Graph Coloring Using State Space Search

Sandeep Dasgupta* Tanmay Gangwani† Mengjia Yan‡ Novi Singh §

October 29, 2014

1 Heuristics for State-Space Search

The following are the heuristics that we intend to implement in our parallel algorithm. Some of the ideas are borrowed from [2].

- Pre-Coloring and Vertex Removal
  - Coloring a clique : By searching for a clique of a given size and coloring it arbitrarily we can reduce the number of states that need to be searched as the algorithm do not have to color the vertices involved in a clique in different part of the state space search. We would consider a clique size of 3 since larger cliques can be formed from combining size-3 cliques.
  - Removal of vertices with fewer neighbors than k, the number of colors: Any vertex fewer than k neighbors is guaranteed to have a color available to it. These are removed at the starting chare and placed in a local stack for latter LIFO merging. Also, at every stage of the state space search, a local algorithm tries to find if the number of colors available to a vertex (Total number of colors - number of different colors used to color the neighboring colored vertices) exceeds the number of its uncolored vertices. If that is the case, that vertex is removed since it is guaranteed to have a color available to it, and is stored locally.
- Detection of Independent Subgraphs
  This is a simple heuristic which allow us to color smaller graphs independently. However, in the case where a graph is connected we can find an articulation point (a vertex that once removed would split the graph into two independent subcomponents), color that vertex, and then color the subgraphs. This heuristic can be applied recursively to obtain any number of independent graphs that can be colored and then composed. This kind of a splitting generates an AND-OR space search tree instead of just an OR tree. The child chares that are coloring the subgraph maintain proxies to their parents. If a child finds a solution then it reports to its parents and the message continues up the tree. If a solution is not found then the child will report up to its parent and the important point is that the parent needs to recursively kill all of the child chares that it spawned. This is the important distinction about the AND-OR tree, namely that a failure at an AND node (where we split the graph at an articulation point) leads to failure for all child nodes of the AND node. Furthermore, to speed up killing child chares, all chares wishing to spawn a child must ask their parent whether or not they can spawn a child, thus reducing the potential depth a kill command might have to travel down to complete.

*Electronic address: sdasgup3@illinois.edu
†Electronic address: gangwan2@illinois.edu
‡Electronic address: mengjia@illinois.edu
§Electronic address: novi.singh94@gmail.com

- Impossibility Testing and Forced Moves
  - Impossibility testing: If we find that by coloring a vertex that one of its neighbors is reduced to 0 possible colors then we dont generate that state. This helps to reduce total chare count.
  - Forced Move: If by coloring something we find that one of its neighbors is now forced to be a certain color because it only has one color available then we perform that move. However, this might then lead to another vertex not having any possible colors left, which means that the forced move heuristic has to be recursive in nature.
- Value Ordering and Prioritization: When a chare selects the most constraint vertex to color, and spawns off m child chares, where m is the number of possible colors for the vertex, then we assign bitvector priorities to the child chares. The chare in which the assigned color least impacts the color possibilities of the neighboring vertices is given the highest priority. In this way, we sweep the state space search in a left to right manner.
- Adaptive grain size control for Chare creation: By grain size we mean the number of vertices colored at each state of the state space search tree. If each state selects 1 vertex (most constraint) and spawns k children (k = color possibilities for that vertex), then the number of spawned chares grows as:

$$1 + k + k^2 + k^3 + k^4 + \ldots + k^d = O(K^{d+1} - 1).$$

  where d is depth of the state space tree.

  Note that, this is not an exact estimate since the branching factor would reduce (be much less than k as we go down the state space tree since the coloring possibilities reduce). Let the grain size be G. This means that each node would select the g most constraint vertices and then spawn k^G children. Total chares spawned in this case is: Let $a = \frac{d}{G}$

$$1 + k^G + k^{2G} + k^{3G} + \ldots + k^{aG} = O(k^{aG+1-G}) = O(K^{d+1-G})$$

  Comparing the two, we get a reduction in number of chares spawned by a factor of k^G for a grain size of G. A more aggressive way of controlling grain size, which would lead to a less optimal solution, but with much less number of chares: At each level of state space, a chare colors (permanent coloring) G vertices, then finds the next most constraint vertex, and spawns off k children. The permanent coloring of G vertices could be done using some sequential graph-coloring algorithm. The number of chares in this case grows as: Let m=d/G, d = original depth of state space tree, G = grain size

$$1 + k + k^2 + k^3 + \ldots + k^m = O(k^{\frac{d}{G}})$$

  Here, we get a reduction of a factor of $k^{d-\frac{d}{G}}$ for a grain size of G. This is much more than the previous reduction of k^G, and hence we would find a solution much faster, using much less memory. But the downside is that we will not get the most optimal coloring.

2 Code Status[1]

- Input graph: We have a python interface to the main chare. A random graph generated by python is fed to the main chare and stored as an adjacency list.

[1] Find the .ci interface file at https://github.com/sdasgup3/ParallelSudoku/blob/master/Source/graphColor.ci

- Conservative Estimate: A linear time sequential algorithm is run on the input graph to conservatively estimate the chromatic number of the graph. This algorithm simply assigns the lowest possible color to a newly discovered vertex in the graph.
- Parallel graph coloring: Starting from the conservative estimate, say X, we run repeatedly the state space search algorithm with chromatic numbers X, X-1, X-2 and so on. If Y is the chromatic number in the first iteration not to yield a full coloring solution to the graph, then Y+1 is the optimal coloring. This is reported and verified by the main chare. In the present implementation, a child chare reports its findings back to the parent. The results are percolated all the way to the main chare, which then declares if the input graph is colorable with the chromatic number selected for the running iteration. We use an SDAG for the parent to wait on its children, and marshaled parameters for data transfer. The heuristics mentioned in the separate file are not currently used to prune the search space and optimize for fast results. We plan to work on these next.
- State Space Evolution: The state space search is akin to a tree. A node in this tree represents the whole input graph with some partial coloring, and is handled by a singleton chare. The input graph, stored as an adjacency list, is made a read-only data structure. A new chare receives the partial coloring of the graph from its parent, runs local algorithms on it, and then passes on the new partial coloring to its children in the tree. At a very high level, the local algorithms perform the task of selecting the next few vertices in the graph to color, assigning them colors, and firing off new chares. The algorithms are guided by various heuristics as discussed above.

3 References

References

[1] E. Boman, D. Bozda, U. Catalyurek, A. Gebremedhin, and F. Manne, *A scalable parallel graph coloring algorithm for distributed memory computers*, in Euro-Par 2005 Parallel Processing, J. Cunha and P. Medeiros, eds., vol. 3648 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 241–251.

[2] L. V. Kalé, B. H. Richards, and T. D. Allen, *Efficient parallel graph coloring with prioritization*, in Proceedings of the International Workshop on Parallel Symbolic Languages and Systems, PSLS '95, London, UK, UK, 1996, Springer-Verlag, pp. 190–208.

[3] V. Saletore and L. Kale, *Consistent linear speedups for a first solution in parallel state-space search*, in Proceedings of the AAAI, August 1990, pp. 227–233.

[4] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé, *ParSSSE: An Adaptive Parallel State Space Search Engine*, Parallel Processing Letters, 21 (2011), pp. 319–338.