# Final Report:
# Large Scale Parallel Graph Coloring

Laith Barakat and Hyun Duk Cho and Garrett Kiel and Phil Miller

May 12, 2009

## 1 Introduction

Graph coloring is the mathematical abstraction of many problems that require resource allocation without interference. It appears in fields as diverse as scheduling rooms, assigning variables to registers in compilers, and setting flight levels for air traffic paths. The abstraction is that uses of a resource are vertices in a graph, and interference between uses is represented by edges between corresponding vertices. Each vertex is assigned a color indicating what resource will serve the corresponding use. The problem, then, is to find an assignment that either minimizes the resources used, or fits the problem in question into available resources (reporting impossibility as necessary).

The minimization problem (or alternately, determining if a number of colors is the minimum necessary) is NP-Complete. The problem of determining whether a given coloring exists

Find the minimum number of colors necessary to color a graph (the chromatic number $\chi(G)$). This means finding two things: the assignment of colors to vertices that proves such a coloring possible, and failed search for a smaller coloring to show that this is the minimum.

## 2 Sequential Implementation

The sequential implementation involves recursively searching every possible coloring of the graph until a solution is found, or no solution is found. It starts by selecting an uncolored node, and coloring that node with a color that it is not neighbored by. This step recurses until a solution is found, or until it is unable to color a node. If a node cannot be colored by any color, the recursion backtracks to a previously colored node and tries coloring it a

different color. To find the optimal coloring, we would run a search on the graph, giving it 1 color to work with. If the graph can't be colored with that number of colors, we increment the number of colors it is allowed to choose from, and try searching again. We repeat this until we successfully color a graph.

## 3 Implementation

In our implementation, we are using Charm++ to parallelize the algorithm. We are using it because the dynamic structure of the problem makes it a much better fit for Charm++ than the other technologies. This is because the creation of a new task and communication of its success or failure is a very natural pattern for Charm++. Additionally, much earlier work demonstrated good results for this problem using the predecessor to Charm++, the Chare Kernel [**?**].

However, while trying to parallelize the algorithm, we ran into a few challenges. One of our first attempts at parallelization spawned work too aggressively, causing us to run out of memory in trying to solve even some of the smallest problems. This memory consumption was caused by spawning a new node for every possible coloring of every vertex, and sending a large message of all the data the new node would need to do the search to the newly created node. Another significant challenge we've had is coming up with an effective way to parallelize the algorithm so that we would see some real performance gains. Another strategy we tried resulted in minimal performance gains on 2 or 3 processors, and no gains on more than 3 processors. This strategy involved spawning a new node off after some arbitrary number of steps with a different coloring to continue searching with. The problem with that strategy is that we don't know if we're spawning a new worker off at a useful point in the search or not, possibly spawning off lots of nodes at the very end of the search, and not very many early in the search.

## 4 Scaling Experiments and Analysis

How well does the parallel code scale (what kind of speedup do you observe/expect on hundreds of processors? Hundreds of thousands?).

What are the bottlenecks?

What results do you have so far?

What experiments are you planning to run?

What do you expect your results to be?