence, the algorithm assigns every vertex a unique number at the beginning. The algorithm does all the work in a three-level nested loop. The outer while loop checks if there are still vertices uncolored in the graph and will get into its loop body if the answer is yes. The middle loop constructs an independent set by looping through every vertex in parallel. A vertex will be added to the independent set if its unique number is larger than all of its neighbors in the current graph. Then within each independent set, the inner loop would go through every vertex in the set in parallel, collect their neighbors' information (color). The vertices in an independent set are not necessary to be colored the same. Actually, they are colored individually at their smallest available color that has not been assigned to their neighbors. At the end of each iteration in the outer loop, the vertices in the independent set will be removed from the graph, and the outer loop keeps going until there left no vertex in the remaining graph, which means all vertices got colored.

In 2000, Gebremedhin and Manne proposed a parallel algorithm called block partition based algorithm (Gebremedhin, Manne, 2000). It firstly partitions all vertices into p blocks equally, where p is the number of processors (cores) can be taken use of. Then it operates the p blocks in a parallel loop. For each block, it assigns every vertex in the block the smallest legal color they can use. Then the program backs to sequential and counts how many colors used. In the second step, it takes each color set in turn. For each class set, it partitions the set into p blocks, and re-assign colors to vertices in each block in parallel. In the third step, again, all vertices in the graph get partitioned into p blocks, and each block in parallel checks if any clash exists for vertices in the block. Any clashed vertex with a smaller color will be collected into a set A. Finally the set will be colored sequentially.

3. Program Design

There are five programs implemented, each adapted a different algorithm. The third one algorithm, which implements a stochastic local search, is the main algorithm to propose in this paper. The first and the second algorithms, the exhaustive search and the backtracking algorithm, are exact algorithms. Since the proposed algorithm is an inexact algorithm, so the exact algorithms helps with establishing tests cases with small size graphs. In this way, we can make sure the proposed algorithm has the ability to find the optimum solution. The fourth and fifth algorithms, the greedy and iterative greedy algorithms, are inexact algorithms. They serves as comparison algorithms against the proposed one, in order to evaluate how effective the proposed algorithm is.

All the programs are written in Java, along with the library, Parallel Java 2 (in short as PJ2), served as API for parallel computing. Specifications about PJ2 could be found at https://www.cs.rit.edu/~ark/pj2/doc/index.html

3.1 Exhaustive Search

The pseudocode for the exhaustive search program is shown below.

Generally speaking, the exhaustive search program do all the work in a loop, which increases the number of colors “c” gradually from 1 to V-1 (V is the number of vertices in the given graph). In an iteration with c colors, the number of total possible color configurations within c colors is $N=c^V$.

Algorithm 1 Parallel Exhaustive Search

```
1    globalSolution = null
2    for c ← from 1 to V-1
3        N = $c^V$
4        parallelFor i ← from 1 to N
5            config = generateAConfig(i,c)
6            if(noClash(config)) then
7                globalSolution = config
8                stop()
9            endif
10       endparallelFor
11   endfor
12   if(globalSolution==null) then globalSolution = worstCase() endif
```

Therefore, a parallel for loop is used to split the work. Each thread is given a serial number first, then generates a color configuration according to the number, and finally check whether it is a solution (a configuration without any clash). The serial number given determines the color configuration. Here, the number base conversion computation is used. Say we have a serial number k<=N, which is a decimal, we can convert it into a c-base number. For example, when V=6 and c is 2, which is two colors, we have totally $2^6 = 64$ configurations within 2 colors. Then we randomly pick a k, say k=56, and convert k into binary $(56)_{10} = (111000)_2$ .So 111000 determines the configuration, which is to assign the second color to the first three vertices, and to assign the first color to the last three vertices.

Once any thread found a solution, it will call a method (provided by PJ2) to stop all threads and reported the solution. In the worst case, no any solution found within V-1 colors, we have to color every vertex a unique color, and thus the number of minimum colors required is c=V.

3.2 Backtracking

The pseudocode for the backtracking program is shown below.

Usually a backtracking algorithm is hard to be parallelized. So in this program, a hybrid search structure as shown in the figure 1 below, which combines breadth-first-search and depth-first-search together, is applied. This idea is inspired by my advisor, professor Kaminsky. He adapted this hybrid search structure in solving the Hamiltonian cycle problem in his book "Big CPU, Big Data"(Kaminsky, 2019). In order to describe the searching status and search-related behaviors, a class represented the search state is created. The program starts with a queue initialized. The queue is intended to keep all the search state instances. The program gradually increases the number of colors used until a solution found. In an iteration with k-colors, the queue would first add a new search state instance initialized with k-colors. Then a loop, which executes the items in the queue, will run in parallel. So each thread takes one search state instance (the "solution" object in the pseudocode), and calls a search method (which actually do the search work). If the search procedure returns a valid solution, the program will stop immediately and the solution will be reported.

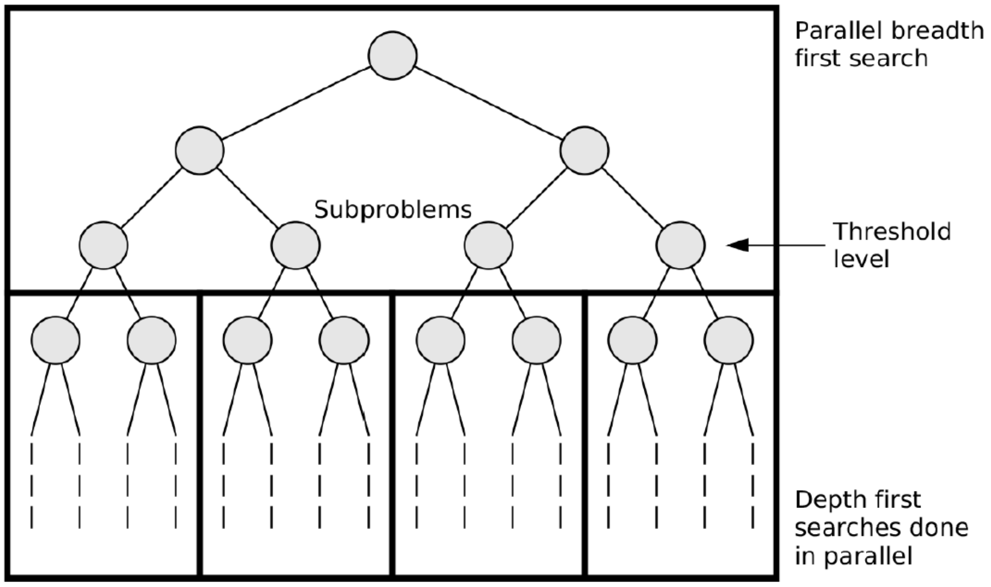
Parallel breadth first search
Subproblems
Threshold level
Depth first searches done in parallel

Figure 1 Parallel Backtracking

Algorithm 2 Parallel Backtracking (Part 1)

```
globalSolution = null
queue
for(k ← from 1 to V-1)
    queue.add(new solution(k))
    parallelFor(solution from queue)
        if(solution.seaerch()!==null) then
            stop()
            globalSolution = solution
        endif
    endparallelFor
if(globalSolution==null) then globalSolution = worstCase() endif
```

At the very beginning, an empty search state instance was put into the queue. Coming to the parallel for loop, a thread (since there is only one item--the empty instance--in the queue) takes the instance from the queue. It examines how many vertices, which is also called "level", are already colored in the instance. Then it will take different strategies according to the current level. We have a "threshold" as a static field set up in the search state class. The program allows user to configure it as an argument. If the level is smaller than the threshold, the algorithm takes breadth-first-search (BFS) as the next step; otherwise a depth-first-search (DFS) will be taken. In the BFS, the program will try all possible colors for the next level. In other words, each time the program will creates a new search instance based on the current one, with coloring the next level vertex a new color. Then all these new search instance (they have identical colors from 0 to "level", but have different colors at "level+1") will be added into the queue. On the other hand, in the DFS, the program is doing a real backtracking work. It would try a color at the next level, then do a recursion (by put its cloned instance into the queue). If at any level, the program couldn't find a proper color for the vertex, the program will go backward (go to the previous level vertex) to change a color, and then go forward again. So the process continues until it

---

Algorithm 2 Parallel Backtracking (Part 2)

```
solution.search()
{
    if(level<threshold) then bfs() else dfs() endifelse
}

solution,bfs()
{
    if(level==V) then return this endif
    for c from 0 to maxColorAtThisLevel( level, k)
          colors[level] = c
          if(comflict(colors)==0) then
              level++
              enqueuer(this)
              level—
           endif
}

solution.dfs()
{
    if(level==V) then return this endif
    for c from 0 to maxColorAtThisLevel( level, k)
          colors[level] = c
          if(conflict(colors)==0) then
              level++
              if(dfs()!=null) then return this endif
              level--  endif
     endfor
     return null
}
```

found a solution or it went backward all the way to the threshold level (which is it doesn't find a solution).

3.3 Stochastic Local Search

This is the main algorithm in this paper to propose.

The pseudocode for the stochastic local search program is shown below.

This stochastic local search (SLS, also referred as MPSLS in this paper) algorithm is inspired from the WalkSat algorithm (Kautz and Selman ,1996). The WalkSat algorithm was proposed by Kautz and B. Selman in 1996 to solve the Boolean Satisfiability problem.

Similar to the backtracking program, as a preparation, a class represented search state is created. By using a for-loop, the program gradually increases the number of colors until it found a solution. Within each iteration, the program will repeat the search N times in parallel. Each thread starts with a random color configuration, and finishes its own search in S steps. Here, N and S are arguments specified by users. Within each step, the thread first find a random vertex with conflict in the current color configuration. It is expected to replace a color for the vertex. Here are the strategies for which color to take: if there is a color which makes totally no conflict for the vertex, then assign the color to the vertex. If there are multiple such colors, pick a random one of them; else, let a constant P and a random fraction (between 0 and 1) to decide next choice. If P is larger than the random fraction, then a random color will be assigned to the vertex; else the program adapts the color which makes least conflicts for the vertex. Again, if there are more than one of such color, pick a random one of them. Finally, the thread would test the updated configuration. If there are still conflict(s), the iteration will keep going until it found a solution or S steps reached (which means the thread ends up with no solution found); else, if the configuration is a valid solution, which means no conflict for the whole graph, the thread will store the thread local solution back to the global solution. And it will also call the stop method, so that all the other threads will stop immediately. Finally, the main thread prints the solution found, or if all the number of colors within V-1 has been tried, the program will take the worst case solution—to assign each vertex a unique color.

Algorithm 3 Parallel Stochastic Local Search

```
1     globalSolution = null
2     for k ← from 1 to V-1
3         parallelFor(i ← from 1 to N)
4             threadSolution = threadLocal(globalSolution)
5             threadSolution.search(k)
6         endparallelFor
7     if(globalSolution==null) then globalSolution = worstCase() endif

//////
8     threadSolution.search(k)
9     {
10        colors[] = randomColorConfiguration()
11        for j ← from 1 to S
12            v = pickARandomVertex() with conflict(v) > 0
13            if( a color c <=k s.t conflict(v)==0) then colors[v] = c
14            else if (random(0,1)<P) then colors[v] = pickARandomColor()
15            else colors[v] = pickTheColor() with min conflict(v)
16            endifelse
17            if( conflict(colors)==0 ) then
18                globalSolution = threadSolution
19                stop()
20            endif
21        endfor
22    }
```

3.4 Greedy

The pseudocode for the greedy program is shown below.

The design of the parallel greedy algorithm is very similar to the MPSLS program. The main difference is that each thread applies greedy algorithm. In order to avoid duplication, each thread shuffles its order of vertices first. Finally, the program will do a reduction (which is literally the program compares all the threads' solutions and keeps the best one among them) and store the best solution (which has least colors) back into the global solution. The reduction is a provided function of PJ2. Developers only have to configure some methods (e.g. methods of comparing which solution is better) and then the whole reduction process will automatically complete.

Algorithm 4 Parallel Greedy

```
globalSolution = null
parallelFor(i ← from 1 to N)
    threadSolution = threadLocal(globalSolution)
    threadSolution.search()
endparallelFor

//////
threadSolution.search()
{
    shuffle(vertices)
    foreach v∈vertices
        set.add(allAdjacentColors(v))        //in each iteration, use a new set
        colors[v] = firstColorNotInSet(set)
    endforeach
}
```

3.5 Iterative Greedy

The pseudocode for the iterative greedy program is shown below.

Again, the program design structure is similar to the previous two algorithms. Each thread instead applies iterative greedy algorithm (Culberson and Luo, 1996). Each thread starts with shuffling vertices as well, then iterates S times, where S is another argument entered by users. In each iteration, it would first apply the greedy algorithm, followed the current order of vertices. Meanwhile, the program would collect information about how many number of colors used and what vertices are in each color set. Then before next iteration, a new order of vertices is formed in this way: "Putting vertices with the same color into adjacent position in the new order. The order of color sets are decided randomly." And each thread should remember the best solution it has ever found. Finally, a reduction will help to keep the best solution found.

Algorithm 5 Parallel Iterative Greedy

```
1   globalSolution = null
2   parallelFor(i ← from 1 to N)
3       threadSolution = threadLocal(globalSolution)
4       threadSolution.search(S)
5   endparallelFor

//////
7   threadSolution.search()
8   {
9      shuffle(vertices)
10     for i ← from 1 to S
11        greedy()   //same as line 10-13 of Algorithm 4
12        vertices = shuffle(colorSetList)
13     endfor
14  }
```

4 Experiment Setup

All the tests except the scaling tests were ran on Kraken. Kraken is a multicore parallel computer at the Computer Science department of Rochester Institute of Technology. It has 2 processors, each has 22 dual-hyper-threaded CPU cores, which in total has 88 threads available. However, because it is a public resource, sometimes experiments have to be executed with less than 88 threads.

The scaling tests were ran on Tardis. Tardis is a cluster parallel computer at RIT CS department. It has 10 backend computational nodes, where each node has 12 threads available. The reason for running scaling tests on Tardis is to avoid the hyper-thread core interfere (when competing the same resources, one thread will block another a little bit if there are on the same core) on Kraken.

The graph data were generated as an instance of the RandomGraph class (provided by the PJ2). Users could specify the number of vertices, the number of edges in the graph. Also, they could choose a seed (long type) which could affect the graph to be generated. With the same seed, the same graph will be generated if with the same number of vertices and edges.

All the MPSLS tests were using P=0.5.

5 Results

5.1 Comparison with Exact Search algorithms

The first set of test cases were set up to guarantee the MPSLS is able to get the correct answer for small size graphs. It was done by comparing the MPSLS program output with these two exact search programs on small and medium size graphs. Around 100 test cases had been tried, up to graphs with 65 vertices, and it turned out that the MPSLS was always able to get the right answer. Very rarely, the MPSLS got an answer slightly larger, but just by increasing the number of steps or repetitions, it was then improved to the get the optimum answer. And usually on medium and large size graphs, the MPSLS program shows great superiority than others on runtime. However, since the MPSLS is an inexact algorithm, the only disadvantage of the MPSLS algorithm is its uncertainty: without running an exact algorithm or analyzing the graph, no one can ensure what the MPSLS gets is the true optimum. Due to the limited space of this paper, here only lists a few test cases. Table 1 shows every programs runtime (milliseconds) with the correct answer (except those time marked as N/A). 86 threads on Kraken were used. And for the MPSLS program, since there are randomness in its runtime, once a least stable configuration found, which means I ran it 20 times with these configurations and got 20 optimum solutions with a smallest pair of repetition/steps, I recorded the min runtime value among them. Table 2 shows the arguments configuration that I used. “S” means the number of steps and “N” means the number of repetitions.

Graph 1 (V = 12, E = 60, Seed = 234561)
Graph 2 (V = 20, E = 65, Seed = 142587)
Graph 3 (V = 35, E = 500, Seed = 316666)
Graph 4 (V = 45, E = 400, Seed = 732469)
Graph 5 (V = 60, E = 400, Seed = 146230)

	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
Exhaustive	126831	34854	N/A	N/A	N/A
Backtracking	205	212	467164	132888	878331
MPSLS	296	329	559	653	2193

Table 1 Runtime Comparison with Exact Algorithms
(The N/A in the Table 1 means it’s too long and thus no sense to run.)

Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
N=86, S=20	N=86, S=50	N=86, S=200	N=86, S=3000	N=86, S=5000

Table 2 Arguments Configurations for the MPSLS program

5.2 Increasing Number of Steps

For the second set of test cases, I picked some random large graphs. For the same graph, with other condition unchanged, I gradually increased the number of steps. The output data are shown in Table 3 below. It shows that as the number of steps increased, the reported answer of the MPSLS program gradually decreased, which means it’s approaching or arriving at the true optimum. Here lists three cases. (N is the number of repetition)

Steps	Time	Min Num of Colors
100	13455	32
200	15823	20
500	35605	18
1000	64409	17
2000	128985	17
5000	300344	16
10000	607300	16
20000	1257179	16

Table 3 Graph = (V=250, E=5976, Seed=342587), threads=88, N=880

Steps	Time	Min Num of Colors
100	29251	50
200	29607	26
500	55933	21
1000	106611	20
2000	215826	20
5000	528602	20
10000	1071445	20

Table 4 Graph = (V=300, E=8970, Seed=342587), threads=88, N=880

Steps	Time	Min Num of Colors
100	52211	70
200	56083	39
500	113114	35
1000	251695	33
2000	428609	33
5000	1104153	33

Table 5 Graph = (V=200, E=3980, Seed=342587), threads=88, N=880

5.3 Comparison with Inexact Search algorithms

The third set of test cases were intended to compare the MPSLS with the other two implemented inexact algorithms. Since all the three programs' runtime can be controlled by users, so a natural idea is to compare their answers' quality with approximated same runtime. Here I list five of my test cases, as the charts shown below. For all these charts, the x-axis indicates the runtime, and the y-axis means their answers (the min number of colors).

The conditions of ran these test cases are listed as the caption of these charts. Here is the way that I set up the experiments: I firstly ran the MPSLS program by these steps: 100, 200, 500, 1000, 2000 and 5000. Since there are randomness factor in the program, I ran each number of steps 5 times, and picked the answer that appears most (mode) as the result, and recorded the min runtime with the mode answer. Then I ran the other two programs, the parallel greedy and parallel iterative greedy, by controlling their runtime as around the recorded runtime. Again, I picked the mode and corresponding min runtime of 5 times ran of these two programs.

Please note that the x-axis of charts (runtime) aren't scaled equally. The purpose of presenting these charts rather than table is just to make each program's trend more intuitionistic.

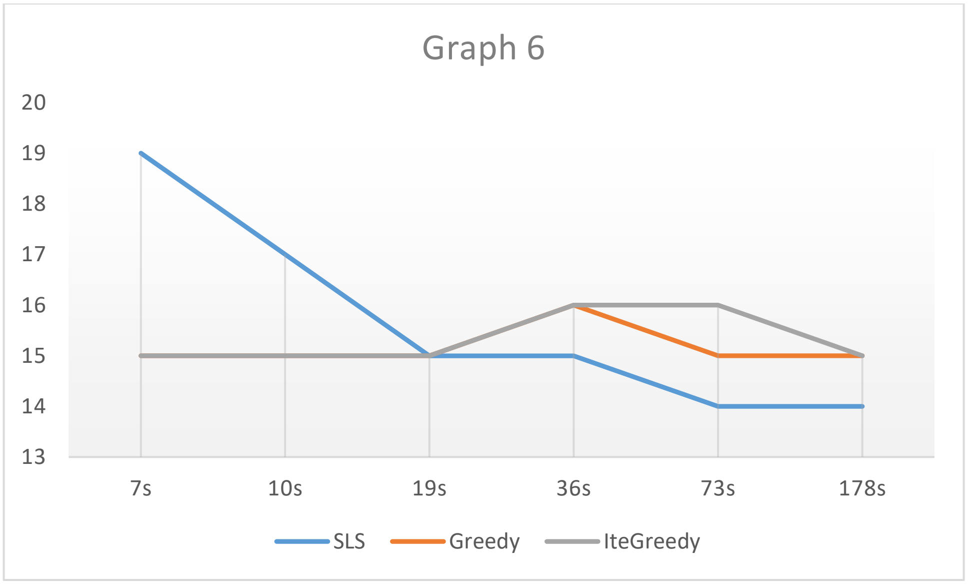

Figure 2   Graph=(V=100,E=1980, Seed=642532), threads=84, density=0.2

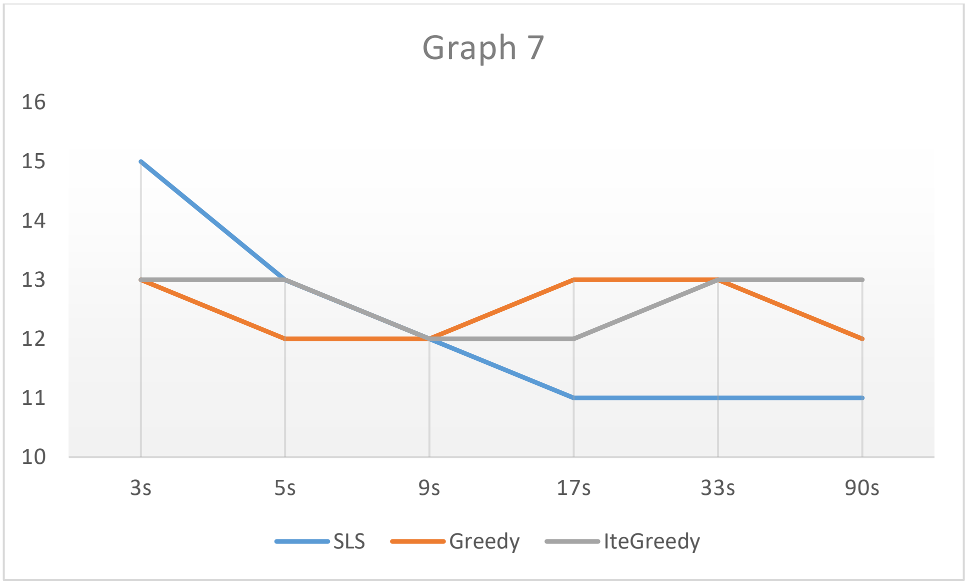

Figure 3   Graph=(V=200,E=3980, Seed=342587), threads=86, density=0.2

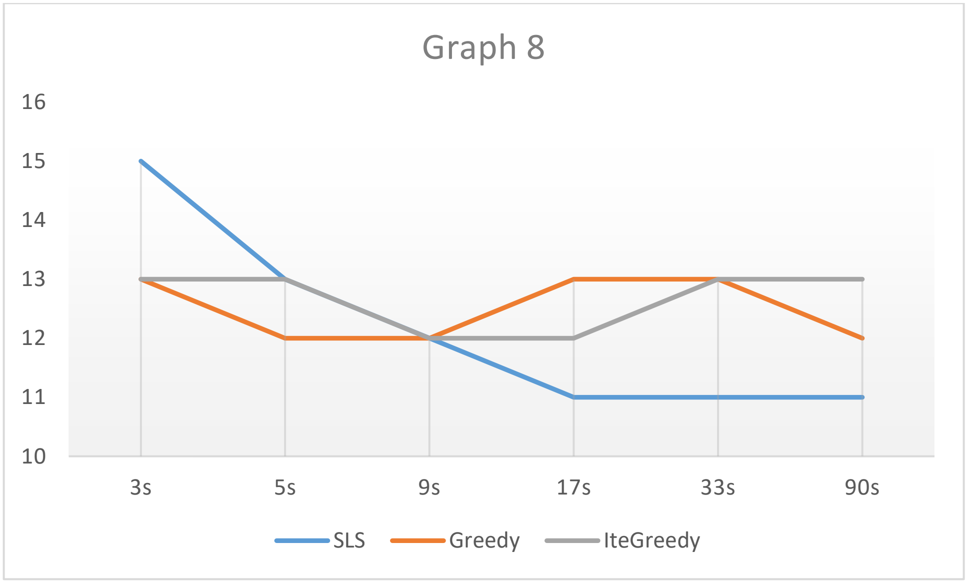

Figure 4   Graph=(V=150,E=2235, Seed=342587), threads=86, density=0.2

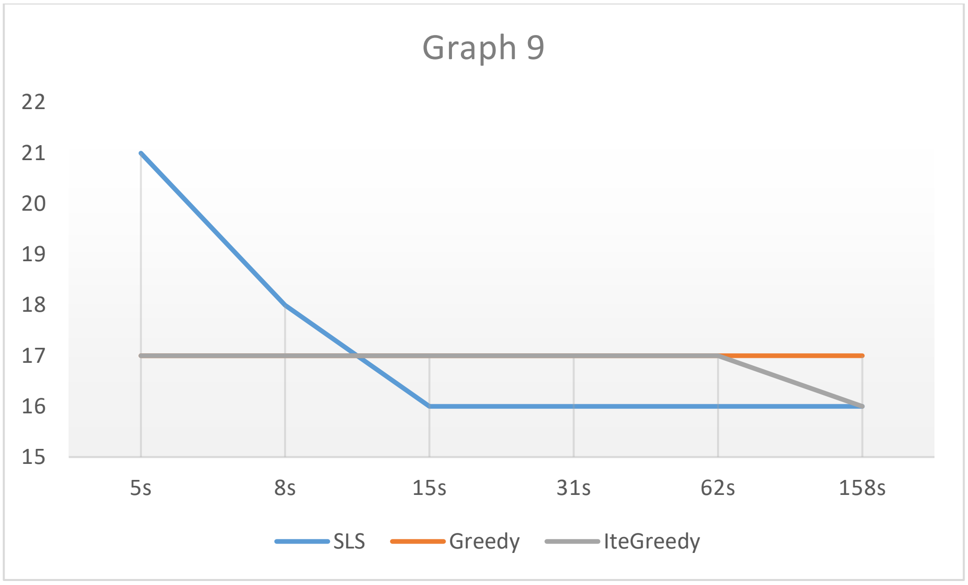

Figure 5 Graph=(V=150,E=3352, Seed=142857), threads=86, Density=0.3

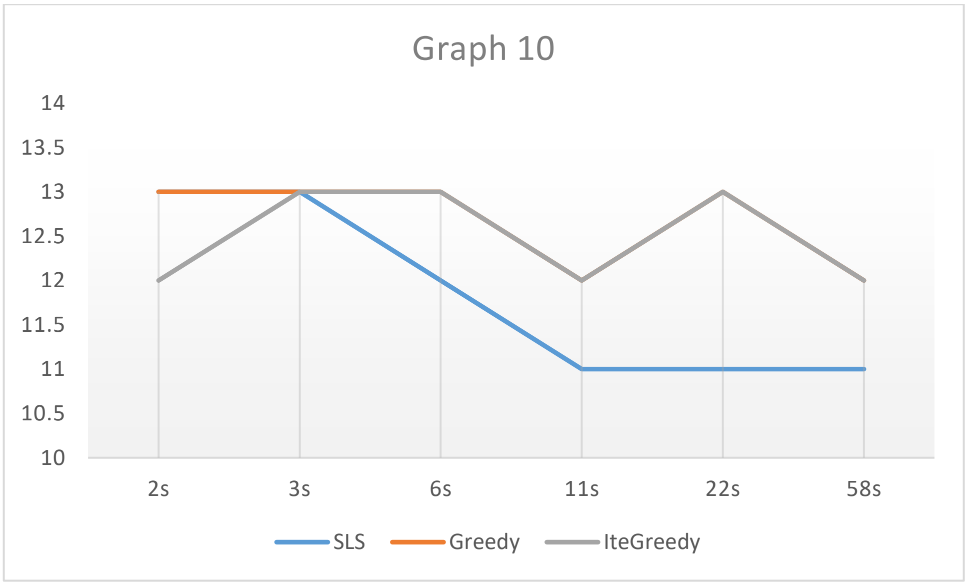

Figure 6 Graph=(V=100,E=1485, Seed=142857), threads=86, Density=0.3

These tests reveals something really interesting. So usually, the greedy and iterative greedy got fairly good answers within pretty short time e.g. 5 secs. But then their answers didn't get better as the runtime increased. Instead, their answers usually wandered around a little bit. On the other hand, for the MPSLS program, things were totally different. Within short runtime, the MPSLS often started with a far-away-from-good answer, up to 4 colors more than the greedy and iterative greedy in these test cases. However, the MPSLS has one big advantage over the other two: its answer decreased as the runtime went up. So as can be seen in the charts, the MPSLS usually had a better answer than the greedy and iterative greedy ultimately. It has the ability to dig out better solutions.

Here is the possible reason that I guess. Because the MPSLS program starts with random configuration, which has high possibility of including configurations with lots of conflicts, so within short runtime, the program is still struggling with numerous conflicts. As the runtime increases, as it’s always trying to solve conflicts, the program gradually approaching the optimum answer. For the greedy algorithm, all it does is applying greedy to different order of vertices. This is kind of “try its luck”, which on the other side means “it doesn’t dig into its answers”. Usually many orders generate the same answer (number of colors), then it can’t upgrade its answer unless a better order was met. The same is with the iterative greedy. Although it reorders its vertices based on previous work, it doesn’t face conflicts directly.

This set of experiments suggests the initialization of the MPSLS is weak. Some variants, for example initializing with k-color greedy after vertices shuffled, might upgrade the performance of the program in terms of short runtime. It might also shorten time in its searching process.

5.4 Weak Scaling Metrics

To study the MPSLS program’s performance under weak scaling, I ran the program on 1 to 12 cores of a Tardis node and measured the running times. I ran the program with steps = 2000 on five random graphs, with V = 50, 100, 150, 200, and 250 and a density of 0.2 for graphs, as listed below. Figure 7-9 plot the program’s running times, size-ups, and efficiencies. The program exhibits good weak scaling, with nearly constant running times and with efficiencies of 0.75 or better out to 12 cores.

Graph 11 (V = 50, E = 245, Seed = 142587)
Graph 12 (V = 100, E = 990, Seed = 920225)
Graph 13 (V = 150, E = 2235, Seed = 651015)
Graph 14 (V = 200, E = 3980, Seed = 661226)
Graph 15 (V = 250, E = 6225, Seed = 342587)

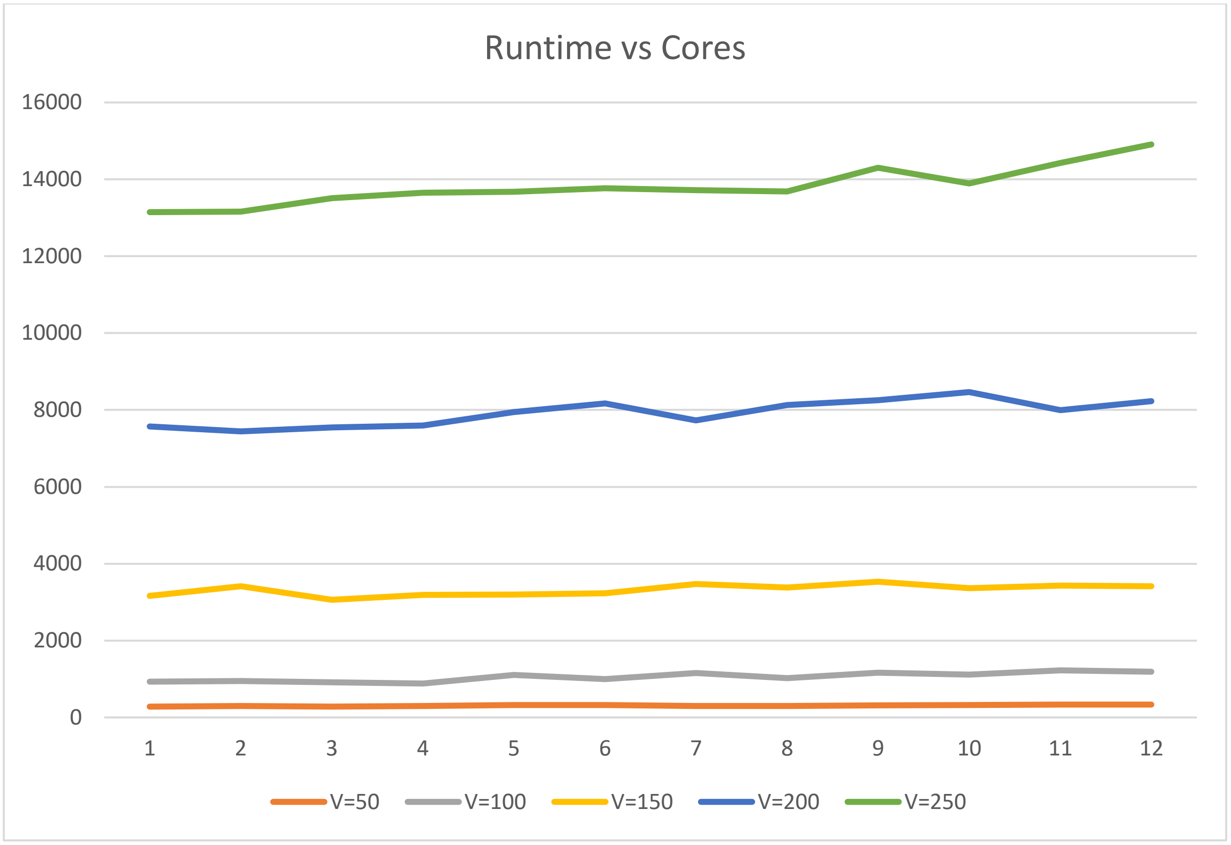

Figure 7 Running Time vs Cores (x-axis: cores, y-axis: runtime in milliseconds)

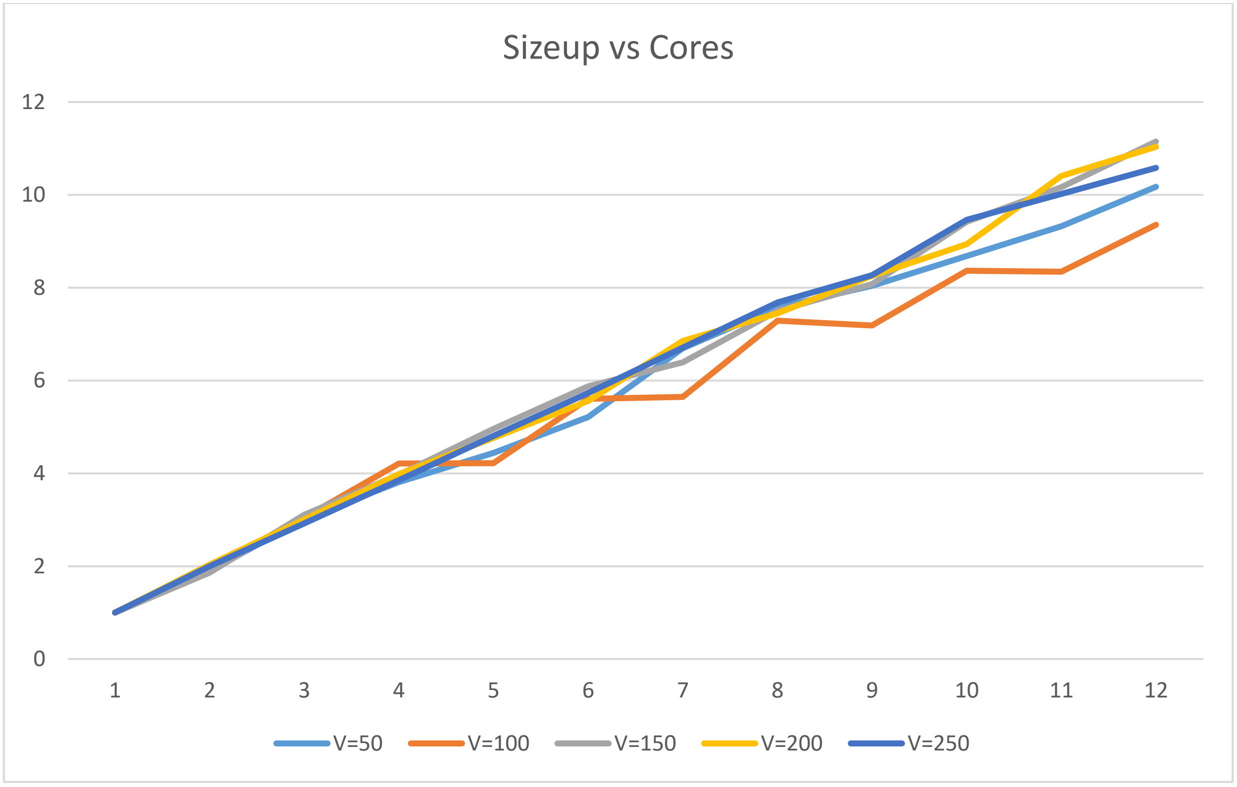

Figure 8 Size-up vs Cores (x-axis: cores, y-axis: size-up)

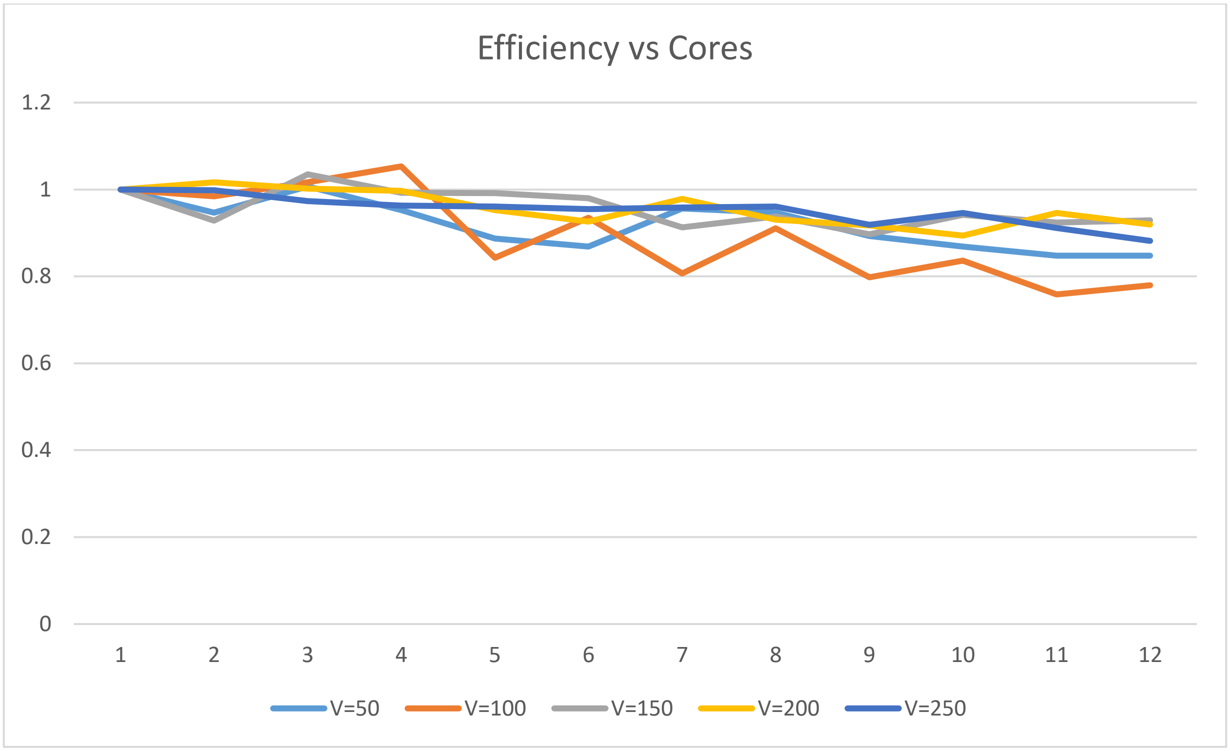

Figure 9 Efficiency vs Cores (x-axis: cores, y-axis: efficiency)

5 Future Work

First of all, as experiments indicated, some variants are worth to be applied to the MPSLS program. Currently every thread starts searching from random color configurations. Developers could apply some other algorithms, such as k-colors greedy after shuffle vertices, to build an initial state as a start point. Another issue worth notice is the parameter “P”. Due to the time issue of the project, limited number of experiments had been tried with different P values but no any clear conclusion drawn so far. Developers could continue to try more test cases with different “P”, and hopefully conclude a rule about the relationship between P’s value and search time.

Next, in order to further evaluate how effective the MPSLS program is, there should be more other recent parallel algorithms to compare with. Currently we have four other programs implemented beside the MPSLS program: the parallel exhaustive search program, the parallel backtracking program, the parallel greedy program and the parallel iterative greedy program. Among them, the exhaustive search and backtracking programs belong to exact search program. There is no much sense to compare the MPSLS program against them on large size graphs. The greedy and iterative algorithms are inexact algorithm. They are classical but pretty old, all proposed before 2000. Some good candidates to compare with are: the Hill-Climbing algorithm (Lewis, 2009); the PARTIALCOL algorithm (Bl¨ochliger, Zufferey, 2008); the ANTICOL algorithm (Thompson, Dowsland, 2008) and the Hierarchical PGAs (Abbasian, Mouhoub, 2013).

Currently, there is a drawback for all these programs: it doesn't allow users to fully specify the graph they want to analyze. Instead, the program takes use of the RandomGraph class from PJ2 to generate and represent graphs. Users could assign the number of vertices, the number of edges, and select a seed number to specify the graph to be generated, but they couldn't control the adjacent relationship within the graph. In other words, users don't know what graph exactly will be generated although they have the right to select from different graphs. So it's worthwhile to build a user input interface which allows users to fully enter the graph they want to analyze. For example, developers could enable the program to read graph information in some certain format (e.g. adjacent matrix) from files. In this way, users possess a full freedom to enter any graph.

Since there are some variants of the graph coloring problem: k-coloring problem (is it possible to cover the graph with k-colors?); graph coloring with more constraints (e.g. some vertices are only allowed to be colored with some specific colors). Developers could modify the current program as different versions to fit those variant problems.

An additional work is to build a Graphics User Interface (GUI) for the MPSLS program. This doesn't matter with the algorithm or efficiency. It's just to beatify the output and make solutions more intuitionistic and vivid.

6 Conclusion

Compared to the parallel exhaustive search and backtracking algorithms, the MPSLS program shows great superiority over them on runtime with the same optimum solution found. Also, if users want a quick not-bad answer, like within 5 seconds, they could use greedy-related algorithm. However, as indicated from experiments, if users could tolerate longer runtime, the MPSLS algorithm is a nodoubt choice. It’s able to get better quality solutions compared to the parallel greedy and parallel iterative greedy algorithms.

Based on these and other experiments, we could believe that the MPSLS algorithm is a viable approach for solving high precision graph coloring problems. The program is able to find approximate solutions that are extremely close to the exact solutions with a reasonable amount of computation. This is fine for a practical application, where a slight reduction in value from the true optimum solution is acceptable.

7 References

1. Abbasian, Reza, and Malek Mouhoub. “A Hierarchical Parallel Genetic Approach for the Graph Coloring Problem.” *Applied Intelligence*, vol. 39, no. 3, 2013, pp. 510–528., doi:10.1007/s10489-013-0429-5.
2. I. Bl¨ochliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers and Operations Research*, 35:960–975, 2008.
3. D. Br´elaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4): 251–256, 1979.
4. J. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. *American Mathematical Society: Cliques, Coloring, and Satisfiability – Second DIMACS Implementation Challenge*, 26:245–284, 1996.
5. Gebremedhin, A. H., & Manne, F. (2000). Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, *12*(12), 1131-1146
6. Kaminisky, Alan. *Big CPU, Big Data: Solving the Worlds Toughest Computational Problems with Parallel Computing*. 2nd ed., 197-216, Barnes & Noble Press, 2019.
7. Henry Kautz and B. Selman (1996). Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201.
8. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
9. F. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–506, 1979.

10. R. Lewis. A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing. *Computers and Operations Research*, 36(7):2295–2310, 2009.
11. M. T. Jones and P. E. Plassmann, A Parallel Graph Coloring Heuristic, SIAM Journal of Scientific Computing 14 (1993) 654.
12. J. Thompson and K. Dowsland. An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*, 156:313–324, 2008.

Appendix

Developer Manual

For all the five programs, with the source code provided, in order to compile and run any program, at first you must have the PJ2 library installed on machine. Where to download and how to install the library could be found here https://www.cs.rit.edu/~ark/pj2.shtml#installation. For RIT students/faculty, with permission given from the CS department, Kraken (kraken.cs.rit.edu) could be the ideal parallel computer to run these programs. Kraken is PJ2 pre-installed.

After the library installed, you have to set up the Java path to include PJ2. If the developer is using RIT CS computers, here is an example of a commands for the bash shell to set the class-path to the current directory plus the PJ2 JAR file:

```
$ export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
```

The program should be compiled with JDK 1.7 or higher. However, the PJ 2 middleware on Kraken is running with JDK 1.7. Programs compiled with later JDK versions will not successfully run on it. Here is the command to set which JDK version to use as follows for the bash shell:

```
$ export PATH=/usr/local/dcs/versions/jdk1.7.0_51/bin:$PATH
```

How to set the Java path and configure JDK on a RIT CS machine is described in full detail at https://www.cs.rit.edu/~ark/runningpj2.shtml.

In the directory where the source code are stored, use this command to compile

```
$ javac  *.java
```

After compilation, to run the program, type this command:

```
$ java pj2 MainTask ...
```

Replace *MainTask* with the class name of the program you want to run.

How to configure the arguments of running each program is described below in the “User Manual” section.

In addition, in order to watch the runtime information, you have to put a “debug=makespan” in front of the *MainTask* in these command, like this:

```
$ java pj2 debug=makespan MainTask ...
```

User Manual

Note: the <ctor> below in all these programs means “constructor”. The PJ2 library takes this argument as a constructor expression string. You are supposed to enter a GraphSpec constructor string, where GraphSpec is an interface in PJ2. In the program, the instance of the class RandomGraph, which is a class from PJ2 implemented GraphSpec, would represent the graph to be analyzed. Follow the format below to construct a RandomGraph instance:

```
edu.rit.util.RandomGraph(V,E,Seed)
```

1. Parallel Stochastic Local Search

   The program can be executed by typing this command line:

   ```
   $ java pj2 MpslsSmp <ctor> <N> <S> <P>
   ```

   - <ctor> = GraphSpec constructor expression
   - <N> = Number of repetitions for k-color search
   - <S> = Number of steps within each repetition
   - <P> = The possibility to take a random choice color

2. Parallel Exhaustive Search

   The program can be executed by typing this command line:

   ```
   $ java pj2 MpesSmp <ctor>
   ```

   - <ctor> = GraphSpec constructor expression

3. Parallel Backtracking

   The program can be executed by typing this command line:

   ```
   $ java pj2 MpbSmp <ctor> <threshold>
   ```

   - <ctor> = GraphSpec constructor expression
   - <threshold> = The level split BFS and DFS

4. Parallel Greedy

   The program can be executed by typing this command line:

```
$ java pj2 MpgSmp <ctor> <N>
```

- <ctor> = GraphSpec constructor expression
- <N> = Number of trails

5. Parallel Iteraitve Greedy

   The program can be executed by typing this command line:

```
$ java pj2 MpigSmp <ctor> <N> <S>
```

- <ctor> = GraphSpec constructor expression
- <N> = Number of repetitions for k-color search
- <S> = Number of steps within each repetition