Parallel First-Fit Graph Coloring in Java

Loutfouz Zaman

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

Abstract. We implemented an existing parallel First-Fit graph coloring algorithm using JCSP in Java and present the results of its execution on a 32 core machine. An average speedup of 12.31 was observed with 32 cores and 140 parallel CSP processes as compared to the sequential execution. The algorithm was found to perform best when the number of processes exceeded the number of cores by an order of magnitude. On a four core laptop computer a peak performance was found for a graph of 2000 nodes and 999001 edges when between 128 and 256 parallel processes were used. Some properties of the implementation were checked using the Java PathFinder model checker, which detected a bug.

1 Introduction

Vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored vertices, see e.g. Figure 1. A *k-coloring* of graph G is an assignment of integers $\{1,2,..,k\}$ to elements of G in such a way that neighbors receive different integers [1]. Graph coloring in general has a number of applications. The most notable are scheduling, register allocation [2], and the popular game Sudoku. Today, graph coloring is an active area of research.

The *chromatic number* of graph G, denoted $\chi(G)$, is the smallest k such that G has a k-coloring [1,3]. Determining the chromatic number of a graph is an **NP**-hard problem and one cannot expect to solve it efficiently for large graphs. Thus approximation is often used, which produces fast but not necessarily minimal colorings. One of such approximations is First-Fit, a greedy on-line coloring algorithm, which assigns the smallest possible integer as color to the current vertex of the graph without looking ahead or changing the colors previously assigned. This algorithm is often called inherently sequential because of its strictly ordered procedure. However, Umland [3] presented a parallel variant, which in practice yielded satisfactory speedup results on a four processor shared memory machine. In this paper, we present runtime and speedup results of our implementation of this algorithm in Java using JCSP[1]. Some properties of our implementation were checked by means of the model checker Java PathFinder[2]. JPF was able to detect a bug in our implementation and in JCSP.

The remainder of this paper is structured as follows. Section 2 discusses the importance of graph coloring in general and of First-Fit in particular. It also describes the

[1] `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`
[2] `http://babelfish.arc.nasa.gov/trac/jpf/`

history of graph coloring and the current state-of-the-art randomized algorithms. Section 3 describes a sequential First-Fit coloring algorithm and two of its parallel versions as introduced by Umland [3]. Our implementation is also briefly described. Section 5 describes our verification efforts.

2 Related work

2.1 Applications of Graph Coloring

The use of graph coloring algorithms can be found in scheduling. We refer the reader to a paper by Marx [1] who provides numerous examples. Register allocation, which is a compiler optimization where frequently used values of the compiled program are kept in the fast processor registers, is another example where graph coloring is applied. Koes and Goldstein [2] restate the traditional approach as follows. An interference graph of the program is built. If variables interfere, they cannot be assigned to the same register. Thus, if there are k registers, register allocators attempt to solve the problem of finding a k-coloring of the graph. If not all the variables can be colored with a register assignment, some variables are spilled to memory and the process is repeated. Interestingly, the authors argue that contrary to intuition, the quality of the register allocation found by a graph coloring register allocator would not be primarily dictated by the performance of the coloring algorithm. Moreover, the authors found that a heuristic coloring algorithm usually finds an optimal coloring, but an optimal coloring algorithm performs poorly as it lacks extensions to the pure graph coloring model specific to register allocation.

Sudoku can be viewed as a graph coloring problem. Sudoku is a number placement puzzle where the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9. Davis[3] expresses it as follows. Imagine that every one of the 81 squares is a vertex in a graph, and there is an edge connecting every pair of vertices whose squares are buddies. Each vertex will be connected to 20 other vertices, thus the Sudoku graph will consist of 81 × 20 / 2 = 810 edges. Finding a valid Sudoku grid amounts to finding a way to color the vertices of the graph with nine different colors such that no two adjacent vertices share the same color. Essentially, the aim of the puzzle is to perform a 9-coloring. Note that this is also a pre-coloring extension problem described by Marx [1], since a Sudoku puzzle starts with a partially completed grid (typically having a unique solution).

2.2 History of Graph Coloring

The problem of graph coloring dates back to the 19$^{\text{th}}$ century when it was tackled in the context of planar graphs. Francis Guthrie first posed the Four Color Problem[4], which states that any map in a plane can be colored using four colors or less in such a way that regions sharing a common boundary (other than a single point) do not share the same color. It remained unproven until 1977 when Appel and Haken constructed a

[3] `http://geometer.org/mathcircles/sudoku.pdf`
[4] `http://mathworld.wolfram.com/four-colortheorem.html`

computer-assisted proof, see an online article[5]. Graph coloring has been studied as an algorithmic problem since the early 1970s. Christofides [4] introduced a minimal vertex coloring algorithm using brute-force search. Another brute-force search algorithm was later introduced by Wilf [5]. Determining the chromatic number of a graph is an **NP**-hard problem and one cannot expect to solve it efficiently for large graphs [1]. That is why approximation algorithms, which guarantee performance at the expense of the quality of the produced solution can be used. Brelaz's Heuristic Algorithm [6] can be used to find a good, but not necessarily minimal edge or vertex coloring for a graph. However, the algorithm does minimally color complete k-partite graphs. Some of the modern research in graph coloring is directed towards pushing the limits of the trade-off between performance and the maximum number of colors. For example, recently Schneider and Wattenhofer [7] introduced a new technique for distributed symmetry breaking, which produces $\Delta + 1$ coloring in $O(log\,\Delta + \sqrt{log\,n})$ time and obtains an $O(\Delta + log^{1+1/log^* n} n)$ coloring in $O(log\,\Delta + \sqrt{log\,n})$ time (Δ - maximum vertex degree, n - number of vertices).

2.3 On-line coloring and First-Fit

On-line coloring refers to heuristic algorithms used to produce a proper graph coloring, which is not necessarily minimal. An on-line coloring algorithm immediately colors the vertices of a graph G taken from a list without looking ahead or changing colors already assigned [8]. On-line coloring has been widely studied in the past. Zarrabi [9] summarized the history of obtaining competitive ratios of on-line coloring over the years. Halldórsson and Szegedy [10] proved that the performance ratio of any deterministic on-line coloring algorithm is at least $O(2n/log^2 n)$ and the expected performance ratio of any randomized on-line coloring algorithm is at least $O(n/(16log^2 n))$, where n is the number of vertices.

The simplest and most intuitive of all on-line coloring algorithms is First-Fit, which works by assigning the smallest possible integer as color to the current vertex of the graph [8]. The simplicity of First-Fit has made it one of the main choices for the purpose of coloring, but the competitive analysis of First-Fist is not complete [11]. First-Fit coloring appears extensively with *interval graphs*. A graph is an interval graph if it captures the intersection relation for some set of intervals on the real line, see an online article[6]. On-line coloring of interval graphs is motivated by the scenario where resource requests arrive dynamically in an unpredictable order and the First-Fit algorithm, in particular, allocates the lowest color to the current interval that respects the constraints imposed by the intervals that have been colored [11]. Let $\chi_{FF}(G)$ denote the maximum number of colors used among the colorings of a graph G produced by First-Fit for all orderings of the vertices of G. Basically, it indicates the worst-case behavior of the algorithm. Gyárfás and Lehel [8] define *performance ratio* of First-Fit as $R_{FF} = \chi_{FF}(G)/\chi(G)$. Today, part of the research on First-Fit is directed at finding a smaller interval for R_{FF}. Recently, Pemmaraju *et al.* [12] found that $R_{FF} \leq 8$ and Smith [13] found that $R_{FF} \geq 5$. The use of First-Fit also appears in the literature outside of the

[5] `http://www.maa.org/devlin/devlin_01_05.html`
[6] `http://mathworld.wolfram.com/intervalgraph.html`

theory domain. Wan *et al.* used First-Fit scheduling as an approximation algorithm for minimum latency beaconing schedule [14].

3 The First-Fit Algorithm

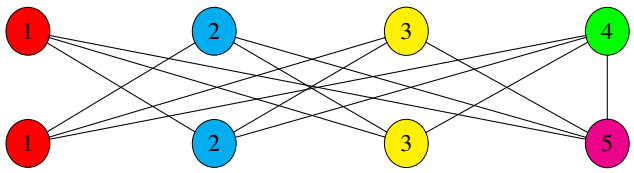
3.1 Sequential Algorithm

Umland [3] presented a two-step algorithm for sequential First-Fit coloring using the procedures below:

1. $Build(L_i, v_j)$: Determine a list of all possible colors for vertex v_i by excluding colors used by vertex v_j, $j < i$ adjacent to v_i. A Boolean possibility list L_i of vertex v_i is used with the property $L_i[k] = false \Leftrightarrow \exists v_j$ such that $j < i, (v_i, v_j) \in E, f(v_j) = k$.
2. $Color(L_i, v_i)$: Determine the smallest of all possible colors for vertex v_i. Look for the smallest entry in L_i with $L_i[k] = true$ and assign color k to v_i.

Using an example, the contents of L_i for each vertex and each vertex color after the final step of the algorithm is executed are illustrated in Figure 1.

$L_1 = \{t,t,t,t\}$ $L_3 = \{f,t,t,t\}$ $L_5 = \{f,f,t,t\}$ $L_7 = \{f,f,f,t\}$



$L_2 = \{t,t,t,t\}$ $L_4 = \{f,t,t,t\}$ $L_6 = \{f,f,t,t\}$ $L_8 = \{f,f,f,f\}$

Fig. 1: An example showing the state of the system after the final step of sequential FF is executed.

3.2 Parallel Algorithm

Figure 2 illustrates an example of how sequential First-Fit coloring described previously can be parallelized given 4 vertices and 4 processors. The actions of each column in the picture have to be executed sequentially on the corresponding processor while the actions listed in each row can take place in parallel at the specified time step of the algorithm. The control over L_i between two processors changes in the following two situations:

– between $Build(L_i, v_j)$ and $Color(L_i, v_i)$ when $i = j + 1$

– between $Build(L_i, v_{j_1})$ and $Build(L_i, v_{j_2})$ when $j_2 = j_1 + 1$

In this representation the control over the possibility lists flows through the processors in a pipelined fashion.

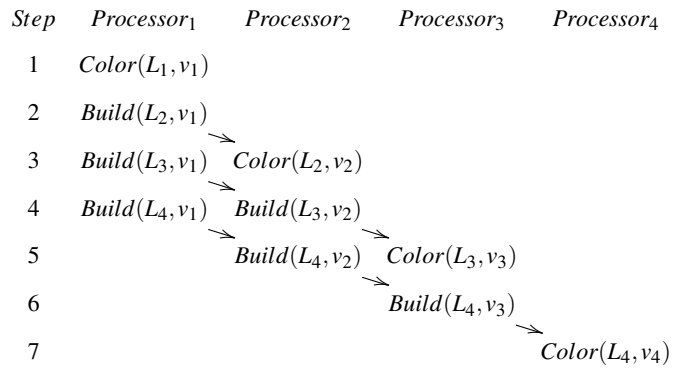


Fig. 2: Parallel First-Fit coloring with 4 vertices and 4 processors.

3.3 General Parallel Algorithm

The parallel algorithm described above has a major disadvantage since it requires as many processors as there are vertices in the graph [3]. Therefore, a generalized algorithm is necessary for any number of processors $(P_1, ..., P_N)$ and any number of vertices n in the graph such that $1 \leq N \leq n$. In this case, every processor is responsible for coloring a whole subgraph with $\lfloor n/N \rfloor$ vertices instead of coloring a single vertex as in the previous version of the algorithm. The function $Build(L_i, V_j)$ used in the general version performs the exclusion of the colors of *all* vertices $V_j = \left\{v_{1+(j-1)n/N}, ..., v_{jn/N}\right\}$ contained in the j^{th} subgraph from the possibility list L_i of vertex v_i, which will be colored later by another processor. Figure 3 illustrates an example of how the general parallel algorithm is executed with 4 processors and 8 vertices assuming that N divides n and L_1 and L_2 are already prepared for v_1 and v_2 using the previous algorithm. The flow of control over L_i for subgraphs between two different processors occurs in the same manner as described for vertices in the earlier algorithm.

3.4 Limitations

If we look at the graphical representation of the pipeline flow of this algorithm in Figure 2, we can tell that roughly 50% of the resources are unused throughout the execution of the algorithm (the area below the last active step for each processor). Even though in the generalized version of the algorithm (Figure 3) the pipeline does not empty as quickly as in the first version of the algorithm, it is unlikely that the speedup can significantly exceed half the number of processors used.

3.5 An Implementation of the General Parallel Algorithm

We refer the reader to section A in the appendix for details of our implementation. Here, we explain the procedure using an example illustrated in Figure 4. Each CSP process is responsible for coloring a subgraph and building the possibility lists, which are later used by other processes for coloring vertices and building the possibility lists of vertices of the subgraphs assigned to them. The procedure starts by obtaining the lowest and highest indices of the subgraph scheduled for coloring with the given process. The obtained indices are stored in the variables named `bottom` and `top` respectively. The scheduling distributes $\lceil N/n \rceil$ vertices for each of the first $N-1$ processes, and the remainder for the last process if $N \, mod \, n \neq 0$. Here, n is the number of processes. Next, if the current process is the first process with `identity` $= 0$, the first `top` vertices are colored using the sequential algorithm. Otherwise, they are colored using the general parallel algorithm. Each CSP process has two `One2One` channels communicating with the next and previous processes. For each vertex, the coloring operation is preceded by a `read`, which synchronizes the coloring by waiting on a token being passed from the previous process. If the current process is the last process, then the process terminates as the last process doesn't need to build any lists. Otherwise, the building phase of the process starts, which is split in two parts. The first part builds dependencies for V2 taking into account the colors used in V1 and the interdependencies. V2 is later colored by the next process. The second part excludes colors used in V1 from all the remaining vertices beyond V2. Each iteration of the subgraph is synchronized by being encapsulated within a token `read` from the previous process and a token `write` to the next process.

4 Performance Tests

We ran a performance test on MTL which featured a 32 core CPU. A random graph with a pre-defined seed was pre-generated using a random graph generator provided by the yFiles™Graph Visualization Library[7]. The graph featured 2000 vertices and 999001 edges in line with the previous work [3]. The algorithm was executed for n_{cores} in $\{1,4,8,12,16,20,24,28,32\}$, for $n_{processes}$ in $\{1,4,8,12,16,20,24,28,32,64,128,140\}$, where $n_{processes} \geq n_{cores}$, and for $n_{iterations} = 12$. The results of the first iterations were not included in the performance analysis below, but were included in the analysis of the number of obtained colors.

4.1 Number of Obtained Colors

For the 864 reported executions of the algorithm, the number of obtained colors averaged $\mu = 222.74$, $\sigma = 1.62$, *min* = 219, *max* = 227. The distinct obtained numbers of colors are explained by the fact that the algorithm was executed using different number of processes. The obtained subgraphs were different each time as the subgraph depends on the number of processes, which determines the partitioning of the graph.

[7] `www.yworks.com`

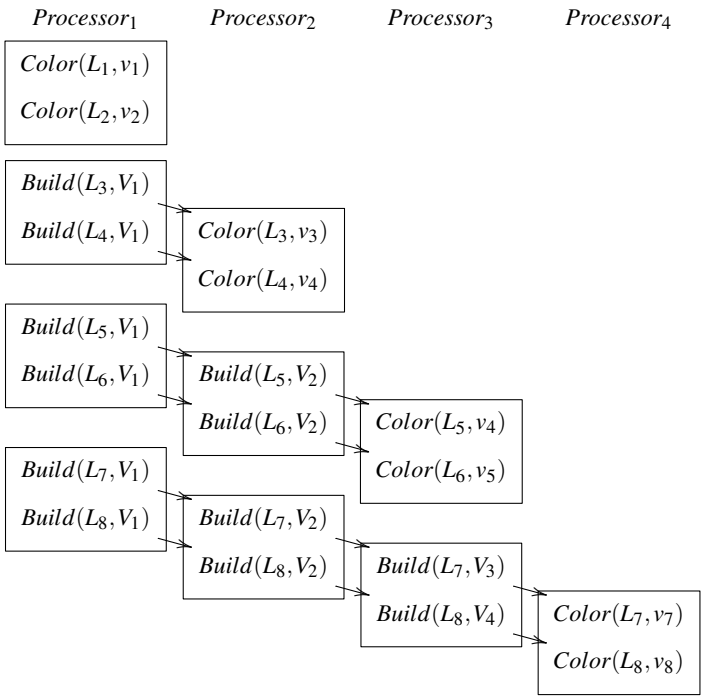


Fig. 3: General parallel First-Fit coloring with 4 processors and 8 vertices.

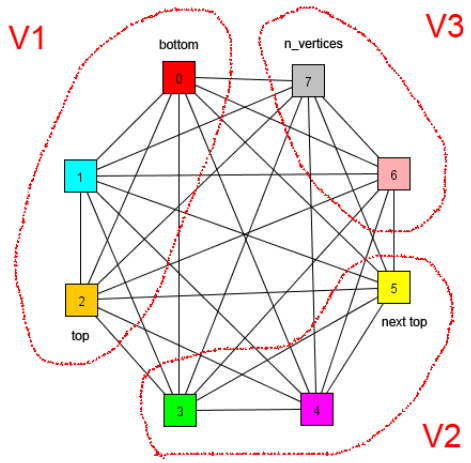


Fig. 4: An example illustrating an execution of the general parallel First-Fit graph coloring algorithm.

4.2 Runtime

Figure 5 illustrates overall runtimes for each core and each number of parallel CSP processes. Interestingly, a single core and process execution outperformed the one with four cores and processes. It can be inferred that the parallelization overhead outweighs the benefits at that level. On a single core, using more than a single process resulted in slower runtime, initially. Interestingly, as the number of processes increased, the runtime decreased and matched that of the purely sequential execution. This can be partially explained by the decreasing impact of the sequential algorithm. The exact reason is unclear. However, using more parallel processes may provide further hints.

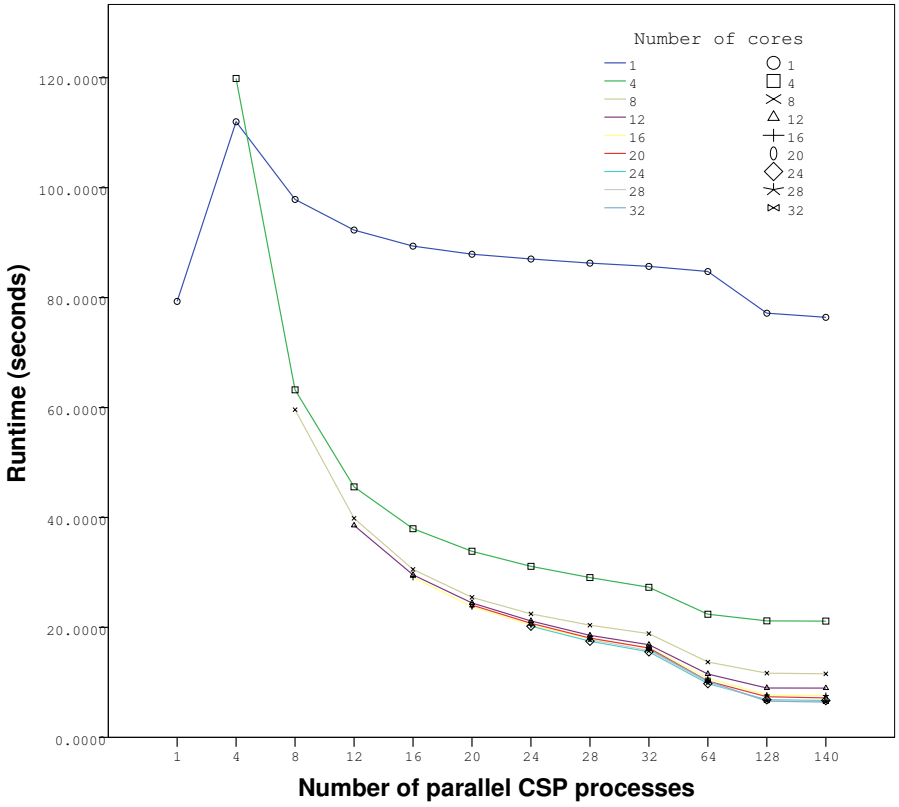


Fig. 5: General parallel FF runtime on MTL.

4.3 Speedup

We define the speedup for a given number of cores and processes as the average runtime for a single core and a process divided by the average runtime for the given number of

cores and processes, see below:

$$Speedup_{n_{cores},n_{processes}} = \frac{\mu_{Runtime_{1,1}}}{\mu_{Runtime_{n_{cores},n_{processes}}}}$$

A more compact scale can be obtained by visualizing the results using this method where the performance differences appear more prominent. Figure 6 illustrates speedup for each combination of core and process. It can be inferred from the figure that the number of cores is clearly a factor affecting the speedup of the algorithm and it is most prominent for $n_{processes} = 140$ where an average speedup of 12.31 was observed with 32 cores. Here, the highest speedup can also be observed when fewer cores were used. A detailed look at the speedup for each number of concurrent CSP processes starting from 16 is presented in Figure 7. Interestingly at 20-64 processes, the performance cannot be observed as continuously improving with the addition of cores. However, at 128 and 140 processes, a steady improvement can be clearly seen. It can be inferred that the algorithm performs best when $n_{processes}$ exceeds n_{cores} by an order of magnitude.

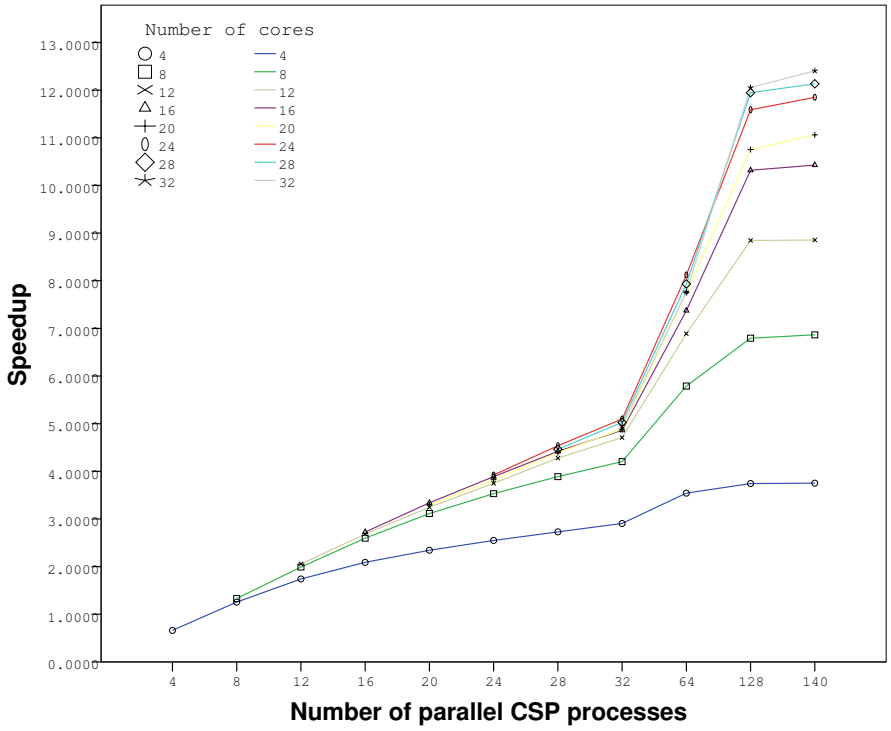


Fig. 6: Speedup on MTL.

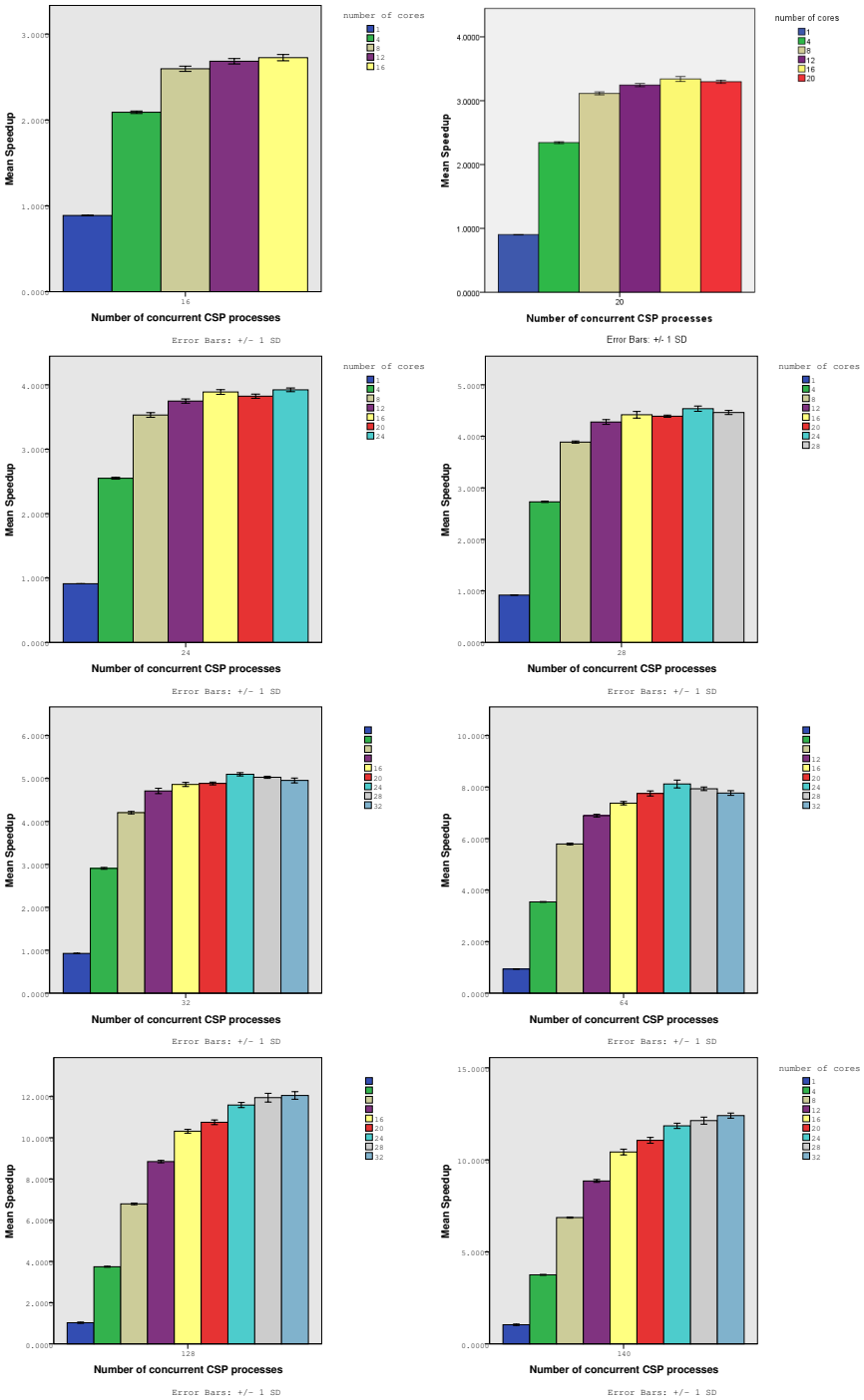


Fig. 7: Speedup for each number of concurrent CSP processes starting from 16

4.4 Peak Performance

By looking at Figure 6 it is apparent that the peak performance for this algorithm was not achieved at 140 processes. Executing more than 140 concurrent processes on MTL could potentially affect the stability of the system negatively, and, as a result, the peak performance test was not conducted. Instead, for this purpose a high end laptop with Intel®Core i7 720QM @ 1.60 GHz CPU, 6 Gb of DDR3 RAM and running Microsoft®Windows™7 Pro 64-bit SP1 was used. The CPU features 4 cores with hyperthreading. The test was performed for $n_{processes}$ in $\{2^2, 2^3, .., 2^{10}, 1\frac{1}{2} \cdot 2^{10}, 2^{11}\}$ for five iterations. A graph with the same properties was used as described in Section 4. This peak performance test was conducted twice: the second time after a bug in the implementation was fixed (see Section 5.1). The results of the first four iterations when the test was performed for the second time are summarized in Figure 8. Both times, 224 colors were obtained and the results did not appear to be drastically different from the earlier results and a slight improvement was observed. Both times, the benefits of parallelization appeared to diminish somewhere between 128 and 256 processes mark.

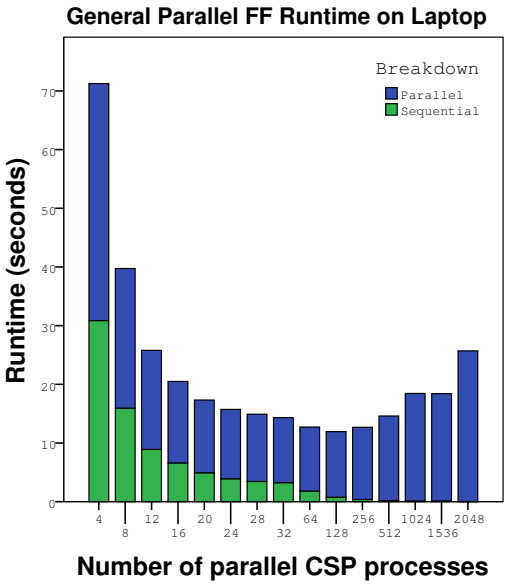


Fig. 8: Runtime of general parallel FF on laptop.

Figure 6 illustrates dramatic increases in speedup as the number of processes increases. Part of the speedup is due to the decreasing impact of the sequential algorithm used to color the initial vertices as the number of concurrent processes increases. This is illustrated in Figure 8 with a breakdown of the runtime the algorithm spends executing

sequential and parallel procedures. As the number of concurrent processes increases, the time the algorithm spends executing sequential procedures decreases. Somewhere between 128 and 256 processes the peak performance is reached. The parallelization overhead starts to increase resulting in the overall performance decrease. This is also demonstrated in Figure 8.

5 Model Checking with the Java PathFinder

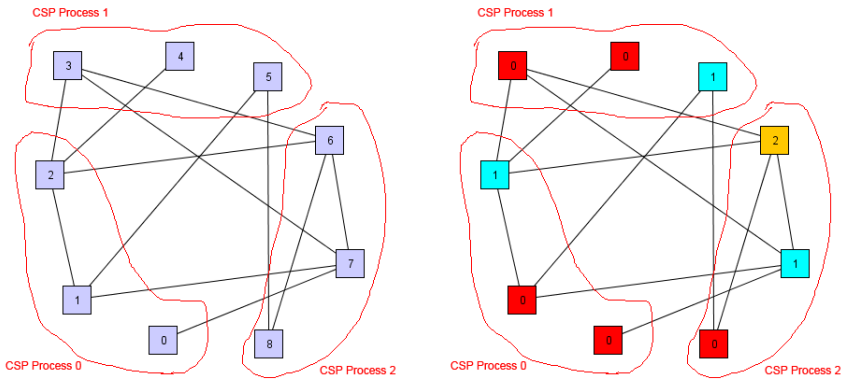
We used Java PathFinder to perform model checking. Two bugs were detected: one in the implementation and the other in the JCSP library.

5.1 Bug in the Implementation

JPF was able to detect a `java.lang.NullPointerException` in the line that was accessing the list of assigned colors in the first iteration of the building part of the algorithm. The bug was initially detected by running the algorithm with a randomly generated graph of 10 nodes and 20 edges through JPF with four parallel CSP processes using the `DFSearch` algorithm. We then encapsulated the problematic Java statement within `try` and `catch (java.lang.NullPointerException e)` blocks and made the catch block print a message describing variables used locally as follows: process id, possibility list, excluded node, mapping of colors to nodes, and the `bottom`, `top` and `next_top` values. However, at 10 nodes and 20 edges three statements were produced, so we reduced the graph size to 9 nodes and 12 edges. At this threshold exactly one statement was printed, which made it easier to analyze the problem. A layout of the targeted graph and a produced proper coloring are shown in Figure 9. Note that the last of the four processes was not assigned any nodes for coloring. For this scenario, the possibility list for node 7 could not be built because the color for node 6 was not available yet. The printout of assigned colors indicated that exactly the first five nodes were colored. This suggested that the process was simply "ahead" and attempted to build a possibility list using a node, which did not have a color assigned yet. As it turned out, synchronization using token passing was not enough in the first iteration of the algorithm as there may be interdependencies between the nodes in the current subgraph, which need to be taken care of with additional synchronization. The issue was easily fixed by encapsulating the entire statement within a `while` loop, which would make the process wait for the color to become available before proceeding further. See pseudo Java code below:

```
int color = -1;
while (color == -1)
{
 try
 {
  color = getColor(i);
 }
 catch (java.lang.NullPointerException e)
```

```
 {
  System.err.println("Waiting for color from node " + i
    + "to build" + "L["+j+"]");
 }
}
build(color, j);
```



(a) Minimal graph producing the bug. (b) The produced proper coloring.

Fig. 9: The minimal graph used in detection and correcting of the bug. The last of the four processes was not assigned any nodes for coloring.

5.2 Correctness Tests

Two tests for correctness were performed. The first one was performed on small graphs prior to the full-scale evaluation on the MTL machine. The goal of the test was to make sure that the graph was properly partitioned among the available processes. The correctness of graph partitioning was challenged in this test. Random small graphs with certain distinct properties, such as a star arrangement among others, were created using a graph visualization and editing tool implemented for this purpose. The correctness of the partitioning method was tested for $n_{processes}$ in $\{1,2,..,N+1\}$, where N is the number of vertices in the graph. The second test was included in the JPF tests and featured an execution of a polynomial time algorithm at the end of each trial on a laptop. The test checked if a proper coloring was produced. The source code is presented in Section B.1 of the appendix. No errors were observed for all search methods that successfully terminated.

JPF Tests Initially, running the test on the graph with 2000 nodes and 999001 edges (as done previously on a regular JVM) made JPF cause a stack overflow error. The same

was observed for a graph with 10 times as fewer nodes and edges. However, JPF was able to handle a 20 node and 190 edge graph for up to 32 parallel CSP processes using the `DFSearch`. Beyond 32 CSP processes the test was not conducted. In reality, only up to $n+1$ CSP processes are required in testing the algorithm for a graph of size n, as the remaining processes will not be doing any work and only one of such processes needs to be checked. The results of the tests are summarized in Table 1b of the appendix. Another group of tests were conducted with the graph in Figure 9 for some search strategies. The results are summarized in Table 1a of the appendix. No data races were found with the search strategies that terminated. This finding supports the fact that data races are not supposed to occur in CSP applications, where low level synchronization constructs are abstracted from the programmer.

Bug in JCSP A suspicious error `gov.nasa.jpf.jvm.NotDeadlockedProperty` was thrown by JPF pointing to the lines originating from the JCSP library. The error would be thrown even for a unit graph of a single node. Exactly the same error was thrown for a much simpler JCSP application of a similar kind. The application used a single `One2One` channel to transfer integers from 0 to 10 between two processes. Because of the simplicity of this application, it is unlikely the deadlock is due to the higher level code of the application and likely can be explained by a bug in JCSP.

6 Conclusion & Future Work

In this paper, we discussed why graph coloring in general and First-Fit coloring in particular are important problems. We described an implementation of a parallel First-Fit graph coloring algorithm using the CSP approach, which was previously proposed by Umland [3]. We presented the results of its execution on a 32 core machine and confirmed that the number of cores was indeed a factor that determined the performance of the algorithm. We also found that the algorithm performed best when $n_{processes}$ exceeded n_{cores} by an order of magnitude. A peak performance was found for a graph of 2000 nodes and 999001 edges on a four core laptop. Some properties of the implementation were checked using the JPF model checker, which was able to detect a bug in the implementation. Another bug was found in the JCSP library. Arguably, it would be hard to find the bugs without using this tool.

There is another correctness test that can be done to verify the earlier mentioned property $5 \leq R_{FF} \leq 8$. This, however, requires an implementation of a minimal coloring algorithm to obtain $\chi(G)$. Also, in the future, modeling can be applied for predicting optimal $n_{processes}$ given n_{cores}, $n_{vertices}$, and n_{edges}.

Acknowledgements

Thanks to the management, staff, and facilities of the Intel®Manycore Testing Lab[8].

[8] `www.intel.com/software/manycoretestinglab`

References

1. Marx, D.: Graph coloring problems and their applications in scheduling. Periodica Polytechnica Ser. El. Eng. **48**(1-2) (2004) 5–10

2. Koes, D., Goldstein, S.C.: An analysis of graph coloring register allocation. Technical Report CMU-CS-06-111, Carnegie Mellon University (March 2006)

3. Umland, T.: Parallel graph coloring using JAVA. In: Architectures, Languages and Patterns for Parallel and Distributed Applications, IOS Press (1998) 211–218

4. Christofides, N.: An algorithm for the chromatic number of a graph. The Computer Journal **14**(1) (1971) 38–39

5. Wilf, H.S.: Backtrack: An O(1) expected time algorithm for the graph coloring problem. Information Processing Letters **18**(3) (1984) 119–121

6. Brélaz, D.: New methods to color the vertices of a graph. Communications of the ACM **22**(4) (April 1979) 251–256

7. Schneider, J., Wattenhofer, R.: A new technique for distributed symmetry breaking. In: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM (2010) 257–266

8. Gyárfás, A., Lehel, J.: On-line and first fit colorings of graphs. Journal of Graph Theory **12**(6) (1988) 217–227

9. Zarrabi-Zadeh, H.: Online coloring co-interval graphs. Scientia Iranica **12**(6) (2009) 1–7

10. Halldórsson, M.M., Szegedy, M.: Lower bounds for on-line graph coloring. Theoretical Computer Science **130** (1994) 163–174

11. Narayanaswamy, N., Babu, R.: A note on first-fit coloring of interval graphs. Order **25** (2008) 49–53

12. Pemmaraju, S.V., Raman, R., Varadarajan, K.: Buffer minimization using max-coloring. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2004) 562–571

13. Smith, D.A.: The First-Fit Algorithm Uses Many Colors on Some Interval Graphs. PhD thesis, Arizona State University, United States (2010)

14. Wan, P.J., Wang, Z., Du, H., Huang, S.C.H., Wan, Z.: First-fit scheduling for beaconing in multihop wireless networks. In: Proceedings of the 29th conference on Information communications, Piscataway, NJ, USA, IEEE Press (2010) 2205–2212

A Details of the Implemention

The class diagram of the system is shown in Figure 10. `FF` is an abstract class which contains the implementation of `build` and `color` methods. `SharedFFData` is the class which contains the shared data such as the boolean possibility list `L`, the hashtable which stores the assignment of colors for each vertex and the shared graph. The two classes `SequentialFirstFit` and `ParallelFirstFit` extend `FF`. These two classes implement the corresponding algorithms. Each of them also implements the abstract method `obtain_coloring` inherited from the parent class. The method is responsible for performing the actual coloring operation. The parallel implementation of the method creates a parallel CSP process and runs it as follows. It creates a `One2OneChannel` between the current process and the previous process and another such channel between the current process and the next process. The first process doesn't have a pointer to the previous channel and the last process doesn't have a pointer to the next channel.

There are two more classes, `FFProcess` and the extending `FFGeneralProcess` declared inside the class `ParallelFirstFit`, see Figure 10. `FFProcess` extends `CSProcess` from the JCSP library. The two classes implement the corresponding parallel (vertex-based) and general parallel (subgraph-based) First-Fit coloring algorithms. Every instance of `FFGeneralProcess` (and `FFProcess`) has an `identity` of type `Integer` associated with it, which is used in the `run` method for controlling the logic and partitioning the graph. The partitioning is obtained using the methods `get_bottom` and `get_top`.

B Correctness Tests and Model Checking

B.1 The Correctness Test for Proper Coloring

```
int size = ff.sffd.nodeArray.length;
for (int k = 0; k < size; k++)
{
 Node node = ff.sffd.nodeArray[k];
 int node_color = ff.sffd.hashtable_coloring.get(k);
 for (NodeCursor nc = node.neighbors();  nc.ok(); nc.next())
 {
  Node neighbour = nc.node();
  int neighbour_index = neighbour.index();
  int neighbour_color =
   ff.sffd.hashtable_coloring.get(neighbour_index);
  //if fails then improper coloring is produced
  assert(neighbour_color != node_color);
 }
}
```

B.2 Detailed Results of the JPF Tests

`RandomSearch` and `RandomHeuristic` resulted in a `[SEVERE] JPF out of memory` warning message and did not finish. The premature termination explains the fact that

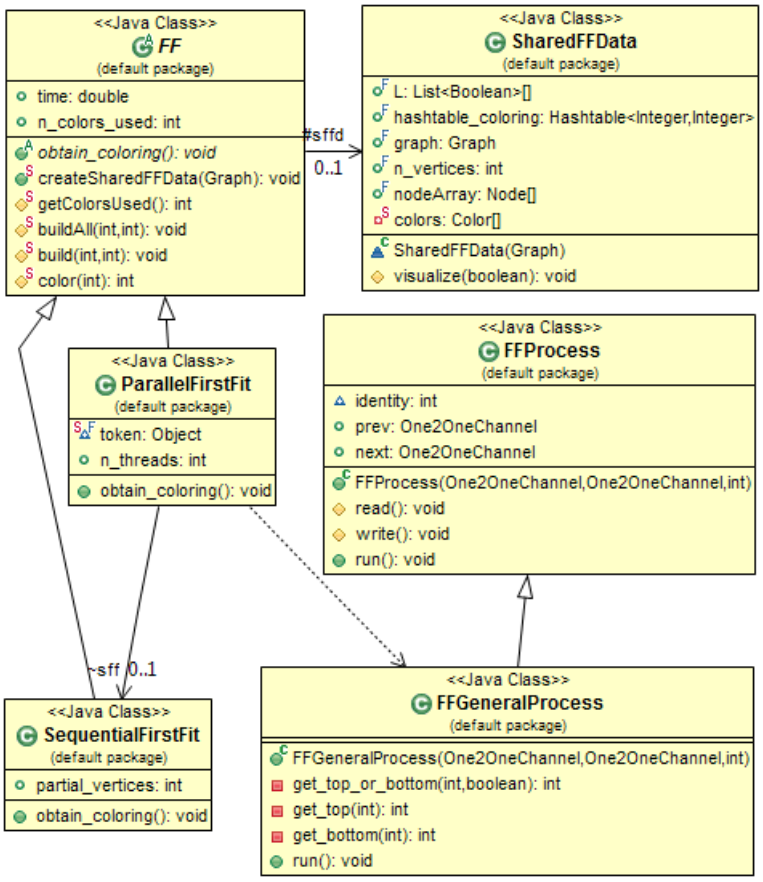


Fig. 10: The class diagram of the implementation

a bug in the JCSP was not reported for these two search strategies. Encapsulating the sequential part of the algorithm within the `Verify.beginAtomic()` and `Verify.endAtomic()` sped up the search process but did not solve the issue. Both search strategies used up around 1362 MB of memory before terminating.

	DFSearch	DFSHeuristic	RandomSearch	MostBlocked	RandomHeuristic
Time:	0:00:04	0:00:07	0:04:25	0:00:47	0:36:15
New states:	1689	4388	1255	24749	221706
Visited states:	3	4	230934	24151	225642
Backtracked states:	10	4391	0	48899	447347
End states:	8	8	0	1	0
maxDepth:	1681	1680	0	370	192
Threads:	1682	1681	232190	24229	119218
Heap new:	1231	1244	462033	51941	1053611
Heap free:	295	295	462033	19343	137926
Max memory:	91MB	182MB	1362MB	295MB	1361MB
JCSP bug found:	Yes	Yes	No	Yes	No

(a) 9 nodes, 12 edges, 4 CSP processes

Nodes	20	20	40
Edges	190	190	400
CSP processes	20	32	20
Time:	0:00:47	0:01:41	0:04:23
New states:	24802	44777	98266
Visited states:	0	33	30
Backtracked states:	7	40	37
End states:	8	8	8
maxDepth:	24794	44769	98258
Choice generators thread#:	24795	44770	98259
Heap new:	2623	6545	4313
Heap free:	1191	4851	2347
Max memory:	453MB	994MB	1361MB
JCSP bug found:	Yes	Yes	Yes

(b) Other graphs, all using `DFSearch`

Table 1: Results of JPF tests