Preliminary Design: Large Scale Parallel Graph Coloring

Laith Barakata and Jason Cho and Garrett Kiel and Phil Miller

February 24, 2009

1 Problem Description and Basic Algorithm

We intend to solve the vertex coloring problem for very large graphs. The basic algorithm for this fixes the colors of two adjacent nodes and then recursively searches through possible colorings of the rest of the graph. There are a number of heuristics and tricks used to make this NP-hard problem more tractable. The primary parts of the problem to which heuristics are applied are the order in which nodes are colored, and the order in which colors are tried for each node. One of the tricks is to notice when the currently colored nodes form a cut of the graph, and the subsequently color the resultant partitions independently. This is also useful in a parallel implementation, because it allows simultaneous exploration of both partitions.

2 Parallel Strategy

This work follows in the footsteps of a 1995 paper by Kale et al. on small-scale parallel graph coloring. As it notes, exploring the search space consistently with a good sequential heuristic is important to achieving parallel speedup. With priorities placed on the various possibilities to explore, we then expose the possibilities in priority order to the numerous processing elements available in the system.

Subgraph coloring problems resulting from partitioning will each be considered at a priority consistent with their likelihood of contributing to a solution to the overall coloring problem. One avenue not apparently explored in the earlier paper is that the progress in exploring colorings for one side of a partition should influence not only the priority of its own descendants in the search tree, but also the priority of its partner(s), since they cannot form a complete coloring in isolation.

Two elements of the data structures we intend to use contribute to our ability to scale as intended. The first is the use of a shared array construct to expose the graph structure to all PEs without requiring that it fit in memory on any one of them. This is discussed further in section 3. The other component is

the use of functional data structures to represent partial colorings, rather than duplicating this data for each grain of work enqueued in the system.

3 Predicted Scaling

Because of the nature of this problem, there is no simple scaling equation that can be used to predict what problem sizes will be tractable on a computer with a given number of PEs. However, there are some outer bounds to consider. Suppose the graph's representation in adjacency list form consumes 16 bytes per edge. If compute nodes have 2 GiB of main memory (and we ignore all other storage needs), this means that a single node could hold 134 million edges. A partial coloring will occupy (pessemistically) 16 bytes per vertex, and there could be several independent partial colorings (such that data isn't shared) in a single compute node. At 512 bytes per vertex (about 32 colorings), 2GiB could hold partial colorings of 2 million vertices. Since these, and other data structures, such as work queues, must share main memory, the overall problem size will end up somewhat lower. The primary limit we expect to encounter is in exposing large-scale parallelism in the problem space.

4 Project Planning

Initials indicate assignment, with dates being deadlines.

1. Initial sequential implementation in Charm++ (LJ: 3/17)
   (a) Architecture (PJL: 3/6)
   (b) Input data structure definition (PLJ: 3/6)
2. Heuristics (G: 3/17)
3. Sample Inputs (P: 3/17)
4. Spawning grains of work
5. Partitioning and independent subgraph coloring
6. Functional data structures for partial colorings within compute nodes
7. Performance analysis and tuning
8. Presentation of results (4/24)