DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **00**, 0000

Experiments with Parallel Graph Coloring Heuristics and Applications of Graph Coloring

GARY LEWANDOWSKI AND ANNE CONDON

November 1994

ABSTRACT. We introduce a new hybrid graph coloring algorithm, which combines a parallel version of Morgenstern's S-Impasse algorithm [**26**], with exhaustive search. Our goal is progress towards a coloring heuristic that works well without extensive tuning of algorithm parameters. We also contribute new test data arising in five different application domains, including register allocation and course scheduling. Hybrid was implemented on a Connection Machine CM-5, and tested on the application data as well as several types of randomly generated graphs. The results are compared with results of two simple sequential heuristics, the Saturation algorithm of Brélaz [**5**] and the Recursive Largest First (RLF) algorithm of Leighton [**24**], as well as with previous work reported by Morgenstern [**26**] and Johnson et al. [**17**].

On many random graphs, the performance of Hybrid without tuning of parameters is comparable or better than tuned sequential algorithms; on large random graphs, Hybrid does not come close to the best colorings found by tuned time-intensive algorithms such as XRLF [**17**] and Morgenstern's tuned S-Impasse [**26**].

Of the application data, three applications are easily colored even by the simple sequential heuristics; one (the course scheduling data) is optimally colored by Hybrid but not by the simple heuristics, and one appears to be very hard. In several cases, however, we found that finding an optimal coloring is not sufficient to solve the problem at hand, rather colorings satisfying additional restrictions are needed. We find that the course scheduling applications is not well-modeled by random graphs, which suggests that more application data should be collected for testing heuristics and that new random generators are needed to model these problems.

1991 *Mathematics Subject Classification.* 90C35; Secondary 68U30,05C15, 05C90, 68R05,68R10.

Key words and phrases. parallel, graph coloring, scheduling.

Research by both authors was supported by NSF grant number CCR-9257241, and by a grant from the AT&T Foundation.

Research by the first author was also supported by NSF grant DCR-9208639

Research on the Connection Machine CM-5 was supported by NSF Institutional Infrastructure grant CDA-9024618 and by the Thinking Machines Corporation.

1. Introduction

A classical problem in graph theory, the graph coloring problem is to color the nodes of an undirected graph with as few colors as possible, such that no adjacent nodes share a color. The earliest reference to the problem is the conjecture of Guthrie in 1852 that all maps (planar graphs) can be colored using no more than four colors. Subsequent work on that problem not only spurred the development of graph theory, but ultimately led to the famous four-color theorem of Appel and Haken [**2**], based on an extensive computer search.

Graph coloring has many uses beyond map coloring, and one would expect that computers can be used to solve not just the four-color problem, but general graph coloring problems. It is an abstraction of time-tabling problems, in which lists of courses desired by students are given, and the minimal number of class periods such that all students can take their desired courses must be determined (de Werra [**10**], Leighton [**24**], Opsut and Roberts [**29**]). Garey, Johnson and So [**11**] showed that graph coloring can be used for short circuit testing for printed circuits. Chaitin [**7, 8**] reduced register allocation to graph coloring and used it in a compiler. Poole and Ortega [**30**] showed how to use graph coloring to decompose matrices used to solve sparse systems of linear equations; the decomposition gives a method for easy parallelization of the solution.

Given the practical importance of the graph coloring problem, it is unfortunate that, in theory at least, the cards are stacked against the designer of graph coloring algorithms. Not only is it NP-complete to determine if a graph can be colored with a given number of colors [**20**], but it is also hard even to approximate the chromatic number of a graph. Lund and Yannakakis [**25**] proved that for some $\epsilon > 0$, approximating the chromatic number within a factor of n^ϵ is NP-hard. The best known approximation algorithm, due to Halldórsson [**15**], provides an extremely poor performance guarantee of $O(n(\log\log n)^2/(\log n)^3)$ for an n-node graph. (The performance guarantee is the maximum ratio, taken over all inputs, of the number of colors used over the chromatic number).

On the other hand, the results of Grimmet and McDiarmid [**13**] on coloring algorithms for random graphs offer the algorithm designer some reason for optimism. They consider random graphs such as the $G_{n,p}$ graphs, which have n nodes, where each pair is connected with independent probability p. Their algorithms run in polynomial time and expect to use no more than $2\chi(G)$ colors, where $\chi(G)$ is the chromatic number. However, there is no guarantee that algorithms which work well on random graphs will also work well on data arising from real applications.

There is a fairly extensive body of literature on experimental graph coloring algorithms. These have been tested primarily on random graphs, such as the $G_{n,p}$ graphs mentioned above. One of the simplest coloring algorithms is the Saturation algorithm of Brélaz [**5**]. It is based on the following principle: the vertex which is adjacent to the greatest number of differently-colored neighbors is colored first, with a new color if necessary. Thus, if ever the color of a vertex is forced, that is, there is at most one possible choice from the current set of colors, that vertex is colored first. Another example of such a "successive augmentation" algorithm is the Recursive Largest First (RLF) algorithm, proposed by Leighton [**24**] when studying the exam scheduling problem at Princeton. This algorithm colors the vertices one color class at a time, adding vertices one at a time to the current color class so as to reduce as much as possible the number of edges left in the uncolored subgraph.

These two algorithms have very efficient implementations, but as we will see, do not produce very good colorings on standard test data. Johnson et al. [**17**] pushed the successive augmentation approach much further with the XRLF algorithm, which is essentially a semi-exhaustive version of Leighton's RLF algorithm, based on ideas of Johri and Matula [**18**]. The XRLF algorithm finds better colorings than the simpler successive augmentation algorithms on random $G_{n,p}$ graphs, but takes significantly more time and is beaten by the simpler Saturation algorithm on other randomly generated classes of graphs.

A quite different approach has been taken with iterative improvement algorithms, which include the simulated annealing algorithm of Johnson et al. [**17**], the Tabu Search algorithm of Glover [**14**] and the S-Impasse algorithm of Morgenstern [**26**]. Briefly, iterative improvement algorithms differ from successive augmentation algorithms in that the colors of individual nodes may change several times over the course of the algorithm. Of the iterative improvement algorithms, Morgenstern reports the best results for the S-Impasse algorithm. However, there are several parameters in this algorithm that need to be tuned for different problem instances. We describe his algorithm in more detail later, as our approach is based on his.

Based on reports of this previous work, we draw the following conclusions. It is hard to design a "robust" graph coloring algorithm, that is, one which works well on a wide range of inputs: none of the current approaches clearly dominates the others. The best algorithms, such as XRLF or S-Impasse, are quite sophisticated, with several parameters that need to be tuned, based on knowledge of the input, or by trial and error. Moreover, in order to get good results, sequential implementations may take several hours on a standard workstation, even on relatively small graphs of 1,000 nodes or less, and the time to reduce the size of a coloring by 1 increases greatly as better colorings are found (results reported by Johnson et al. [**17**] and Morgenstern [**26**] are on machines 20-100 times slower than the processors of the CM-5 we used for our experiments). Finally, there is little experimental work on data from real applications of graph coloring.

One natural approach towards overcoming the limitations of previous algorithms is with a parallel implementation, and this is the approach we take in this project. The potential for good speedup is clear. A MIMD environment also provides one way to achieve robustness, namely to run different algorithms on different processors and to take the best solution found. This hybrid approach has the potential for the whole to be greater than the sum of the parts. For example, the different algorithms can progress in a symbiotic fashion, one using a good coloring obtained by another as a basis for further improvement. Finally, it may be possible to dispense with tuning, either through simultaneous use of several parameter settings or simply because the parallel implementation is fast enough that the additional time needed without tuning may be significantly less than the time spent by an implementer in finding the right settings.

We describe experiments with a parallel algorithm, Hybrid, which combines a parallel version of Morgenstern's S-Impasse algorithm with an exhaustive search algorithm. Our algorithm is run on the CM-5, a powerful state-of-the-art parallel computer which gives much flexibility in designing hybrid algorithms. We compare the performance of this algorithm with RLF and Saturation, and with previous results reported by Morgenstern and Johnson et al. Our parallel algorithm outperforms the sequential algorithms, both in the quality of the colorings obtained and in the time spent to obtain the colorings, on all but the simplest test graphs, where all algorithms find an optimal coloring very quickly. However, both Morgenstern's tuned sequential implementation of S-Impasse [**26**] and Johnson et al.'s XRLF algorithm find better colorings than our algorithm on large $G_{n,p}$ graphs. (Recall that our tests of Hybrid are untuned.) Our conclusion is that our parallel methods are certainly useful in solving the graph coloring problems, and are a step towards creating a heuristic that does not require tuning, but do not yet eliminate the need for tuning.

We have collected new test data for our experiments. One test graph is an example of a course scheduling graph, which models the problem of constructing a timetable that allows all students to take their desired courses. The data was obtained from a local high-school, yielding a graph with approximately 400 nodes. Unlike the simpler heuristics, our hybrid algorithm performed extremely well on the course scheduling graph data, finding an optimal coloring very quickly. Another set of test data that we have contributed is based on the problem of register allocation, which arises in compiling programs. Further data, relating to sparse matrix computations, a problem on latin squares, and an exam schedule, are also contributed.

We describe our parallel algorithm in detail in Section 2. Our test data, including both our new contributions and other, randomly generated data, is described in Section 3. The quality of the colorings obtained for our test data, and the running times of our parallel algorithm are discussed in Section 4. A brief concluding summary and directions for future work are presented in Section 5.

2. Algorithm Description

We first briefly review Morgenstern's S-Impasse algorithm [**26**]. We then describe a parallel version of the S-Impasse algorithm, a parallel exhaustive search algorithm, and finally our Hybrid algorithm.

2.1. The S-Impasse Algorithm.

This is an example of an iterative improvement algorithm, proposed by Morgenstern [**26**]. In the following description, the parameters of the algorithm are italicized.

Initially, a *target* number of color classes is chosen, and a naive coloring of the graph is quickly computed. All vertices from color classes beyond the target are placed in an impasse set. The algorithm then repeatedly does the following. If the current best coloring uses less than the target number of colors, then the target is reduced to one less than the current coloring and a new impasse class is created. (This is different from Morgenstern's original algorithm, which halted upon reaching its target. The user can then choose to rerun the algorithm with a smaller target. Our method removes the tuning of the target parameter from the algorithm, following our goal of creating a general purpose algorithm with little need for tuning.) A random vertex is removed from the impasse set and is placed into a random color class, moving neighbors of the vertex into the impasse set. Moves are selected so as to reduce the average degree of vertices in the impasse set; if several such moves are possible, one is chosen at random. With some small probability, a disimprovement is allowed, i.e. a move which increases the average degree of nodes in the impasse set. The probability of disimprovement is controlled by a *temperature* parameter. This allows the algorithm to avoid being trapped at local minima.

The S-Impasse algorithm also incorporates s-chain moves to keep the neighborhood of moves large. An s-chain consists of a tuple $(v, V_0, \dots, V_s)$ where v is a member of the set V_0 and each set V_i is a subset of a distinct color class. The sets V_i have the property that placing set V_i into the color class $V_{(i+1) \bmod s}$ maintains a valid coloring. That is, the s-chain "shuffles" the coloring. (This is a generalization of Kempe chains, an idea used by Kempe [**21**] in his flawed proof of the four-color theorem.) A parameter, μ, is a scaling factor that determines how often the s-chain moves are performed. Morgenstern's version of the algorithm ran for a number of *iterations* before halting. Our algorithm runs until it has reached its *time bound.*

We set our parameters to be the same for all graph classes. The initial target is simply set to be the number of nodes in the graph. For explanations of exactly how the other parameters are used, see Morgenstern [**26**]. In our implementation, the length s of a Kempe chain is set to 3; the temperature is 0.6 and the parameter μ is set to 10. These are the parameters used by Morgenstern [**26**] for $G_{n,p}$ graphs of size less than 500. The temperature was lowered for graphs of size 500 or more. Our hope was that the higher temperature (and corresponding higher acceptance of disimprovements) would be offset by the parallelism finding good solutions faster through multiple search paths. The time bound on the algorithm was set to three hours; this time bound was chosen mainly because the job scheduler on the Connection Machine CM-5 we were using gave priority to jobs of less than three hours.

2.2. The Parallel S-Impasse Algorithm. In the parallel S-Impasse algorithm, several processors are run independently, each finding an initial coloring independently and then setting the initial target to be one less than the number of colors in the best coloring found. The processors then independently explore improvements starting from their initial coloring. When a processor finds a new best coloring, the new bound is broadcast to the other processors and all processors lower the size of the target coloring, moving a color class (the smallest) into the impasse set if necessary. When all processors have completed their total number of iterations, the algorithm stops. Previous studies of simulated annealing algorithms for other applications indicate that if the number of processors is small, this is a reasonable approach to parallelizing the S-Impasse algorithm, because the more paths that are explored independently, the smaller the expected time for one to reach a new best coloring. (See Azencott [**3**], for example.) We later show that this is in fact true for the S-Impasse algorithm on our test data. Another advantage of this parallel approach is that, since the computation of distinct processors is almost completely independent, the amount of interprocessor communication is kept to a minimum.

Processors do occasionally communicate. When a new coloring is found, a limited number (five of thirty-two in our experiments) of processors whose last target met is three greater than the new bound may abandon their partial solution and receive the current solution. These values were selected after some initial experiments on a $G_{500,0.5}$ graph (comparing the amount of time needed for S-Impasse to find a 53-coloring of the graph). They then do an s-chain move as an extra precaution against duplicating the search the sending processor. (Other processors who similarly appear to be following a dead end in their search but do not receive the current solution, do a large s-chain move instead.) Recent results by Aldous and Vazirani [**1**] give theoretical support to the approach of following the current best solution.

2.3. The Parallel Exhaustive Search Algorithm. This algorithm is a straightforward branch-and-bound procedure, as described by Johnson et al. [**17**]. A tree of partial colorings is expanded, using the size of the current best coloring to prune the tree. Each node in the tree represents a partial coloring and its children are all the possible extensions of that coloring obtained by coloring one more node. In the parallel implementation, a polling mechanism is used by idle processors to obtain work from busy processors. When a processor finds a new best coloring, the number of colors is broadcast to all of the other processors.

2.4. The Parallel Hybrid Algorithm. In this algorithm, a manager process starts a group of processors on parallel exhaustive search and another group on parallel S-Impasse. The manager maintains the current best coloring, and whenever either algorithm finds a new best coloring, this is sent to the manager, which broadcasts it to the other algorithm. Hybrid halts when one of the algorithms finishes its computation. Thus, the upper bound on the running time is the maximum of the time bound given to the S-Impasse algorithm and the completion of the exhaustive search.

We note that an execution of either the parallel S-Impasse or exhaustive search algorithms cannot be easily serialized to give an execution of the corresponding sequential algorithms. In particular, the parallel S-Impasse algorithm follows several paths in the search space, whereas the sequential algorithm follows just one path. Similarly, the order in which nodes of the search tree are expanded in the parallel exhaustive search algorithm may be quite different than in the sequential algorithm. (Time slicing could be used to implement the parallel algorithms on a sequential machine, as long as separate random streams are used for each virtual processor.)

3. Test Data

Our algorithms have been tested on random graphs, Leighton graphs, register allocation graphs and two graphs constructed from a course scheduling problem. Below we give a brief description of each graph class.

3.1. Random Graphs.

3.1.1. *Leighton Graphs.* Leighton graphs [**24**] are random graphs with a fixed number of edges and predetermined chromatic number. The graphs are constructed by implanting cliques of sizes ranging from $\chi(G)$ to 2 into the graph. The vertices in the cliques are chosen in a manner that guarantees the chromatic number will not be larger than the pre-specified value. The density of these graphs is always less than 0.25, which Leighton claims is commonly the case in applications such as exam scheduling. (We have collected real data that supports this claim.) The experiments test the algorithms on Leighton graphs with 450 vertices and chromatic numbers 5, 15 and 25. In our experiments we label these graphs as **le450_**χ**x**, where χ is the chromatic number of the graph and **x** is a letter a, b, c or d, designating a particular graph of the given chromatic number.

3.1.2. $G_{n,p}$ *Graphs.* Commonly used in testing graph coloring algorithms, a $G_{n,p}$ graph has n vertices, and an edge between each pair of vertices with independent probability p. We test our algorithms on $G_{n,0.5}$ graphs for $n = 70$, 125, 250, 500 and 1,000.

3.1.3. *Geometric Graphs.* Geometric random graphs, $Rx.y$, are formed by randomly placing x vertices in a unit square, then putting edges between any two vertices which are within $0.y$ of each other. These graphs may model applications such as assigning cellular phone frequencies [**17**]. The algorithms are tested on eight instances of geometric graphs, of size 125, 250, 500, and 1000, with y parameters of 1 and 5. We also test them on complements of $Rx.1$ graphs, denoted $Rx.1c$. (Some of these geometric graphs are actually labeled DSJRx.y, indicating they come from the Johnson et al. [**17**] benchmarks. The Johnson et al. paper referred to these graphs as $U_{n,d}$ graphs, where n is the number of vertices and d is the distance parameter.)

3.1.4. *Flat Graphs.* These graphs are $G_{n,p,k}$ graphs modified to have the density of a $G_{n,p}$ graph. The experiments look at graphs of size 300 with $k = 20, 26, 28$ and of size 1000 with $k = 50, 60, 76$. The target p is 0.5. The graphs are designated **flat_**n**_**k. See Culberson and Luo [**9**] for more discussion on the construction of these graphs.

3.2. Application Graphs. We have gathered data from several problems commonly referred to as applications of coloring. Our experiments test the saturation and RLF algorithms on these graphs as well as the Hybrid algorithm. We also attempt to get an exact coloring of each of these graphs.

In the course of experimenting with these graphs we have found that for several of the applications, the graphs are easy to color but the problem posed in the application is not being exactly solved. Instead colorings involving additional restrictions or relaxations of proper coloring are needed. Below we describe the applications and how there solution varies from simply solving the graph coloring problem presented.

3.2.1. *Register Allocation Graphs.* Register allocation graphs are used in compilers to determine a mapping of variables to registers. Variables that are active in the same range of code cannot be placed in the same register. A conflict graph is constructed, with variables as vertices and an edge representing variables live in the same range of code. Coloring this graph yields a mapping of variables to registers. If more colors are needed than there are registers, not all variables can be placed in registers. In this case, spill code must be inserted to remove some variables from the registers. Spill code removes some edges from the conflict graph; this subgraph can then be colored to see if the variables can be mapped to the registers with the spill code.

Preston Briggs of Rice University has constructed a system to test register allocation schemes. He has provided us with many program files along with code to output the original conflict graphs and several subgraphs constructed as spill code is inserted. The conflict graphs range in size from a couple of hundred vertices to several thousand vertices. In this study we consider conflict graphs ranging in size from around 200 vertices to around 850 vertices. Specifically, we study twelve graphs from four different base programs, **mulsol**, **zeroin**, **fpsol2** and **inithx**. These graphs were constructed for a compiler having thirty-two registers available. There are three graphs from each of these programs, resulting from the initial graph and the graphs formed from spill code insertion (conflict graphs were generated from spilling until the graph was colored with thirty-two or fewer colors and the spill code did not spill vertices that had negative cost – which indicated that there was no need to spill that vertex). The register allocation scheme used by Briggs was based on the arena algorithm for memory allocation of Hanson [**16**].

3.2.2. *Course Scheduling Graphs.* Many high schools ask students to select a set of desired courses for the coming year and then attempt to construct a timetable scheduling sections of the courses and assigning students to specific sections in a way that allows students to take as many of their desired courses as possible. A timetable with no conflicts corresponds to a coloring of a graph with vertices corresponding to sections of courses and edges between two vertices if a student is assigned to both sections or the same teacher teaches both courses.

The Course Scheduling Problem is quite difficult. The timetable construction is obviously NP-Complete since graph coloring can be reduced to it. The assignment of students to sections has a large effect on the chromatic number and ease of coloring the conflict graph that is used to construct the timetable because the section assignments determine the structure of the conflict graph. Unfortunately, assigning students to sections in the best possible way is also difficult, and one approach, minimizing the density of the resulting conflict graph (in the hopes that this will reduce the chromatic number of the conflict graph), is NP-Complete [**22**]. Lewandowski [**22**] discusses this problem in much greater detail. Here we note that the problem is more involved than simply coloring, but concentrate only on the coloring problem, namely the construction of the timetable after students are assigned to sections of courses.

We have obtained the scheduling data from a high school that has approximately 500 students. There are seven periods in a school day and the entire year (two semesters) is scheduled at one time. Counting each section of a course separately, there are 385 vertices in the class graph. Since we are working from a final schedule, the graph is guaranteed to be 14-colorable.

For this study we have looked at algorithm performance on the entire graph, called **School**, as well as the subgraph corresponding to removing all references to study halls, called **School-nsh**.

3.2.3. *Generated Course Scheduling Graphs.* We have also experimented with a generator for graphs modeling the class graphs used in this study. We started by examining the real data to get an idea of the important factors affecting the composition of the class graph.

Since we have only one data set we cannot be sure of all the factors, nonetheless we believe the following factors are important in the composition of a class graph. The first important factor is division of students into separate groups, such as different grades and/or different tracks of study. These divisions are important because each of these groups will have a core set of classes taken only by students in this group – and the vertices corresponding to classes taken by different groups will be independent. The second important factor is the selection of courses that intersect the interests of various groups; for example music courses will be shared between all grades. The probability of students in a group taking courses that also interest another group will vary among the groups. This variance will affect the number of edges between the courses from these groups. Currently, our generator concentrates on handling these two factors.

Our generator uses the *CourseGroup* model described above to generate random data for the course scheduling problem. The generator takes as input the number of students, the number of courses, and the number of courses selected by each student. It also receives as input specification of *student groups* dividing the students into subgroups and specification of *course groups*, specified by listing the number of courses in the group and probabilities p_s that a student in student group s will select a course from this group. The generator builds a schedule for each student by selecting courses randomly from different course groups, following the probabilities specified in the course group descriptions. Selection of courses within a given course group is uniform, though students are not allowed to select the same course more than once.

After student schedules are constructed, students are placed into sections of their desired courses by arbitrarily ordering the students and then placing them into the first available section of each of their courses. (There are always enough sections of every course to accommodate all students.) The resulting conflict graph is the data used in this coloring study. For further discussion of the generators and other possible techniques for assigning students to sections, see Lewandowski [**22**].

As a basis for comparison, we compare our results on a generated graph, **CG0** with a $G_{n,p}$ graph of the same number of nodes and density, $G_{326,0.22}$, and with a graph based on the real data but with students placed into sections of their courses in the naive manner described above. This graph is referred to as **School-as**.

3.2.4. *Latin Square Graph.* This graph represents a problem from design theory, relating to latin squares. The graph is a 900 vertex graph, with independent sets no larger than 10 vertices. The graph represents a formulation of a question regarding the number of orthogonal size 10 latin squares. It is an open problem whether or not this graph can be colored in 90 colors. (This graph was provided by Wendy Myrvold [**28**].) Unlike the other application and random graphs in our test suite, this graph has a high density of about 76%.

3.2.5. *Graphs for Parallelizing Iterative Solutions of Sparse Linear Systems.* Solving large sparse linear systems can be done iteratively in parallel by updating independent components of the system in parallel. Treating the matrix representing the system as an adjacency matrix, with positive entries representing edges, the coloring of the graph with k colors reveals a decomposition of an iteration into k steps. At step i of an iteration, all the components in color class i are updated in parallel. The parallelization does not actually require a true coloring of the graph; it is sufficient to color the vertices so that no positive cycle contains only vertices of the same color. (For more discussion see, for example, Poole and Ortega [**30**] and Jones and Plassmann [**19**].) There are many applications of sparse linear systems; our graphs are examples from power systems, of sizes 1993, 1084, 707, and 147. We refer to these graphs as **sparse1993**, **sparse1084**, **sparse707** and **sparse147** respectively.

3.2.6. *Final Exam Scheduling.* As with class scheduling, final exam scheduling must avoid scheduling courses taken by the same student simultaneously. Vertices are courses, edges are between courses taken by the same student. We have acquired data from Florida Institute of Technology (provided by Lynn Kiaer [**23**]) for exam scheduling. Each edge has an integer weight between one and three, representing the severity of the conflict. (The severity of the conflict increases by an order of magnitude as the weights increase, thus it is imperative to avoid conflicts of type three.) To schedule the exams, the graph must be colored with six or fewer colors – this may require leaving conflicts in the coloring. The goal of the problem is to have as few as possible conflicts of the highest weights. We examine three graphs from this data; graph **fl-tech.1** contains all the edges, **fl-tech.2** removes edges of least severity, and **fl-tech.3** contains only the edges of highest weight.

4. Coloring Results

In this section, we discuss the performance of the Hybrid algorithm on the different test graph types listed in Section 3. For each class of graphs, we compared Hybrid with the simpler RLF and Saturation heuristics, and also compared the quality of our colorings with those obtained in previous work.

All of our algorithms were run on Thinking Machine's Connection Machine CM-5. RLF and Saturation were run on 1 processor, while Hybrid was run on 32 processors, which were partitioned with one acting as manager, 8 executing the parallel exhaustive search algorithm, and 23 executing the parallel S-Impasse algorithm. This partitioning was arrived at through initial experiments which revealed that for $G_{n,p}$ graphs in which exhaustive search finished within an hour, speedup was near linear for up to eight processors and then began to drop. We also reasoned that for more difficult graphs, the S-Impasse portion of the heuristic would be more important in finding better colorings during much of the algorithm. Thus our partitioning was set to maximize parallel speedup on easily colored graphs while providing as many processors as possible for S-Impasse. As noted in our discussion of S-Impasse, each computation was stopped after three hours; we report the time less than that if the best coloring was found earlier. A nice feature of our Hybrid algorithm is that if the exhaustive search procedure is completed, the optimal coloring is known. Thus, we are able to report optimal coloring sizes for several graphs. All of these results are summarized in the tables and figure found in Appendix I. Briefly, our main conclusions are as follows.

- Overall, the Hybrid algorithm performs well, producing good colorings on a wider range of graphs than any previously reported algorithm. This is perhaps not too surprising, since we are using a powerful machine, and combine more than one previously touted technique. Still, our results are obtained with no tuning of parameters; this is an important advantage over previous work. Our algorithm performs most poorly on large, randomly generated graphs. For example, using his tuned S-impasse algorithm, Morgenstern [**26**] obtains better results on large $G_{n,p}$ graphs. Also, Johnson et al. [**17**] obtain better colorings with their XRLF algorithm. A detailed comparison of Hybrid with other algorithms on different classes of graphs is given in Section 4.1.
- Of the five types of test data, three are easily colored even by the simple RLF and Saturation heuristics; one is optimally colored by Hybrid but not by the simple heuristics, and one appears to be very hard.
- The Hybrid algorithm parallelizes well. One reason is because the number of iterations needed by S-impasse decreases as the number of processors increase, supporting our approach of independently exploring several coloring modifications independently. Also, in the exhaustive search algorithm the work involved in expanding the search tree is effectively shared among the processors. We present some experimental data supporting this in Section 4.3.
- There is often quite a variance in the running time depending on the random seed. The variance exists even when disregarding runs in which the size of the coloring differed. Table 8 summarizes the average running time and variance for the Hybrid algorithm on several graphs. We also examined repeated runs of Hybrid using the same random seed, to see if machine effects cause the variance, but found very little difference in running time while using the same seed.

4.1. Evaluation of Randomly Generated Data.

4.1.1. *Leighton Graphs.* (See Table 1.) Hybrid outperformed the simpler RLF and Saturation heuristics on all Leighton graphs. Hybrid found optimal colorings on all of the 5-colorable graphs, two of the four 15-colorable graphs and two of the four 25-colorable graphs. Close to optimal colorings (off by one or two) were found for the remaining graphs. The time required to find the best colorings ranged from less than half a minute (on a 25-colorable graph) to almost three hours (on a 15-colorable graph).

In the execution of the Hybrid algorithm on the 5-colorable Leighton graphs, the exhaustive search and parallel S-Impasse alternated regularly in decreasing the current best number of colors, finishing with the exhaustive search. On the other Leighton graphs, with chromatic numbers 15 and 25, the exhaustive search algorithm was useful in decreasing the current best coloring for several colors in the initial stages, and to end the computation if the chromatic number was found. Progress after the initial stages, however, was made by the S-Impasse algorithm.

Hybrid also surpassed previous results of Morgenstern [**26**]. His tuned, sequential S-Impasse algorithm only found a 20-coloring, in somewhat less than an hour, for each of the two 15-colorable graphs on which we find a 16-coloring.

4.1.2. *Random $G_{n,p}$ Graphs.* (See Table 1.) While Hybrid easily outperformed Saturation and RLF on $G_{n,p}$ graphs, it failed to come close to the lower bound estimates on the large $G_{n,p}$ graphs. On the $G_{500,0.5}$ graph, which has an estimated lower bound of 46, the best coloring found was 52 colors, found by Hybrid in about 2.19 hours. On the $G_{1000,0.5}$ graph, which has an estimated lower bound of 80, the best coloring found was 99 colors, found in just under 2.5 hours. Still, Hybrid always outperformed the simpler heuristic algorithms on all of these graphs.

The poor results are perhaps not too surprising, as Morgenstern [**26**] also reported difficulty in obtaining good colorings for large $G_{n,p}$ graphs, even with his tuned sequential S-Impasse algorithm. For example, he reports running times of about 65 hours to find a 90-coloring of a $G_{1000,0.5}$ graph, and 28.2 hours to find a 50-coloring of a $G_{500,0.5}$ graph (these are times on a VAX 11/780, around twenty times slower than the CM-5 used in our experiments).

4.1.3. *Geometric Graphs.* (See Table 1.) On six of the twelve geometric graphs (R125.1, R125.1c, R250.1, R250.1c, DSJR500.1, R1000.1) we were able to find optimal colorings using Hybrid. There are no previous results on R125.5, R250.5, R1000.1c, and R1000.5 and Hybrid did not prove it found an optimal coloring so we do not know how close our colorings are to the optimal colorings. For the 500 vertex graphs, DSJR500.1, DSJR500.1c and DSJR500.5, previous results were reported by Johnson et al. [**17**] so we compare our results with theirs on these graphs.

The performance of Hybrid was mixed on the 500 vertex graphs. On DSJR500.1, the graph for which an edge is between two points if the distance between them is less than 0.1, 12 colors is optimal, and Hybrid found a 12-coloring in less than half a minute (and most of this time is actually spent reading the graph). On the graph with an edge between two points if the distance between them is greater than 0.9 (DSJR500.1c), Hybrid found an 85-coloring in three out of four runs, in an average time of about two hours. In contrast, the best colorings obtained by the RLF and Saturation algorithms were of size 89 and 88, respectively. Also, Johnson et al. [**17**] reported that the best coloring they obtained was of size 85, using a tuned version of an annealing-type algorithm (the Fixed-K Annealing algorithm) which ran for over 75 hours on a Sequent. On the third graph, with an edge between two points if the distance between them is less than 0.5

(DSJR500.5), Hybrid obtained a coloring of size 128 in about 1.5 minutes, but failed to find a better coloring in three hours. Johnson et al. report that in several runs of the Saturation algorithm, a coloring of size 124 was obtained (they report that 1% of 1000 runs of Saturation on permutations of the vertices of the graph give a 124-coloring) so Hybrid fails to match this coloring.

4.1.4. *Flat Graphs.* (See Table 1.) Again, Hybrid did not come close to the optimal colorings on these graphs, although it found better colorings than RLF and Saturation. When given an initial target corresponding to k, instead of having Hybrid start from an initial coloring and steadily decrease the number of colors, Hybrid successfully found the optimal coloring of two of the 300 vertex graphs (Table 1 presents results achieved by setting the target automatically so these optimal results do not appear).

4.2. Evaluation of Application Data. Table 4 summarizes the information about the application graphs discussed in this section.

4.2.1. *Course Scheduling Graphs.* (See Table 2.) For both course scheduling graphs, an optimal 14-coloring was found by Hybrid. Surprisingly, for the School graph, the exhaustive search algorithm was the main workhorse in Hybrid. To explore how important the exhaustive search was, we ran the parallel S-Impasse algorithm alone on this graph and found that it took (on average) 78 seconds to find the 14-coloring, compared to an average of 46.2 seconds for the Hybrid algorithm. The fast sequential heuristics did not do as well on the School graph. The Saturation algorithm found a 17-coloring of the School graph, while the RLF algorithm never did better than a 26-coloring.

The School-nsh graph was more difficult to color than the School graph. Hybrid took 66.4 seconds to find an optimal coloring, with exhaustive search finding several early colorings, then S-impasse taking over the work until the last few colorings, including the optimal, which were found by exhaustive search. We again ran the parallel S-Impasse algorithm on this graph to evaluate the usefulness of the exhaustive search component of Hybrid. The parallel S-Impasse algorithm took longer, 2.7 minutes, to find the optimal coloring. Among the fast sequential heuristics, RLF out did Saturation on the School-nsh graph, using 22 colors compared to Saturation's 28 colors.

Even though the graphs have over 300 vertices and are of density around 0.25, exact coloring is quite feasible. The Hybrid algorithm completed its exhaustive search on the School graph in an average of 78 seconds, and on the School-nsh graph in an average of 90 seconds.

The assignment of sections greatly affects the chromatic number, as we see below in our comparison of the generated class graph to the real data with section numbers added automatically.

4.2.2. *Generated Course Scheduling Graphs.* (See Table 2.) We found that the results on the generated graph appear to be more similar to the results on the real data with sections added by the generator than to the $G_{n,p}$ graph of the same density. In particular, the $G_{n,p}$ graph was colored with far fewer colors.

School-as was colored in 28 colors by both the RLF and Saturation algorithms. Hybrid found a 23-coloring of the graph. The graph CG0, built by our generator based on parameters similar to the real data, was colored by RLF in 30 colors, and Saturation in 31 colors. The Hybrid algorithm found a 28-coloring of the graph. The density of both School-as and CG0 is around 0.22. We built a random $G_{326,0.22}$ graph, having the same number of nodes and the same density as CG0. This graph was colored with far fewer colors by RLF and Saturation, 20 and 22 respectively. Hybrid found a 17-coloring.

Our conclusion is that our generator has constructed a graph that is a better model for testing heuristics on class graphs than a $G_{n,p}$ graph. We note, however, that model graphs do not yet appear to capture all aspects of the real data. The largest clique in the model graph is 26 while the school-as graph has a clique of size 18. Further discussion of models for the course scheduling problem can be found in Lewandowski [**22**].

Unlike the 14-colorable school graphs, the model graph and the school graph with sections generated automatically are not easily exactly-colored. The Hybrid algorithm spent around an hour on each graph without finishing its search.

4.2.3. *Register Allocation Graphs.* (See Table 2.) The Saturation and RLF algorithms performed just as well as Hybrid on these graphs, and all algorithms halted quickly. We verified that the results were optimal by using dfmax [**6**] to find the largest clique in each of the graphs; these cliques were the same size as the colorings found.

Although Hybrid did not prove that it exactly colored the register allocation graphs, it would be easy to add dfmax to the implementation to prove the colorings optimal.

4.2.4. *Graphs for Parallelizing Iterative Solutions of Sparse Linear Systems.* (See Table 2.) The RLF algorithm optimally colored all of the graphs. The Saturation algorithm exactly colored three of the four graphs and used four instead of three colors to color sparse1084. Two of the graphs, sparse147 and sparse1084, were quickly exactly colored by the sequential exhaustive search algorithm. Lower bounds on the other two graphs were found using dfmax.

Because the sequential heuristics worked so quickly, we did not run Hybrid on these graphs. However, we did attempt to exactly color the graphs using the parallel exhaustive search algorithm alone. The search proved the optimal colorings for sparse147 and sparse1084, but could not prove the coloring for sparse707 was optimal and could not run on sparse1993 due to memory problems on the CM-5.

4.2.5. *Final Exam Scheduling Graphs.* (See Table 2.) Graphs fl-tech.1 and fl-tech.2 were quickly and exactly colored using the Hybrid heuristic. The third graph, fl-tech.3 was quickly colored with 6 colors, but Hybrid did not show it to be exactly colored. Both RLF and Saturation found the minimum coloring on each graph. Only fl-tech.3 was colored with the six colors needed to actually effectively schedule the exams. The largest clique in each of these graphs is equal to the size of the minimum coloring.

Kiaer has constructed heuristics to use the weights of the conflicts to find a 6 coloring with no severe (weight 3) conflicts, 5 medium (weight 2) conflicts and 42 small (weight 1) conflicts [**23**].

4.2.6. *Latin Square Graph.* (See Table 2.) All the algorithms performed poorly on this application. The RLF algorithm used at least 146 colors to color this graph. The Saturation algorithm used 132 colors. As in the course scheduling graphs, this is a reversal of the typical performance of RLF and Saturation on random graphs. The Hybrid algorithm always used at least 109 colors on this graph. We conclude that this graph is a hard test for graph coloring heuristics. (Morgenstern [**27**] has the best results on this graph, a 98-coloring.)

The difficulty in coloring the Latin square graph most likely comes from its regular structure. To achieve the hoped-for coloring of size 90, each color class must be of size exactly ten, since no independent set is larger than ten vertices. An error in coloring a single vertex could therefore have dire consequences for the entire coloring.

Exact coloring appears completely infeasible at this point for the Latin Square graphs.

4.3. Advantages of Parallelism. Clearly, the Hybrid algorithm must always produce colorings that are as least as good as the sequential S-Impasse algorithm. Our hopes in undertaking this project were that Hybrid would produce good colorings in significantly less time than the sequential S-Impasse algorithm. We were also curious if the exhaustive search component of Hybrid would be useful. In order to understand the advantages of a parallel implementation, we compared the parallel and sequential versions of the S-impasse and exhaustive search algorithms separately.

4.3.1. *Parallel vs. Sequential S-Impasse.* Our parallel version of S-Impasse found good colorings faster than the sequential version. For several graphs, we counted the number of iterations needed to find the best coloring, in the parallel and sequential algorithms, and found that the number of parallel iterations was consistently much less in the parallel implementation (see Table 6). For example, on the 15-colorable Leighton graph (le450_15c) the sequential algorithm required 2770 iterations to find a 27-coloring, while the parallel algorithm required only 350; the sequential algorithm needed 51,260 iterations to find a 23-coloring, while the parallel algorithm needed 17,663; and the sequential algorithm never did better than 23 colors, while the parallel version found a 21 coloring. Further testing is needed to see a more general correlation between the improvement in performance and the number of processors. Work by Morgenstern [**27**] and Aldous & Vazirani [**1**] supports our belief that in general, having the processors work in parallel yields better colorings faster than simply using multiple independent runs.

4.3.2. *Parallel vs. Sequential Exhaustive Search.* We ran several additional experiments to test the parallel exhaustive search algorithm, and concluded that it performs very well. Table 3 shows the running time of parallel exhaustive search on a $G_{70,0.5}$ graph, with the total number of nodes expanded, and the minimum and maximum number expanded by each processor. While the 70 vertex graphs are not big enough to give all thirty-two CM-5 processors work at all times, a speedup of a factor of 3.2 is observed for four processors and more speedup is observed as processors increase. Also encouraging is that the number of search nodes examined in the parallel implementation is rarely more than five percent more than the total number of nodes examined the sequential implementation. In other studies of several graphs, including the Leighton graphs, we found that the number of nodes expanded by the processors are generally within ten percent of each other.

4.3.3. *Hybrid vs. Parallel S-Impasse.* We compared the parallel S-Impasse algorithm with the Hybrid algorithm on several graphs: School, School-nsh, $G_{125,0.5}$, $G_{500,0.5}$, le450_5c, le450_15c, and le450_25c. We found that the Hybrid algorithm often performed similarly to the parallel S-Impasse algorithm in terms of the quality of the colorings, but often achieved these colorings faster. On the $G_{500,0.5}$, le450_15c and le450_5c graphs, the Hybrid found better colorings than parallel S-Impasse alone, on the other graphs the two achieved the same colorings. The Hybrid algorithm found the coloring faster on all but two of the graphs, the le450_25c and $G_{125,0.5}$ graphs. Table 5 summarizes the results of the two algorithms on all seven of these graphs.

The exhaustive search component of Hybrid was most useful on graphs with low chromatic number. Table 7 indicates how the exhaustive search and S-Impasse components of Hybrid interacted on three Leighton graphs and the School-nsh graph. In three of the cases, le450_5a, le450_15b and School-nsh, the exhaustive search component helped decrease the target for the S-Impasse algorithm in the early stages of computation, and then proved the coloring optimal when S-Impasse found a coloring that was either optimal or within one of optimal. In the le450_15c graph, the exhaustive search component helped decrease the target for the S-Impasse algorithm, but could not find an optimal coloring even though the S-Impasse algorithm found a 16-coloring of the graph.

In proving that colorings are optimal, the exhaustive search component could perhaps be replaced by a clique finder. In particular, the maximum clique matches the coloring bound on the Leighton graphs, the register allocation graphs, school, school-nsh, the sparse matrix graphs, and the exam scheduling graphs. However, because it is useful in the actual coloring of the school graphs, the Leighton graphs, the exam scheduling graphs and the sparse matrix graphs, it seems unlikely that this replacement would yield equally good colorings in the same amount of the time.

Further investigation of the exhaustive search algorithm on the Leighton graphs revealed that unlike many $G_{n,p}$ graphs, for which it is often the case that it is harder to prove a coloring optimal than to find the coloring, proving that no coloring of size $\chi - 1$ exists took little time ($10 - 15$ seconds) for all but the c and d 15-colorable graphs. However, finding the χ-coloring for the c and d graphs of the 15 and 25 colorable Leighton graphs given an upperbound of $\chi + 1$ was very difficult — the parallel exhaustive search was unable to find the coloring in 2 hours.

5. Conclusions and Future Work

We were pleased to find that the Hybrid algorithm performed efficiently, and produced good colorings on a wide variety of graphs. We believe Hybrid represents a step towards building a coloring implementation that requires no tuning by the user but produces excellent colorings. Based on a comparison with Morgenstern's previous results, we conclude that currently, tuning of Hybrid should be useful in improving our results on the large random $G_{n,p}$ graphs, but on all other graph classes we tested, our untuned algorithm matches or exceeds previously reported best results.

Our study of graphs arising from applications shows that several of the applications (register allocation, matrices for sparse linear systems, exam scheduling) provide graphs which are quite easy to color. Course scheduling is a little harder, with RLF and Saturation being insufficient to color the graphs but being easily colored by the Hybrid heuristic. One of the applications, latin square, is very difficult. We conjecture that many applications will fall into the easy or moderately difficult category, corresponding to our observations in this study.

Our study of applications also shows that unlike random graphs, where RLF often tops the performance of Saturation, there is no clear winner when comparing the two on applications. For register allocation, exam scheduling, and most sparse matrices, they gave the same results; RLF was better on one sparse matrix and one course graph, while Saturation was better on the other course graph and the latin square graph.

We also note that coloring the graphs of many of these applications, does not actually solve the problem originally given in the application. Thus, although these graphs do not provide a great challenge to current heuristics for coloring, they do offer a challenge to modify heuristics or add additional algorithm techniques in order to better solve the exact problem posed by these applications. We have already made some progress in the area of course scheduling (see Lewandowski [**22**]), and plan to work on other problems as well in the future.

In the future we also plan to add XRLF to our Hybrid, to run in parallel with

S-Impasse and exhaustive search. In contrast to S-Impasse, the XRLF algorithm runs well on random $G_{n,p}$ graphs, given enough time; for example, an 86-coloring of a $G_{1000,0.5}$ graph is found in 68.3 hours [**17**]. (This was on a VAX 750, which is 20-100 times slower than current machines.) In our implementation of XRLF, we use Hybrid as a replacement for the exhaustive search alone at the end of the XRLF algorithm.

The issue of variance in running time depending on the random seeds used needs to be further explored. It is not clear whether this is simply an effect of the Hybrid algorithm being untuned or if this effect occurs in many of the time-intensive algorithms that use random choices to color. Experiments using several random seeds and several permutations of the graph would give some indication about how sensitive a particular algorithm is to the seed.

Our experience with the course scheduling graphs strongly suggests that more effort should be made to find applications and real data to compare algorithms. Furthermore, although coloring is important in these applications, it is clear that in many cases that the problem is more complex than simply finding good colorings. It is still unclear whether good coloring heuristics can really be applied in these applications.

6. Acknowledgements

We thank Robert Mills, Lynn Kiaer, Preston Briggs, Monika ten Bruggencate and Wendy Myrvold for their helpful discussions about applications and help in acquiring data, and Alex Chan for his work entering the course scheduling data.

We also thank Mike Trick for his work running the second DIMACS challenge, and the referees, whose comments have helped us improve the presentation of these results.

References

1. D. Aldous and U. Vazirani, *"Go with the winners" algorithms*, Proceedings of the 35th IEEE Symposium on the Foundations of Computer Science, 1994, 492–501.
2. Appel, K. I., W. Haken and J. Koch, *Every planar map is four colorable, Part I: Discharging*, Illinois Journal of Mathematics, **21**, 1977, 429–490.
3. R. Azencott, Editor, Simulated Annealing: Parallelization Techniques, New York, John Wiley and Sons, 1992.
4. B. Bollobas and A. Thomason, *Random Graphs of Small Order*, Ann. Discrete Math, **28** 1985, 47–97.
5. D. Brélaz, *New methods to color vertices of a graph*, Communications of the ACM, **22**, 1979, 251–256.
6. Carraghan and Paradalos, *An exact algorithm for the maximum clique problem*, Operation Research Letters, **9**, 1990, 375–382.
7. G.J. Chaitin and M. Auslander and A.K. Chandra and J. Cocke and M.E. Hopkins and P. Markstein, *Register allocation via coloring*, Computer Languages, **6**, 1981, 47–57.
8. G.J. Chaitin, *Register allocation and spilling via graph coloring*, Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction, 1982, 98–105.
9. J. Culberson and F. Luo, *Exploring the k-colorable landscape with iterated greedy*, Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, David S. Johnson

and Michael A. Trick (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1995.

10. D. de Werra, *An introduction to timetabling*, European Journal of Operations Research, **19**, 1985, 151–162.
11. M. R. Garey and D. S. Johnson and H. C. So, *An application of graph coloring to printed circuit testing*, IEEE Transactions on Circuits and Systems, **23**, 1976, 591–599.
12. J.W. Greene and K. J. Supowit, *Simulated annealing without rejected moves*, IEEE Transactions on Computer-aided Design, **CAD-5**, January 1986, 221–228.
13. G.R. Grimmet and C.J.H. McDiarmid, *On colouring random graphs*, Mathematical Proceedings of the Cambridge Philosophical Society, **77**, 1975, 313–324.
14. F. Glover, *Tabu search, part 1*, ORSA Journal on Computing, **1**, 1989, 190–206.
15. M. M. Halldórsson, *A still better performance guarantee for approximate graph coloring*, DIMACS Technical report 1990, 91–35.
16. D. Hanson, *Fast allocation and deallocation of memory based on object lifetimes*, Software – Practices and Experience, January 1990.
17. D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, *Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning*, Operations Research, **3**, 1991, 378–406.
18. A. Johri and D. W. Matula, *Probabilistic bounds and heuristic algorithms for coloring large random graphs*, Technical report, Southern Methodist University, Texas, 1982.
19. M. Jones and P. Plassman, *The efficient parallel iterative solution of large sparse linear systems*, Graph Theory and Sparse Matrix Computation, Alan George, John Gilber, J. Liu, Editors (eds.), IMA volumes in Mathematics and its Applications, **56**, Springer Verlag, 1993.
20. R. M. Karp, *Reducibility among combinatorial problems*, Complexity of computer computations (R.E. Miller and J.W. Thatcher, eds.), Plenum Press, New York, 1972, 85–103.
21. A. B. Kempe, *On the geographical problem of the four-colors*, American Journal of Mathematics, **2**, 1879, 193–200.
22. G. Lewandowski. Practical implementations and applications of graph coloring, Ph.D. thesis, Computer Sciences Department, University of Wisconsin- Madison, 1994.
23. Kiaer, Lynn. Discrete optimization strategies for timetabling, Ph.D. thesis, Department of Applied Mathematics, Florida Institute of Technology, June 1992.
24. F.T. Leighton, *A graph coloring algorithm for large scheduling problems*, Journal of Research of the National Bureau of Standards, **84**, 1979, 489–506.
25. C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, Proceedings 25th ACM Symposium on Theory of Computing, 1993, 286–293.
26. C. A. Morgenstern, *Algorithms for general graph coloring*, Ph.D. thesis, Technical report CS89-16, Department of Computer Science, University of New Mexico, Albuquerque, 1989.
27. ———, *Distributed coloration neighborhood search*, Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, David S. Johnson and Michael A. Trick (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1995.
28. W. Myrvold, Personal communication, October 1993.
29. R.J. Opsut and Fred S. Roberts, *On the fleet maintenance, mobile radio frequency, task assignment and traffic phasing problems*, *The Theory and Applications of Graphs* (G. Chartrand, Y. Alavi, D.L. Goldsmith, L. Lesniak–Foster and D.R. Lick, eds.), John Wiley & Sons, New York, 1981, 479–492.
30. E. L. Poole and J. M. Ortega, *Multicolor ICCG methods for vector computers*, SIAM Journal of Numerical Analysis, **24**, 1987, 1394–1418.
31. D. C. Wood, *A technique for coloring a graph applicable to large scale time-tabling problems*, Computer Journal, **12**, 1969, 317–319.

Appendix I

In the tables below, graphs are described by file name, with the known or estimated chromatic number in parentheses. For each of RLF, Saturation and Hybrid, the following data is listed: (i) The size of the best coloring. (ii) The number of runs achieving this coloring, over the total number of runs. (iii) The average running time, over all runs obtaining the best coloring. Time is given in *minutes:seconds* format.

FIGURE 1. Comparison of Hybrid Colorings with best known sequential colorings. Difference between number of colors used by Hybrid and best coloring known is plotted. (Points are the number of colors used by Hybrid.) Points above 0 represent graphs in which Hybrid obtained better colorings.

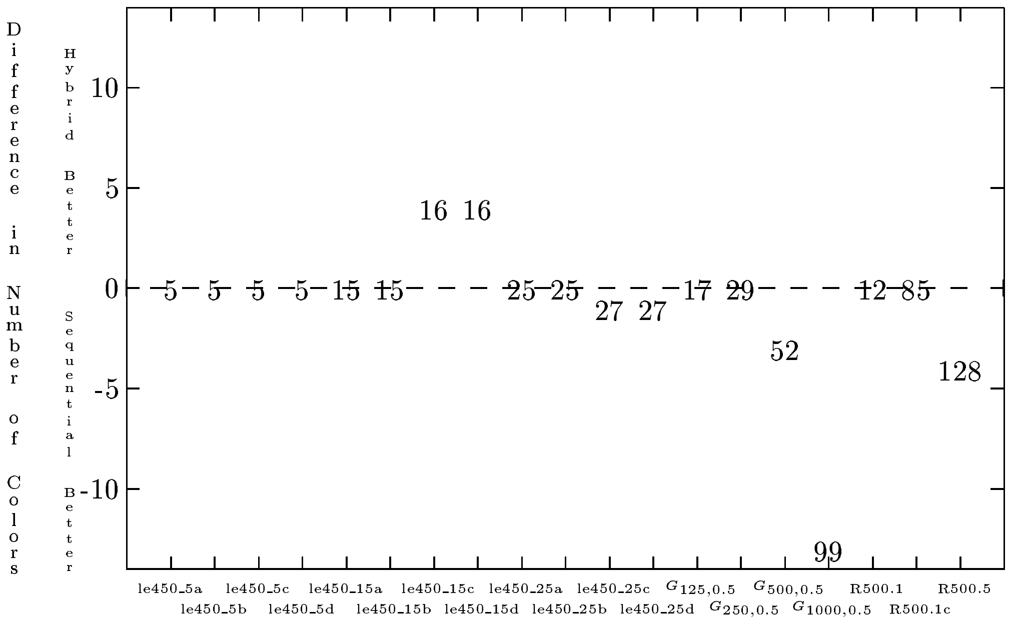

TABLE 1. Random Graphs: Leighton graphs, $G_{n,p}$ Graphs, Geometric graphs, flat graphs.

Graph	RLF			Saturation			Hybrid(32 Procs)		
	(i)	(ii)	(iii)	(i)	(ii)	(iii)	(i)	(ii)	(iii)
le450-5a(5)	8	1/1	0:00:26	12	1/1	0:00:48	5	1/1	0:15:06
le450-5b(5)	7	1/1	0:00:26	11	1/1	0:00:38	5	1/1	0:07:40
le450-5c(5)	5	1/1	0:00:28	13	1/1	0:00:38	5	1/1	0:00:28
le450-5d(5)	8	1/1	0:00:28	13	1/1	0:00:42	5	1/1	0:06:35
le450-15a(15)	17	5/5	0:00:27	17	5/5	0:00:41	15	5/5	0:02:43
le450-15b(15)	17	5/5	0:00:28	17	5/5	0:00:41	15	5/5	0:02:58
le450-15c(15)	24	4/5	0:00:35	24	5/5	0:00:33	16	2/5	0:44:25
le450-15d(15)	24	3/5	0:00:33	24	1/5	0:00:43	16	1/5	1:36:00
le450-25a(25)	25	1/1	0:00:28	25	1/1	0:00:29	25	1/1	0:00:28
le450-25b(25)	25	1/1	0:00:28	25	1/1	0:00:31	25	1/1	0:00:28
le450-25c(25)	28	1/1	0:00:32	30	1/1	0:00:46	27	1/1	0:02:55
le450-25d(25)	28	1/1	0:00:32	31	1/1	0:00:41	27	1/1	0:01:04
$G_{70,0.5}$(11)	14	1/1	0:00:20	13	1/1	0:00:33	11	1/1	0:00:51
$G_{125,0.5}$(16)	22	5/5	0:00:22	23	5/5	0:00:36	17	5/5	1:08:00
$G_{250,0.5}$(27)	35	3/5	0:00:30	37	2/2	0:00:27	29	4/5	1:29:00
$G_{500,0.5}$(46)	62	1/5	0:01:42	63	5/5	0:00:52	52	1/5	2:11:00
$G_{1000,0.5}$(80)	112	3/5	0:09:03	117	5/5	0:01:20	99	1/5	2:17:00
R125.1(5)	5	5/5	0:00:40	5	5/5	0:00:28	5	5/5	0:00:01
R125.1c(46)	46	5/5	0:00:02	46	5/5	0:00:30	46	5/5	0:00:01
R125.5	39	5/5	0:00:20	38	5/5	0:00:34	37	5/5	0:00:32
R250.1(8)	8	5/5	0:00:22	8	5/5	0:00:29	8	5/5	0:00:22
R250.1c(64)	65	1/5	0:00:46	65	5/5	0:00:39	64	5/5	0:04:36
R250.5	70	5/5	0:00:39	67	5/5	0:00:36	66	5/5	0:00:40
DSJR500.1(12)	12	3/5	0:00:25	14	5/5	0:00:38	12	5/5	0:00:27
DSJR500.1c	89	2/4	0:02:30	88	5/5	0:01:26	85	3/4	2:00:00
DSJR500.5	132	5/5	0:02:00	130	5/5	0:00:53	128	5/5	0:01:30
R1000.1(20)	20	5/5	0:00:50	20	5/5	0:01:01	20	5/5	0:00:50
R1000.1c	104	2/4	0:10:00	104	5/5	0:04:18	101	1/5	2:24:00
R1000.5	261	2/2	0:12:00	248	5/5	0:02:36	243	1/5	0:03:42
flat300_20(20)	39	4/5	0:00:58	41	5/5	0:00:52	20	5/5	0:04:30
flat300_26(26)	40	1/5	0:00:44	41	5/5	0:00:39	32	3/5	2:30:00
flat300_28(28)	40	3/5	0:00:43	43	5/5	0:00:38	33	5/5	0:32:00
flat1000_50(50)	109	1/5	0:08:36	114	5/5	0:02:18	96	1/5	2:18:00
flat1000_60(60)	110	3/5	0:08:54	112	5/5	0:02:30	97	2/5	1:54:00
flat1000_76(76)	113	2/4	0:08:54	115	5/5	0:02:30	99	4/4	1:48:00

TABLE 2. Application-related graphs: Register Allocation Graphs, sparse matrix graphs, exam scheduling graphs, latin square graph, and course scheduling graphs.

Graph	RLF			Saturation			Hybrid(32 Procs)		
	(i)	(ii)	(iii)	(i)	(ii)	(iii)	(i)	(ii)	(iii)
mulsol.1 (49)	49	1/1	00:27	49	1/1	00:34	49	1/1	00:27
mulsol.2 (31)	31	1/1	00:26	31	1/1	00:36	31	1/1	00:26
mulsol.3 (31)	31	1/1	00:27	31	1/1	00:33	31	1/1	00:26
zeroin.1 (49)	49	1/1	00:27	49	1/1	00:32	49	1/1	00:27
zeroin.2 (30)	30	1/1	00:26	30	1/1	00:36	30	1/1	00:26
zeroin.3 (30)	30	1/1	00:26	30	1/1	00:36	30	1/1	00:26
fpsol2.1 (65)	65	1/1	00:29	65	1/1	00:29	65	1/1	00:29
fpsol2.2 (30)	30	1/1	00:26	30	1/1	00:25	30	1/1	00:25
fpsol2.3 (30)	30	1/1	00:26	30	1/1	00:26	30	1/1	00:26
inithx.1 (54)	54	1/1	00:45	54	1/1	00:42	54	1/1	00:43
inithx.2 (31)	31	1/1	00:41	31	1/1	00:39	31	1/1	00:39
inithx.3 (31)	31	1/1	00:41	31	1/1	00:39	31	1/1	00:39
sparse1084(3)	4	5/5	00:40	4	5/5	00:06	Did not run		
sparse1993(4)	4	5/5	00:07	4	5/5	00:09	Did not run		
sparse707(10)	10	5/5	00:09	10	4/5	00:03	Did not run		
sparse147(2)	2	5/5	00:01	2	5/5	00:01	Did not run		
fl-tech.1(11)	11	5/5	00:23	11	5/5	00:39	11	5/5	00:23
fl-tech.2(8)	8	5/5	00:24	8	5/5	00:44	8	5/5	00:24
fl-tech.3(6)	6	5/5	00:23	6	5/5	00:38	6	5/5	00:23
Latin Square	146	2/4	00:09:48	132	5/5	00:02:30	109	3/4	1:54:00
School (14)	26	5/5	00:32	17	5/5	00:33	14	5/5	00:46
School-nsh (14)	22	5/5	00:31	28	5/5	00:42	14	5/5	01:06
School-as	30	2/2	0:00:05	28	2/2	0:00:15	23	1/2	0:06:37
CG0	32	2/2	0:00:04	31	2/2	0:00:15	28	2/2	0:02:49
$G_{326,0.22}$	20	1/1	0:00:04	22	1/1	0:00:12	17	1/1	0:32:07

TABLE 3. Parallel exhaustive search. The running time, total nodes expanded, minimum and maximum nodes expanded by a single processor are given for a $G_{70,0.5}$ graph.

No. Procs.	Running	Total Nodes		Min Nodes	Max Nodes
1	19:30	323,881	(100%)	323,881	323,881
2	10:42	334,156	(103.2%)	163,040	171,116
4	06:04	327,621	(101.2%)	73,556	99,279
8	05:29	359,681	(111.1%	35,234	79,744
16	05:13	340,893	(105.3%)	12,382	73,925

TABLE 4. Summary of information about application graphs and generated course scheduling graphs. The latin square result marked with * was found by Morgenstern, all other results are from this paper.

Graph Name	Number vertices	density	Largest Clique known	Best lowerbound	Best coloring
mulsol.1	197	0.20	49	49	49
mulsol.2	188	0.22	31	31	31
mulsol.3	184	0.23	31	31	31
mulsol.4	185	0.23	31	31	31
mulsol.5	186	0.23	31	31	31
zeroin.1	211	0.19	49	49	49
zeroin.2	211	0.16	30	30	30
zeroin.3	206	0.17	30	30	30
fpsol2.1	496	0.09	65	65	65
fpsol2.2	451	0.09	30	30	30
fpsol2.3	425	0.10	30	30	30
inithx.1	864	0.05	54	54	54
inithx.2	645	0.07	31	31	31
inithx.3	621	0.07	31	31	31
school	385	0.23	14	14	14
school-nsh	352	0.24	14	14	14
CG0	326	0.22	26	26	28
school-as	324	0.26	18	18	23
latin_square	900	0.76	90	90	98*
sparse1084	1084	0.004	3	3	3
sparse1993	1993	0.01	4	4	4
sparse707	707	0.01	10	10	10
sparse147	147	0.06	2	2	2
fl-tech.1	70	0.23	11	11	11
fl-tech.2	70	0.13	8	8	8
fl-tech.3	70	0.03	6	6	6

TABLE 5. Parallel S-Impasse vs Hybrid on five graphs.

Graph	Hybrid		Parallel S-Impasse	
le450_5c	5	0:00:28	6	0:09:00
le450_15c	16	0:44:25	17	1:12:50
le450_25c	27	0:02:55	27	0:49:00
school	14	0:00:46	14	0:01:18
school_nsh	14	0:01:06	14	0:02:42

TABLE 6. Comparison of number of iterations and time needed by parallel and sequential S-Impasse algorithms to decrease colorings for three graphs. A * in an entry indicates that the program was not able to achieve this coloring in three hours. Time is given in *hours:minutes:seconds* format.

Graph	Coloring Size	Parallel S-Impasse		Sequential S-Impasse	
$G_{70,0.5}$	14	0	0:00:00	0	0:00:00
	13	12	0:00:08	21	0:00:01
	12	92	0:00:14	557	0:00:06
	11	523	0:00:29	7757	0:01:38
le-450.15c	29	1	0:00:00	434	0:00:37
	27	350	0:00:56	2770	0:03:16
	25	4915	0:04:24	11963	0:10:52
	23	17663	0:13:56	51260	0:51:26
	22	41861	0:28:26	*	
	21	148349	1:16:30	*	
school-as	28	48	0:00:20	0	0:00:04
	27	154	0:00:21	305	0:00:11
	26	823	0:00:43	686	0:00:16
	25	1789	0:00:49	4767	0:01:06

TABLE 7. Cooperation in Hybrid. The progression to the final coloring is given for a single run (each run looked similar). The coloring is reported in the row corresponding to the component of Hybrid that found it. A * indicates the exhaustive search proved the coloring was optimal.

le450_5a	Exh		11	10					5				
	S-Imp	12			9	8	7	6					
le450_15b	Exh			18	17			*					
	S-Imp	21	20			16	15						
le450_15c	Exh			25	24								
	S-Imp	29	28			23	22	21	20	19	18	17	16
school_nsh	Exh	23				15	14	*					
	S-Imp		22	21	20								

Appendix II

Second DIMACS Challenge

Coloring Benchmark Results

GENERAL INFORMATION Authors: Gary Lewandowski and Anne Condon

Title: Experiments with Parallel Graph Coloring Heuristics

Name of Algorithm: Hybrid

Brief Description of Algorithm: Heuristic: Parallel Hybrid of parallel branch and bound exhaustive search algorithm and parallel S-Impasse algorithm.

Type of Machine: Connection Machine CM-5
Compiler and flags used: g++, -g flag

MACHINE BENCHMARKS

User time for instances:

r100.5	r200.5	r300.5	r400.5	r500.5
1.83	14.38	122.88	773.39	2993.58

ALGORITHM BENCHMARKS

Authors' Comments: Each run was time bounded by three hours. We consider machine crashes before three hours to be failures, with the exception of R1000.5.col which always crashed after 15 minutes (for unknown reasons) so the results are the best found in that period of time. The C2000 and C4000 graphs did not run due to lack of memory on the CM-5. (This is partially a problem of size and the memory allocator which allocates too much memory at a time.)

Results on Benchmark Instances

TABLE 8. Results on DIMACS benchmarks

Name	Runs (Fail)	Time Min	Time Avg (Std. Dev.)	Time Max	Solution Min	Solution Avg (Std. Dev.)	Solution Max
DSJC125.5.col	5	1199.22	4043.63 (2508.8)	8037.7	17	17 (0)	17
DSJC250.5.col	5	306.106	4358.12 (2789.37)	8013.02	29	29.2 (0.4472)	30
DSJC500.5.col	5	1172.94	4783.86 (2715.41)	7866.71	52	53 (0.7071)	54
DSJC1000.5.col	5	4171.87	5333.81 (1644.59)	8232	99	100 (0.7071)	101
C2000.5.col	5 (5)						
C4000.5.col	5 (5)						
R125.1.col	5	50	64.6 (22.2666)	104	5	5 (0)	5
R125.1c.col	5	60	85 (22.6716)	120	46	46 (0)	46
R125.5.col	5	31.47	32.986 (1.9874)	36.38	37	37 (0)	37
R250.1.col	5	22	22(0)	22	8	8(0)	8
R250.1c.col	5	110.7	278.16 (160.948)	505.8	64	64 (0)	64
R250.5.col	5	38.9	39.88 (0.5630)	40.3	66	66 (0)	66
DSJR500.1.col	5	24.5	26.64 (4.1198)	34	12	12 (0)	12
DSJR500.1c.col	5 (1)	1331.1	5767.67 (3703.33)	10139.6	85	85.25 (0.5)	86
DSJR500.5.col	5	85.6	90.5 (5.245)	96.2	128	128 (0)	128
R1000.1.col	5	49.4	49.88 (0.2683)	50	20	20 (0)	20
R1000.1c.col	5	259.4	3940 (5009.89)	10178	101	102.6 (1.1402)	104
R1000.5.col	5	210	215.9 (4.9548)	223.5	243	245.6 (1.5166	247

Name	Runs (Fail)	Time Min	Time Avg (Std. Dev.)	Time Max	Solution Min	Solution Avg (Std. Dev.)	Solution Max
`flat300_20_0.col`	5	236.8	274.3 (36.6314)	329.3	20	20 (0)	20
`flat300_26_0.col`	5	2721.3	6637.14 (3654.59)	10518.1	32	32.4 (0.5477)	33
`flat300_28_0.col`	5	454.6	1913.54 (1515.12)	3786.9	33	33 (0)	33
`flat1000_50_0.col`	5	7172.8	7792.66 (503.902)	8374.7	96	97 (0.7071)	98
`flat1000_60_0.col`	5	3766	6288.36 (1469.67)	7518.7	97	97.8 (0.8367)	99
`flat1000_76_0.col`	5 (1)	5697	6497.85 (664.774)	7100.3	99	99 (0)	99
`latin_square_10.col`	5 (1)	5266.4	6520.12 (1532.44)	8588.5	109	109.25 (0.5)	110
`le450_15a.col`	5	88	162.62 (75.522)	278.1	15	15 (0)	15
`le450_15b.col`	5	113.3	178.36 (45.0573)	226.1	15	15 (0)	15
`le450_15c.col`	5	1016.1	2229.61 (1114.42)	3828.8	16	16.6 (0.5477)	17
`le450_15d.col`	5	1303.5	2859.6 (1999.92)	5754.8	16	16.8 (0.4472)	17
`mulsol.i.1.col`	5	27	27.2 (0.4472)	28	49	49 (0)	49
`school1.col`	5	37.6	46.26 (7.8229)	55.2	14	14 (0)	14
`school1_nsh.col`	5	54.2	66.4 (9.3662)	76.4	14	14 (0)	14

Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin 53706
Current address: Department of Mathematics and Computer Science, Xavier University, Cincinnati, Ohio 45207-4441
E-mail address: lewan@xavier.xu.edu

Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin 53706
E-mail address: condon@cs.wisc.edu