Efficient Parallel Graph Coloring with Prioritization

Laxmikant V. Kale and Ben H. Richards and Terry D. Allen

Department of Computer Science,
University of Illinois at Urbana Champaign,
1304 W. Springfield Ave., Urbana IL-61801
{kale,richards,allen}@cs.uiuc.edu
http://charm.cs.uiuc.edu

Abstract. Graph coloring is an interesting problem that is intuitive and simple to formulate, yet difficult to solve efficiently. The applications of graph coloring are numerous, ranging from scheduling to solving linear systems. Because graph coloring is computationally intensive, a parallel algorithm is desirable. In this paper, we present a set of parallel graph coloring heuristics and describe their implementation in an environment supporting machine-independent parallel programming. The heuristics are intended to provide consistent, monotonically increasing speedups as the number of processors is increased. We present some performance results that demonstrate the effectiveness of our heuristics and the utility of our approach.

1 Introduction

The graph coloring problem involves assigning colors to vertices in a graph such that adjacent vertices have distinct colors. Various formulations of the graph coloring problem have been posed. The particular formulation we focus on is the following:

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E such that $E = \{(u, v) | u, v \epsilon V\}$. Given a set of colors $C = \{1, 2, \ldots, k\}$, find a mapping $\sigma : V \rightarrow C$ such that $\sigma(u) \neq \sigma(v)$ for each $(u, v) \epsilon E$.

This problem is known to be NP-complete [5]. Because it is intuitive and simple to formulate yet difficult to solve, this problem has drawn substantial interest over the years (e.g. the extensive work on the 4-Color problem [16] for planar graphs or maps). Graph coloring is an important problem with applications in register allocation, scheduling [1], the efficient computation of sparse Jacobian matrices [4], and the parallel solution of large sparse linear systems [9, 10, 15, 17, 18]. Finding a coloring or determining that no valid coloring exists often involves a large search which is computationally intensive. In this paper, we present an efficient parallel algorithm and its implementation for the graph coloring problem. Our algorithm handles the case in which no k-coloring is possible, as well as the case in which a coloring exists, which is the harder case to parallelize. Our parallel algorithm incorporates many heuristics, including those aimed at reducing the size of the search space and zeroing in on a solution quickly. It uses

a prioritization scheme to keep the parallel search focused on finding the first solution. The algorithm is implemented using Charm, which supports small-grain parallel objects with prioritized, dynamic load balancing. Charm also provides portability, which means the algorithm can run on any one of the many machines on which Charm is supported.

In Section 2, we present the search formulation and heuristics employed for solving the graph coloring problem. Section 3 provides details of the implementation in the Charm parallel programming environment. In Section 4, we evaluate the success of the heuristics, discuss the impact of grainsize control, and provide some performance results. Finally, we summarize the results of the research and offer some conclusions in Section 5.

2 The Search Tree and Heuristics

In order to achieve significant speedups in the solution of the graph coloring problem, it is important to employ a variety of heuristics. In this section, we first present the structure of the basic search algorithm, and then describe the heuristics implemented to improve the performance of the search and evaluation process.

2.1 The Search Tree

The basic structure of our algorithm is that of an exhaustive state-space search. Exhaustive search is essential in order to report that no k-coloring is possible for a particular graph. In contrast, many other heuristic methods (e.g. [2]) aim at quickly finding a k-coloring if one exists. If these heuristic methods fail, it is not clear whether a k-coloring does not exist, or simply was not found. Some other approaches (e.g. [3, 7]) do not start with a fixed number of colors, but instead try to produce a coloring with few colors, without any guarantees of optimality. The computation of a state-space search leads to a search tree in which each node of the tree represents a partially colored graph. The subtrees under each node represent states derived from the current state. The leaves of the tree represent states that cannot generate a new state — either a successful coloring, or a failed attempt.

Given a node[1] n of the search tree representing a partially colored graph G, one can generate its children as follows:

1. Select a vertex v that is not yet colored.
2. For each possible color for v, create a child node with a copy of G, updated to reflect the new color of v.

[1] To avoid confusion in our terminology, we use the word "node" exclusively to refer to nodes of the search tree, and the word "vertex" to refer to vertices of the graph being colored.

A straightforward parallel formulation of this computation is obtained by associating a (lightweight) process with each node. Generating children nodes then corresponds to firing child processes. In this simple formulation, all the children processes are independent and the parallelism is natural. However, the problem is far from simple (i.e. "embarrassingly parallel") because the issue of dynamic load balancing must be addressed. The problem is even more difficult to effectively parallelize if one is looking for a single solution (in contrast to looking for all solutions). This is due to the speculative nature of available parallelism: work done on one subtree is wasted if the solution is found in another subtree. In order to achieve good performance, it is necessary to utilize heuristic methods to control and direct the search.

2.2 Heuristics

Many heuristics are commonly used in sequential implementations of graph coloring [1]. It is easy to get good parallel speedups with poor heuristics, and very difficult to get good parallel speedups with good heuristics. This is because using a poor heuristic typically leads to a larger, more regular search space. Good heuristics lead to pruning of subtrees and generate significant variances in the sizes of subtrees of a given node, contributing to an irregular search space with potentially severe load imbalances.

In order to compete with highly effective sequential strategies, many sequential heuristics are incorporated in our parallel algorithm. These heuristics include the following: pre-coloring, vertex removal, impossibility testing, forced moves, variable ordering, value ordering, and detection of independent subgraphs. In this section, we give a brief overview of the heuristics employed.

Pre-coloring and Vertex Removal The first heuristic aims at reducing the redundancy in the search space, and is known as pre-coloring. Notice that colorings are equivalent under renaming. That is, given a particular successful coloring with k colors for the graph, there are $k!$ colorings which are essentially the same. The state-space search as described above will find all of these colorings in distinct parts of the search tree, which is clearly wasteful. We reduce this redundancy by identifying a large clique (a complete subgraph) of k or fewer vertices and color these vertices arbitrarily with different colors. Ideally, we should find the largest clique for this purpose. However, that itself is an NP-Complete problem. Instead, we choose to search for a clique with 3 vertices (a triangle), which can be done by a fast, polynomial algorithm. As a heuristic extension, we could search for larger cliques using a backtracking search within a fixed time bound. (This extension was not implemented.)

The second *pre-search* heuristic is to check the graph for removable vertices. A vertex can be removed from the graph if it has fewer neighbors than the total number of colors, since it will have a color available to it regardless of how its neighbors are eventually colored. Removed vertices are colored at the end of the computation in a LIFO manner, i.e. the first vertex removed will be the last to

be colored and replaced. This is necessary because the removal of vertices can cause other vertices to be removed which were not removable previously. For example, Figure 1 shows that the removal of vertex A can allow vertex B to be removed (the numbers next to vertices designate their assigned colors). As shown in the figure, when the remaining graph is successfully colored, node B will be added to the graph and colored before node A. (Both B and then A are guaranteed to have a color available for them when they are added back to the graph).

In addition to its use at the beginning of the search, this heuristic can also be used in the middle of the search, when some vertices have already been colored, in a slightly modified form: a vertex with U uncolored neighbors and with a total of C different colors used for its colored neighbors can be removed iff U+C is less than or equal to the total number of colors.

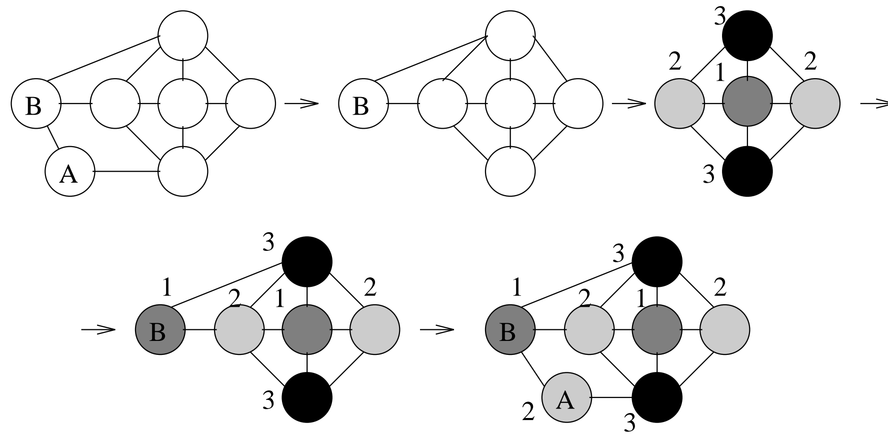


Fig. 1. The vertex removal process.

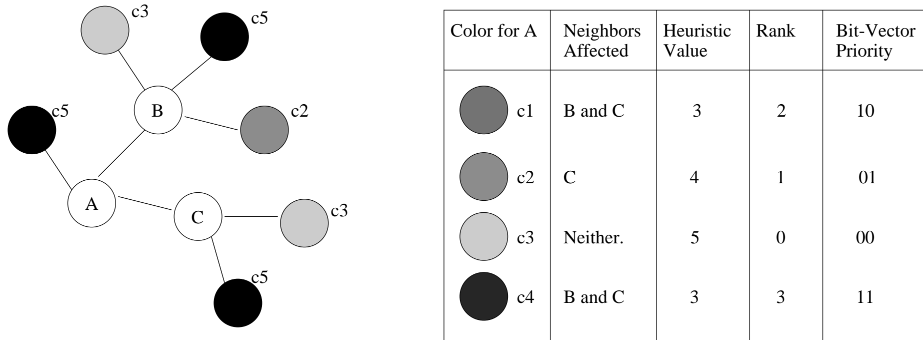
Impossibility Testing and Forced Moves During the search, it is possible that a vertex could be found that has no colors available to it, i.e. no color can be chosen that does not conflict with a neighbor. We eliminate this possibility while generating children states for a selected node, by keeping track of the number of available colors for each vertex. If, by coloring a particular vertex, one of its neighbors is reduced to zero available colors, we do not generate this state. This heuristic is known as *impossibility testing*.

This idea can be extended further, in that if a vertex is reduced to one available color, we know what color it will be assigned, and can determine the effects of that color assignment. The *forced move* heuristic is used to complete all state information that can be deduced from such a situation. When a selected vertex is colored and the availability of a neighbor is reduced to one, the color is

assigned. Of course, this can also cause another vertex to drop to availability of one or zero. The forced move heuristic is made recursive to handle this situation.

Variable and Value Ordering For each state, we must choose the variable (vertex) that will be assigned a color. Our variable ordering heuristic chooses the vertex with the fewest colors available to it. Selecting a vertex to color that has many colors available may lead to uncolorability of a more constrained vertex later in the search. This ordering is essentially the same as the Saturated Degree Ordering of the [1], except that we use this ordering in the context of a backtracking state-space search, instead of a greedy non-backtracking algorithm.

The value ordering heuristic determines which color should be chosen for a particular vertex under consideration. Since the color assignment to a vertex potentially affects the colors available to its neighbors, the goal is to choose a color for the vertex that will result in the state with the highest probability of containing a solution. For example, in Figure 2, assume that A, B, and C are not colored and we wish to select a color for A. Some choices of color for A ($c1$, $c2$ and $c4$) reduce the number of available colors for B and C and therefore constrain the solution space, whereas choosing color $c3$ for A does not constrain the available colors for B and C at all. It is clear that choosing $c3$ provides a higher likelihood of yielding a solution than the other choices. Generalizing from this, our value ordering heuristic explores first the coloring that leads to the smallest reduction in the number of available colors to the neighbors.



Color for A	Neighbors Affected	Heuristic Value	Rank	Bit-Vector Priority
c1	B and C	3	2	10
c2	C	4	1	01
c3	Neither.	5	0	00
c4	B and C	3	3	11

Fig. 2. Updating neighbor availabilities when a node is colored.

Detection of Independent Subgraphs The search space for coloring a graph of n vertices is potentially exponential in n. However, if the original graph is a disconnected one, dramatic reduction in the search space is possible. For example, if we notice that the graph given consists of two disconnected components, each with $n/2$ vertices, the search space is exponential only in $n/2$. In this case, one can look for a solution to each subgraph separately. If a solution for each is

found, then they can be composed into a solution for the whole graph. If there is no solution for one of the subgraphs, then there is no solution for the entire graph. The reduction in search time for unsuccessful searches is quite dramatic. For $n = 100$, the time goes from a^{100} down to a^{50}, for some a.

Even when the original graph is connected, one can still derive benefits based on the above observations. Consider a graph with an articulation point defined by vertex v. That is, if v is deleted, then the graph would partition into two independent subcomponents. In this graph, after the vertex v is colored, the two subgraphs do indeed become independent: colorings in one have no effect on colorings in the other. Again, it is enough to find a solution to each partition of the graph and compose them, or to prove that one of the partitions does not have a solution.

Although it takes some effort to detect whether a graph can be split into multiple independent partitions (this takes $O(V + E)$ time), it is worthwhile to carry out this check often, because the benefit of the potential reduction in search space is substantial.

To check for such partitioning, one considers the subgraph of the original graph that includes only the uncolored vertices. The key observation here is that once a node is colored, its effects are fully localized in potentially reducing the availability of colors to its neighbors. Its connections to its neighbors can then be ignored.

In terms of a search space, this heuristic generates an AND-OR tree instead of the pure OR tree that would result without this heuristic. The parallel processing of such trees becomes more complicated, and is discussed in Section 3.5.

3 Parallelization and Implementation

Having presented the structure of the algorithm and described the heuristics employed, we now provide details of the algorithm's parallel implementation. In this section, we first describe the programming environment in which the algorithm was implemented. We then describe the basic search process, how prioritization can be used to contain speculative parallelism, and how adaptive grainsize control can be used to overcome the problem of large grainsize variance. Finally, we discuss how we deal with independent subgraphs that arise during the computation.

3.1 Charm

Charm is an object based, message-driven, portable parallel programming system that runs on a variety of shared memory and distributed memory machines [11]. Charm, which is C-based, supports the notion of a *chare*, which is the encapsulation of an object with functions operating on that object. A chare is similar to a process or object in other programming paradigms. A chare can include private and public functions which can be invoked by objects on the same processor.

Chares can also include entry functions which are invoked asynchronously from a local or remote object.

Chares can be dynamically created, scheduled and distributed across processors. To pick a message for processing from the pool of available messages on a processor, Charm supports user-selectable, message scheduling strategies, including FIFO/LIFO and prioritized scheduling strategies. The programmer can specify any one of a number of dynamic load balancing strategies that permit the dynamic relocation of a chare after it is created but before it begins execution. Some of the load balancing strategies also take the priorities of messages into account. These strategies [23] not only balance load, but also ensure that higher priority tasks proceed before lower priority tasks across processors, and memory usage is balanced across processors. (While regular load balancing strategies try to ensure that no processor is idle when there is work available in the system, prioritized load balancing aims at ensuring no processor works on a low priority task while a high priority task is waiting somewhere in the system.) As we will see, prioritized load balancing is crucial to support our parallel graph coloring algorithm.

Additional features supported in Charm include *branch office chares* and information sharing abstractions. A branch office chare (BOC) is a "global" chare that has a representative branch on each processor. Branch office chares can be used to implement distributed data structures, distributed services, and static load balancing. Information sharing abstractions are also provided, including readonly variables, writeonce variables, accumulators, monotonic variables, and distributed tables.

3.2 The Node Process

The parallel state space search is encoded using the dynamically created objects (chares) of Charm. Each node of the search tree (as described in Section 2.1) corresponds to a chare. Each chare is given a partially colored graph adorned with some additional information useful for implementing the heuristics. Each chare must fire a number of children chares corresponding to multiple alternatives available at this point in the search tree. This section describes the data structures used and the procedure followed by this chare.

When a node is created, it receives a message from its parent containing a state, which represents a partial coloring of the graph. The state includes the following fields:

- an array *color*[] containing color information for each vertex
- the number of vertices not yet colored
- the number of vertices that have been removed
- an array containing the removed vertices themselves, and
- the depth of the node in the search tree.

There are also fields pertaining to handling independent subgraphs, but we will defer discussing these fields until Section 3.5.

$color[i]$ indicates the color of the ith vertex. If the vertex is not yet colored, this value is negative. To assist in the variable ordering heuristic, $color[i]$ holds $-1 * (number\ of\ available\ colors)$ for an uncolored vertex i. To specify the vertices that should not be colored (e.g. because they have been temporarily removed from the graph as explained in Section 2.2), $color[i]$ holds a special positive value greater than k.

Notice that the graph itself is not passed in this message. The graph is a large structure. Also, a large number of tree nodes (chares) are typically assigned to each processor. Therefore, passing the graph in a message to chares would consume excessive amounts of time and memory. Instead, we utilize the *readonly* abstraction of Charm, which allows sharing of information that is obtained only after the program begins execution and which does not change subsequently. Note that the graph data structure is immutable once initialized, and that all updates are made in the state representation discussed above. By making the graph a readonly variable, all chares can reference the graph without needing to send or receive it in a message.

Initiating the Search The main object on processor 0 initiates the search. First, the input file (containing the graph information and the number of colors to try) and the configuration file (containing the heuristics to apply and a value for the grainsize threshold) are read and used to initialize the readonly variables. Then, preprocessing of the graph is performed, consisting of a pre-coloring stage followed by a vertex removal stage. For each vertex, the vertex removal routine checks the color information array. If the vertex is uncolored then that vertex's uncolored neighbors are counted (not including any neighbors that were previously removed). If the count is less than the number k of possible colors, this vertex is added to the removed list. Because this action reduces the count for each of this vertex's neighbors, each vertex is re-examined to see if it can now be removed. This vertex-removal stage is repeated until no further removals are possible.

After pre-processing, the first state is initialized to a partially colored graph and a chare is fired to begin its evaluation. As each state is evaluated, many new states may be created. This process continues until a solution is found, or no states remain. The latter condition is detected by a quiescence detection algorithm provided by Charm [22].

The Node Algorithm When a chare begins its execution with a given state, it carries out the following steps:

1. Use the variable ordering heuristic to select a vertex to color. With the information maintained in the color array, this can be done efficiently. We simply select the vertex i with the smallest number of available colors: $|color[i]| < |color[j]|$ for all j, with $color[j] < 0$.
2. Identify the possible colors for vertex i. This is done by identifying i's neighbors using the static graph information and then checking each neighbor's color status in the given state.

3. Create a new state for each possible color, and copy the given state into each. Also, set $color[i]$ to a different feasible color in each new state. Update $color[j]$ for all uncolored neighbors j of i to reflect any changes in the number of colors available to them.
4. Carry out forced moves and impossibility testing on each child state separately. The child states shown to be uncolorable are discarded. Also, if any child state represents a fully colored graph, the solution is reported.
5. If the search is not completed, rank the undiscarded children states using the value ordering heuristic. For each child state, its heuristic value H is computed as
   $H = \sum |color[j]|$ for all j such that $(i, j) \epsilon E$ and $color[j] < 0$
   (i.e. H is the sum of colors available to each neighbor of i). The children are ranked in decreasing order of H, as higher H represents higher likelihood of a solution being found in the subtree associated with that child (see Figure 2).
6. Fire a chare for each child node, with a priority assigned using the scheme described below.

3.3 Speculative Parallelism and Prioritization

The search process as defined so far leads to *speculative parallelism*. If we fire a chare for each of two subtrees of a node (which are executed in parallel by two processors), the work done by one processor is wasted when the solution is found in the other subtree. This speculative nature of parallelism leads to two problems for parallel performance. First, the speedups are inconsistent from run to run, as the number of nodes evaluated varies depending on the order in which they are processed. Second, the speedups do not increase monotonically with the number of processors. Adding a processor may sometimes lead to a slowdown if the new processor ends up searching a futile subtree and creating more useless work for other processors. Our objective then is to obtain consistent and monotonically increasing speedups.

This problem was addressed by our earlier work [12, 20, 13] in the context of other search problems. The intuitive idea behind this scheme is that if a parallel algorithm approximates the sequence in which nodes of the search tree were evaluated by the sequential algorithm, the speculative loss will be minimized and consistent and monotonically increasing speedups can be obtained. To this end, the idea of bit-vector priorities was developed[2]. Each node in the tree is assigned a priority which is a bit-vector of an arbitrary length. Priorities are compared with a lexicographic (dictionary) ordering with smaller lexicographic values indicating higher priorities. The priority of the root node is an empty bit vector. Given the priority of a parent node, the priority of each child is obtained by appending bits corresponding to its rank to the parent's priority. If there are m children, the number of bits appended is $\lceil \log m \rceil$. The appended bits are the binary representation of the child's rank. Figure 3 illustrates this scheme.

[2] This scheme can be considered an optimization of a scheme proposed earlier by Li and Wah for constraining speculative parallelism in branch-and-bound computations [14]

This scheme has an important property called the prefix property: despite the fact that the number of bits appended varies from node to node, it guarantees that no two nodes will have identical bit vectors. More importantly, the priority of every node in the left subtree will be lexicographically smaller than each node in the right subtrees. Thus, it provides an efficient left-to-right ordering of tree nodes. Further elaboration of this strategy can be found in [21]. If the parallel system adheres to these priorities, the search space is examined in a characteristic "broom-sweep" manner from left-to-right, leading to a memory-efficient, consistent and monotonic search process.

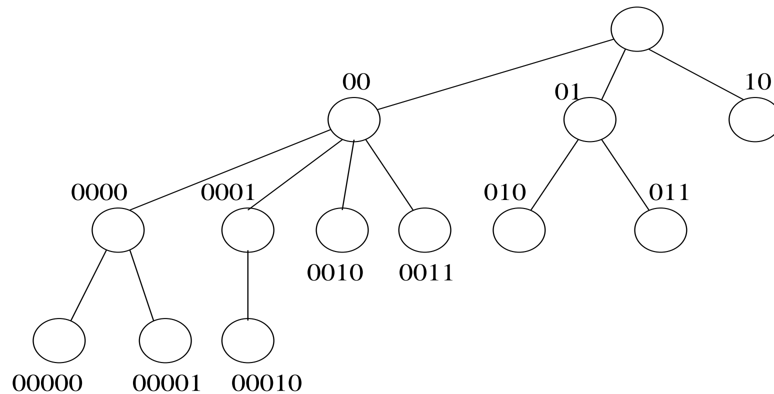


Fig. 3. Priority assignments to children in the search tree.

Charm allows bit-vector priorities to be assigned to messages to prioritize their scheduling for execution. The prioritized load balancing strategy provided in Charm [23] closely adheres to priorities even on large distributed memory machines. Consistency and monotonicity are ensured by using bit-vector priorities. Furthermore, using the heuristic values to rank the children before assigning their priorities ensures that the search is globally focused on the most promising parts of the search tree.

3.4 Adaptive Grainsize Control

There is an overhead associated with creation of a chare. This includes message passing overhead, scheduling overhead, and load balancing overhead. For an effective parallelization, the computation associated with each chare must be sufficiently large in relation to this overhead. As a rule-of-thumb, the computations should be about 10-20 times more than the overhead. (This overhead in Charm is about a few hundreds of microseconds on most of today's machines.) The work associated with a single node of the search tree is quite small, so we need to aggregate many nodes into a single chare.

An effective means of controlling the grainsize is provided through the notion of a cutoff. For every node given to a chare, it estimates how much work is involved in completing the search beneath that node (i.e. fully exploring the search tree under it). If the estimate is below a certain threshold, the computation is carried out by a sequential procedure instead of firing the children nodes as chares. Estimating the amount of work is hard in general. A reasonable heuristic is provided by the number of nodes that remain to be colored. The more nodes that remain, the more work that is likely to be required. The grainsize threshold can be set in terms of this number. This was the heuristic we employed first for controlling the grainsize. Notice that at the higher levels of the tree it creates finer grainsizes — one node per chare. This in itself is not a problem, as long as the average grainsize is sufficiently large, which is easy to ensure as the number of leaves is usually larger than the number of internal nodes.

This grainsize control method works adequately for relatively regular trees and/or a small number of processors. Beyond that, it runs into the following problem: there is a significant variation in the size of subtrees produced by a given grainsize threshold. This means that some grains are substantially larger in size compared to the average. On a large number of processors, such grains would hold back the computation until these grains are evaluated on some of the processors. As shown later in the performance section, grainsizes larger than 40 times the average grainsize were observed in this situation. Although smaller grainsizes are not a problem as long as the average is adequate, larger grainsizes do create problems by lengthening the critical path and impacting the load balancing strategy negatively. This is further exacerbated because one cannot predict which grains will be larger, and the larger grains might be left until the end when there is no other work to keep other processors busy.

To handle these situations, we developed another heuristic that adaptively adjusts the grainsize. In this heuristic, every chare starts evaluating its given state as if it were going to complete the whole search tree beneath it. Instead of using a recursive procedure for this, it uses an explicit stack. It also keeps track of the amount of work (measured in terms of the number of node expansions) it has performed at any point in time. The heuristic uses a threshold defined in terms of a number of node expansions. When the number of node expansions exceeds the threshold, the chare selects a predetermined number of states from the bottom of this stack, copies them into messages and fires a new chare for each state. At this point, it also resets the amount of work done to zero, so that the count can begin anew. The chare repeats this process until all the nodes on the stack are exhausted, repeatedly shedding off a few nodes as new chares as it does more work. (A variant of this strategy that we experimented with splits all the available states on the stack into multiple groups, and fires a chare for each group of states. In this formulation, each chare is given a number of states, instead of just one, with which to begin.) This heuristic ensures that the grainsize of a chare does not grow in an unbounded manner, because work is continually shed from a large subtree.

This idea of adaptive grainsize control was defined and tested on simple

divide-and-conquer programs in our earlier work. It is also related to notions defined by [6, 24].

3.5 Dealing with independent subgraphs

For simplicity, we have omitted any details pertaining to the multiple independent subgraph heuristics in the discussion so far. We will now describe these heuristics.

When a child chare notices that the graph represented by the states can be partitioned into independent subgraphs after a particular coloring (by running a simple connected components algorithm), it becomes an AND chare. An AND chare fires the independent subgraphs as separate chares. Each chare is given a state in which the nodes not belonging to it are marked in the *color*[] array by a special value. The state message also contains the identifier of the parent AND chare. The chares in each subtree use this identifier to report a solution to the AND chare if and when they find one. When the AND chare receives a message indicating a solution to one of its subgraphs, it must store the solution and terminate search for any more solutions for this subgraph. The latter is accomplished by sending a message down the tree. For this purpose, all chares maintain a list of their children chares, and send a termination message to them when their parent requests termination. The termination messages may never fulfill their purpose if every object processes them only after spawning its children. To avoid this infinite "kill-chasing", we require all chares under the subtrees of an AND node to request permission from their parents before spawning their children. If the parent has not received a termination message when the request for permission is received, it will grant the permission and send this message down to the children. This ensures that the termination messages can catch up with the node expansions. The AND node waits for a solution for each of the subgraphs it has spawned as a chare, and when each such solution is returned, it composes them into a solution for itself. The AND node may itself be under another AND node's subtree, indicating further partitioning of partitioned graphs. In that case, the AND node will send its solution to its ancestor AND node[3].

The AND node also needs to know when one of its subgraphs does not have a solution. In that case, it needs to terminate the search on the other subgraphs. To detect this condition, every node in the subtree of an AND node reports the

[3] An alternative scheme for terminating search on futile subtrees is possible. In this scheme, each child node of an AND chare is assigned a unique index. Every processor maintains the status of all such nodes in a table. When an AND chare wishes to terminate a subtree, it broadcasts a message to all processors which then update the status in a table. Instead of requesting permission from their parent chares, individual chares simply check the status of their subtree in the table directly. AND chares which are descendents of other AND chares must still be terminated using messages. The tradeoff between the two schemes depends on the number of nodes for which status must be maintained. If the number is large, the cost of broadcast outweighs the cost of sending permission messages, and the scheme described earlier is better.

failure to find a solution in its subtree to its parent. This way, the AND node eventually finds out about such failures from its children. Prioritization of AND-OR trees is a more complex topic [21]. For this algorithm, we allocated identical priorities to the children of the AND chare.

4 Performance Evaluation

We now discuss the impact of the heuristics and the importance of adaptive grainsize control, as determined by our performance evaluation. The algorithm is programmed in Charm (which runs on many shared and private-memory machines). Here we report some of the performance data on the Encore Multimax with 8 processors in a shared memory configuration, and the nCUBE/2 with up to 256 processors. To generate the input graphs, we used a random graph generator which takes the number of vertices and the number of edges as parameters. Each generated graph, for a given value of k, may or may not have a solution. Only some of the example graphs in the original study [19] are shown here. The characteristics of these graphs are shown in Table 1.

File	Nodes	Edges	Colors	Solution
Example 4	300	1626	5	Yes
Example 7	450	2451	5	Yes
Example 8	600	2338	3	No
Example 9	301	4274	5	No

Table 1. Input file summary.

Our first set of experiments focused on evaluating the utility of various heuristics. We found that the value ordering heuristic gave dramatic improvements in performance for colorable graphs. The variable ordering heuristic was particularly helpful for uncolorable graphs. Typically, without these heuristics the search would not complete in a reasonable amount of time on most of our test graphs. The forced moves and impossibility testing heuristic also led to dramatic improvements. As expected, pre-coloring reduced execution time for uncolorable graphs. For example, the time for an uncolorable graph with 150 vertices and 854 edges with 4 colors decreased from 1289 seconds to 101 seconds on a single processor run. The vertex removal heuristic led to less dramatic, but definite performance improvements.

One of the interesting aspects of our experimentation was the extraordinary success of the heuristics we use for colorable graphs. In most cases, a coloring would be found in a short time — too short to be worthy of parallel processing. As the focus in this paper is on parallel computing, we have included cases where the search for a solution was not as fast. (Of course, it is possible that the randomized generation of graphs leads to easily colorable graphs, whereas

real-life applications such as those mentioned in Section 1, are more difficult. Our search program will be more useful the harder the graphs are to color.)

Table 2 shows the performance on a colorable graph (Example 4) on the Multimax. All the speedups are with respect to the best sequential performance obtained with the highest grainsize leading to a single chare doing all the work. On eight processors, the speedup is approximately 7.2.

Processors	Chares	Execution Time (sec)	Time per Chare (ms)
1	1	1440	1440177
2	1463	602	823
4	2138	311	582
8	2834	199	562

Table 2. Parallel speed-ups on a successful search on Multimax, for the graph of Example 4.

Table 3 shows results obtained on the nCUBE/2. As this machine is large, we show results only from 16-128 processors here. Runs with fewer processors would require much longer times. Results from multiple grainsize settings are shown. For example, a grainsize setting of 540 means that the sequential algorithm was used when the number of uncolored vertices fell below 540 (out of a total of 600 vertices). Performance with a grainsize setting of 570 (leading to a mean grain size of 59 milliseconds). seems to be the best, and the speedups were also reasonable. For example, on 128 processors the program was about 6 times faster than on 16 processors.

GS setting	540	570	580	585
Processors	**Execution Time (ms)**			
16	100907	94093	99954	193217
32	52187	46182	62686	145633
64	28182	26500	47837	129824
128	15648	16147	30022	96426
Statistic	**Statistic Value (across processors)**			
mean GS (ms)	30	59	436	2980
std. dev (ms)	11	82	991	6599
min-max GS (ms)	9-93	14-2411	17-8342	17-39689
Chares	50552	24075	2997	434

Table 3. Example 8 results on nCUBE/2.

One interesting aspect brought out by the performance data was that the grainsize varied dramatically around the mean. This can be seen in the bot-

tom part of Table 3. For example, with the grainsize setting of 570, the grainsize varies between 14 milliseconds and 2.4 seconds. That such variation leads to performance problems was verified using the next experiment. For this experiment (Example 9), another uncolorable graph was used. Figure 4 depicts the performance using the *Projections* performance analysis tool associated with Charm. The plot shows the percent utilization across 64 processors as it varies over time. One can see that the utilization starts to drop from nearly 100 percent to substantially lower values at about 33 seconds into the computation. Further exploration via Projections confirms that this is due to a few processors executing large grain computations that remain. Even the average grainsize on individual processors varied between 100 and 173 milliseconds.

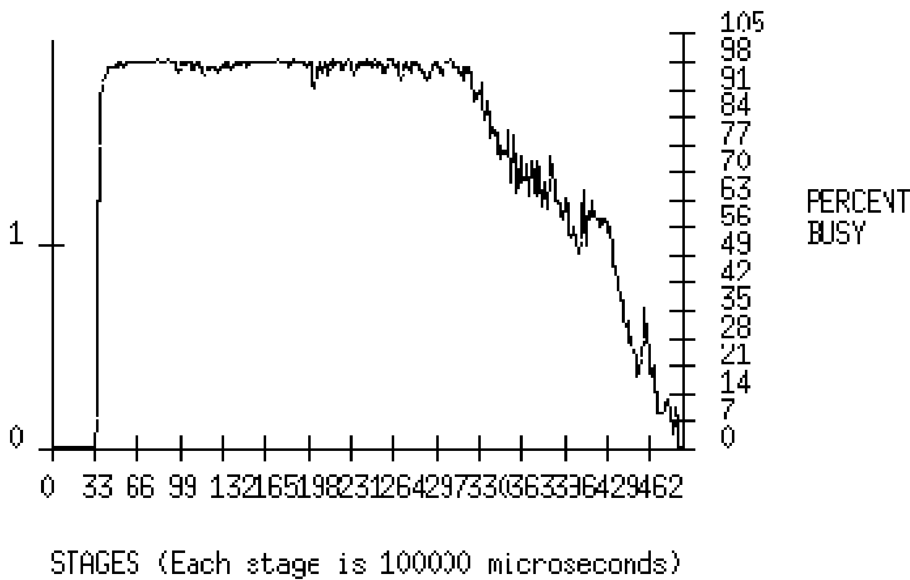


Fig. 4. Projections view of overall performance for Example 9 with average grainsize of 138 msec.

Using the adaptive grainsize control method described in Section 3.4, the same program was run again. This led to a substantial reduction in the variance of grainsize and consequently improved the performance. The Projections utilization in Figure 5 illustrates this gain. The utilization is seen to remain close to 100 percent almost to the end of the entire computation, and the overall computation time decreases by about 15 percent.

Figure 6 summarizes the parallel speedups obtained on the nCUBE/2, nor-

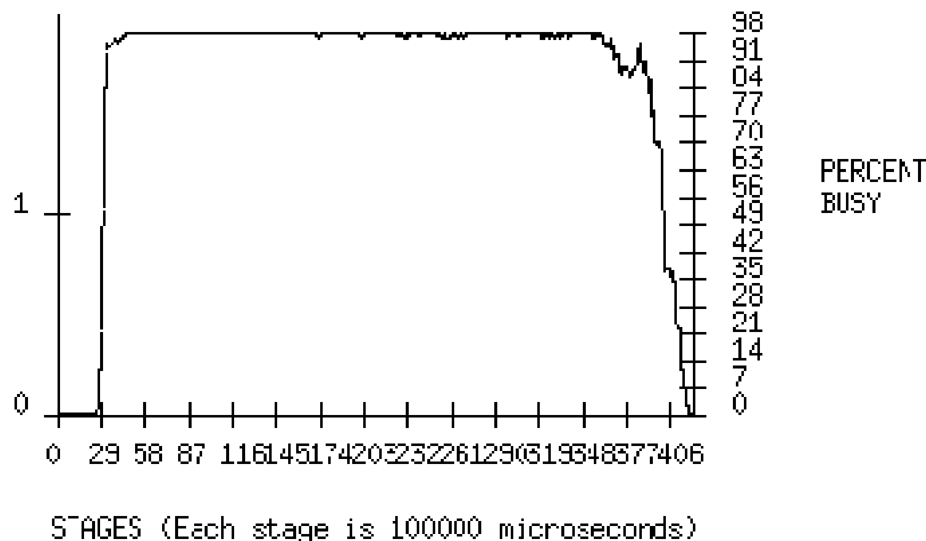


Fig. 5. Projections view of overall performance for Example 9 (stack method) with average grainsize of 107 msec.

malized to $N = 16$ processors. The speedups for Examples 8 and 9 (both of which do not have a valid coloring) are close to linear. Note that the speedup between 32 and 64 processors for Example 7 (which does have a solution) is superlinear. As the number of processors is increased, it is likely that one processor may pick a node to expand that quickly leads to a solution. This node may be explored before the nodes to its left are made available by the prioritized load balancing strategy. The prioritization scheme does not preclude such superlinearity. It does however ensure that out-of-priority-order execution is bounded, meaning that substantially sublinear speedups are unlikely.

The performance data illustrates that despite the somewhat complex heuristics (in that they complicate the parallelization), our algorithm is able to obtain good speedups on shared memory as well as private memory machines.

5 Summary and Discussion

We presented a parallel graph coloring algorithm and its portable parallel implementation. The algorithm is formulated as a state-space search based on heuristics. This enables the algorithm to either produce a coloring or to eventually report that no coloring is possible. Heuristics such as pre-coloring, recursive node removal, and variable ordering (i.e. selection of the most constrained vertex to color) were implemented to reduce the size of the state space. A value

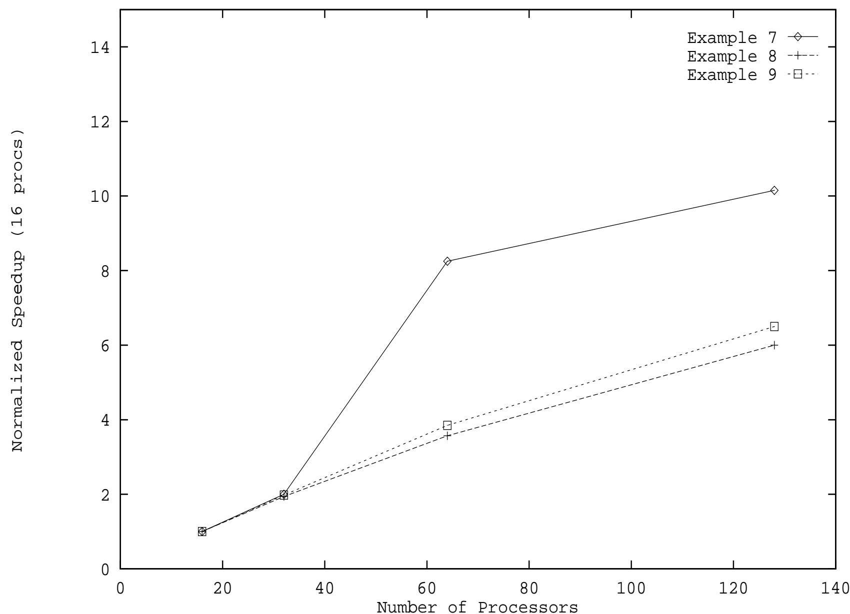


Fig. 6. Summary of parallel speedups on nCUBE/2.

ordering heuristic that chooses the least constraining color to focus on first was also implemented, and supported using bit-vector priorities. Another contribution of this research is the parallelization of a graph partitioning heuristic, which has the potential for reducing the search space dramatically. The incorporation of this heuristic turns the search tree into an AND/OR tree. A scheme for coordinating the exploration of this AND/OR tree while still maintaining the focus via priorities was presented. A further contribution of this paper is the application of the adaptive grainsize control method, which was demonstrated to lead to significant performance improvements.

We are not aware of any other parallel state-space search algorithm implemented for graph coloring which uses prioritization to obtain speedups that are consistent from run to run and monotonically increase with additional processors. (For example, [8] presents an asynchronous parallel algorithm that produces a coloring of a given graph, aiming at using few colors, but without any guarantee that the number of colors used is the smallest possible.) It is possible to further enhance the performance of our algorithm. In particular, the value ordering (or color selection) heuristic can be further refined to zero in on solutions faster. Also, it is possible to pursue partitioning of graphs more aggressively by modifying the value ordering heuristic to aim at coloring those vertices that will help partition the graph — such as the articulation points. One could also identify a minimum cut through the graph, and decide to color the vertices on the

cut first (or give them higher priority, right after the forced moved), in order to force a partitioning. If the cut were a clique, one could a result used in [2] to avoid any backtracking search for coloring the vertices on the cut itself. With such enhancements, we expect this algorithm to be one of the best methods for graph coloring, so that it can be used in applications including scheduling, register allocation, and linear systems solving.

The ("Feasibility") formulation pursued in this paper aims at finding a k-coloring for a given graph, for a given, fixed, value of k. An alternate ("Optimality") formulation found often in literature requires one to find a k-coloring for a given graph, with the smallest possible value of k. The feasibility formulation is useful in register allocation, where the number of registers is fixed. However, when the optimality formulation is required, our algorithm can be modified to obtain the smallest-k coloring, as follows: To begin with, set k to one more than the degree of the highest degree vertex in the graph. In each iteration, run the feasibility algorithm. If a k coloring is found, decrement k and start the next iteration. The coloring found before the final unsuccessful iteration is the optical coloring. To speed the search further, multiple consecutive iterations can be overlapped in a prioritized manner using a scheme developed it for parallel iteration-overlapped IDA* [13]. As mentioned earlier, the algorithm finds a k-coloring rather quickly when one exists. So, the parallel prowess of the algorithm can be demonstrated mainly on non k-colorable graphs. The optimality formulation requires at least one non-colorable search, where the algorithm is therefore quite useful. In addition, of course, graphs that are colorable, but are hard to find a coloring for, are also good targets for our algorithm.

The graph coloring program written in Charm, along with the random graph generation program and benchmarks, is available on the world-wide web from `http://charm.cs.uiuc.edu`.

References

1. D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
2. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
3. M. Chams, A. Hertz, and D. de Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research, Vol. 32, :*, pages 260–266, 1987.
4. T.F. Coleman and J.J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187–209, 1983.
5. M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, 1979.
6. R. Halstead. Parallel symbolic computing. *IEEE Computer*, pages 35–43, August 1986.
7. A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing, Vol. 39, :*, pages 345–351, 1987.

8. Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Statist. Comput.*, 14(3):654–669, May 1993.
9. M.T. Jones and P.E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20:753–773, 1994.
10. R.K. Gjertsen Jr. Parallel graph coloring heuristics. Technical report, University of Illinois at Urbana-Champaign, 1994. Master's Thesis, Dept. of Computer Sceince.
11. L. V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, 1990.
12. L.V. Kale. Parallel problem solving. In Vipin Kumar, P. S. Gopolakrishnan, and L. N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 146–181. Springer-Verlag, 1989.
13. L.V. Kale, B. Ramkumar, V. Saletore, and A.B. Sinha. Prioritization in parallel symbolic computing. *Lecture Notes in Computer Science*, 748:12–41, 1993.
14. G. J. Li and B.W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. In *IEEE Transactions on Computers*, pages 568–573, June 1986.
15. R.G. Melhem and V.S. Ramarao. Multicolor reorderings of sparse matrices resulting from irregular grids. *ACM Transactions on Mathematical Software*, 14:117–138, 1988.
16. O. Ore. *The Four-Color Problem.* Academic Press, New York, 1967.
17. J.M. Ortega. Orderings for conjugate gradient preconditionings. *SIAM Journal on Optimization*, 1:565–582, 1991.
18. C. Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel processors. *SIAM Journal on Scientific and Statistical Computing*, 13:194–226, 1992.
19. B. Richards. Parallel graph coloring with Charm. Technical report, University of Illinois at Urbana-Champaign, 1994. Master's Thesis, Dept. of Computer Science.
20. V. Saletore and L.V. Kale. Consistent linear speedups for a first solution in parallel state-space search. In *Proceedings of the AAAI*, pages 227–233, August 1990.
21. V.A. Saletore. Machine independent parallel execution of speculative computations. Technical report, University of Illinois, Urbana, Illinois, 1990. PhD Thesis, Dept. of Computer Sceince.
22. A. Sinha, L.V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, 1993.
23. Amitabh Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993.
24. M. Wu and W. Shu. A dynamic program partitioning strategy on distributed memory systems. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 551–552, 1990.

This article was processed using the LaTeX macro package with LLNCS style