Efficient Parallel Graph Coloring with Prioritization

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

Department of Computer Science
University of Illinois
Urbana, IL 61801

kale@cs.uiuc.edu

http://charm.cs.uiuc.edu
Objectives

To color a graph with C colors, such that no two adjacent nodes have the same colors.

Relevance

- Proven to be NP complete, but there are some good heuristics.
- Good search problem - techniques developed here can be applied to other search problems.

Aims

- To get the best sequential as well as parallel performance
  - Not just speedups
- Find a coloring, or determine if there are no C colorings.
- Develop techniques applicable to other search problems as well.
Search progression

Read Graph and build initial state

On each processor:
  if (unevaluated states exist)
    choose a state to evaluate.
  if (uncolored nodes exist)
    choose an uncolored node to expand on.
    for each color available to that node:
      assign the color, and create a new state.
  else
    report success.
else
  report failure
Heuristics

“It is difficult to get good speedups using good heuristics”
— they make the search more irregular.

• Variable ordering
• Value ordering
• Precoloring
• Node removal
Variable ordering

Choosing the uncolored node in a state to assign colors to.

- choose the one with smallest number of colors available to it.
- handle the most difficult cases first (decreasing the amount of searching to be done later)

Value ordering

Prioritize the coloring schemes

- Search subtrees more likely to contain a solution first.
- Less constraining choices.
- Using bitvector priorities
### Prioritization

<table>
<thead>
<tr>
<th>Color for A</th>
<th>Neighbors Affected</th>
<th>Heuristic Value</th>
<th>Rank</th>
<th>Bit-Vector Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>B and C</td>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>c2</td>
<td>C</td>
<td>4</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>c3</td>
<td>Neither.</td>
<td>5</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>c4</td>
<td>B and C</td>
<td>3</td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>
Redundancy in color assignments

For each color assignment, switching the colors of all nodes colored C1, and C2 produces an equivalent but different coloring. With C colors, the redundancy is C!.

Precoloring

Fix the colors for some of the nodes without loss of generality

- Bring redundancy down from C! to (C-2)! or even (C-3)!
- Failed searches have fewer states to search.
Node Removal

Remove the nodes in the graph which can be colored no matter what colors are assigned to neighbours (more colors available to it than uncolored neighbours)

- Reduces number of states
- Done recursively - removing a node may allow some of its neighbours to also be removed.
Node removal example: 3-coloring.
Refinements

- Impossibility testing
- Forced moves
- Split Graphs

**Impossibility testing**

Do not create states which have nodes with zero available colors.

**Forced moves**

If the state to be created has a node with one available color, skip it and go directly to the states it will create.
Split graphs

- Work on coloring the different parts of an unconnected subgraph in parallel.

- All of the subparts must be colorable for a solution to be reported.

Timeline:
A  Search begins from the main chare.
B  Split detected, form split parent.
C,D,E  Subgraph Solved
F  Final subgraph solution received send success to main.
G  Success arrives at main.
Parallel Refinements

Kill Chasing
- With split graphs, failure of one branch, can be used to stop the work on the other.
- Used to stop the computations once a solution is found.

Grainsize Control
- Ensure a minimum average grainsize per chare.
- Process states in sequence using a stack, until enough work has been done to merit the creation of child chares.
Performance Results

- Definite performance gains due to addition of heuristics
  - Value ordering helps with colorable graphs
  - Variable ordering, Pre-coloring and Node removal helps with all graphs
## Performance results

<table>
<thead>
<tr>
<th>File</th>
<th>Nodes</th>
<th>Edges</th>
<th>Colors</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 4</td>
<td>300</td>
<td>1626</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>Example 7</td>
<td>450</td>
<td>2451</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>Example 8</td>
<td>600</td>
<td>2338</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>Example 9</td>
<td>301</td>
<td>4274</td>
<td>5</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Input file summary.
<table>
<thead>
<tr>
<th>Processors</th>
<th>Shares</th>
<th>Execution Time (sec)</th>
<th>Time per Chare (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1440</td>
<td>1440177</td>
</tr>
<tr>
<td>2</td>
<td>1463</td>
<td>602</td>
<td>823</td>
</tr>
<tr>
<td>4</td>
<td>2138</td>
<td>311</td>
<td>582</td>
</tr>
<tr>
<td>8</td>
<td>2834</td>
<td>199</td>
<td>562</td>
</tr>
</tbody>
</table>

Table 2: Parallel speed-ups on a successful search on Multimax.
<table>
<thead>
<tr>
<th>GS setting</th>
<th>540</th>
<th>570</th>
<th>580</th>
<th>585</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>100907</td>
<td>94093</td>
<td>99954</td>
<td>193217</td>
</tr>
<tr>
<td>32</td>
<td>52187</td>
<td>46182</td>
<td>62686</td>
<td>145633</td>
</tr>
<tr>
<td>64</td>
<td>28182</td>
<td>26500</td>
<td>47837</td>
<td>129824</td>
</tr>
<tr>
<td>128</td>
<td>15648</td>
<td>16147</td>
<td>30022</td>
<td>96426</td>
</tr>
<tr>
<td>Statistic</td>
<td>Statistic Value (across processors)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean GS</td>
<td>30</td>
<td>59</td>
<td>436</td>
<td>2980</td>
</tr>
<tr>
<td>std. dev</td>
<td>11</td>
<td>82</td>
<td>991</td>
<td>6599</td>
</tr>
<tr>
<td>min-max GS</td>
<td>9-93</td>
<td>14-2411</td>
<td>17-8342</td>
<td>17-39689</td>
</tr>
<tr>
<td>Chares</td>
<td>50552</td>
<td>24075</td>
<td>2997</td>
<td>434</td>
</tr>
</tbody>
</table>

Table 3: Example 8 results on nCUBE/2.
Stack based grain size control

- Somehow, we must avoid large grains, while keeping the average grain size at a reasonable value.
- Estimating work under a node is hard.
- Idea: let a process procreate only when it has worked "Enough".
- Each chare (process) maintains a stack, and fires k children each time it crosses a threshold of work completed.
- Children development are fired from the bottom of the stack.
- This, or similar, method has been used by Halstead et al.
Utilization Plot

With normal grainsize control:

![Utilization Plot Diagram](image-url)
Utilization Plot

With new grain size control:

STAGES (Each stage is 100000 microseconds)
More performance data

Graph Coloring

Normalized Speedup (16 procs)

Number of Processors

Example 7
Example 8
Example 9
Graph Coloring

Processors v. Execution Time on nCUBE/2

Execution Time (seconds)

Processors

Example 7
Example 8
Example 9