Divide and Conquer and State-Space Search

Balancing Task Queues and priorities

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign

Task Parallelism

- The applications in these domains need *task parallelism*
- *Task* is a term that has been overused and overloaded
- Basic idea of tasks here:
  - Closed-environment (i.e. fully described) tasks
  - No pointers to variables in the parent task
  - (unlike *tasks* that are limited to within one shared-memory domain, which is another interesting area of research in parallel programming)
- These fully described tasks can execute on any node/cpu
  - The system has to move the *seed* for tasks (that fully describes the task) to some processor, to balance load
- Tasks are transient: the execute, send the result if any and are done
- In some applications, the tasks end up creating objects that persist:
  - e.g. there children may send results back to them

Balancing and Scheduling Tasks

- In a single process, there are two main alternatives
  1. Use a shared queue that all PEs (Pthreads) will use to enqueue/dequeue tasks
  2. Use one queue per PE, but balance queues as needed
     - Work stealing: when idle, dequeue from someone else's queue
     - Continuous rebalancing: monitor queue sizes and rebalance periodically
- In multi-node multi-process scenario
  - Only Option 2 above is reasonable (with both sub-options possible)
- Charm++ supports tasks, as singleton chares
  - Use a "seed balancer"
- Load balancing is an interesting research topic

Continuous Rebalancing of Queues

- Especially for multi-node execution
- Organize processes in some virtual topology with a relatively small connectivity
  - Low diameter graphs inspired by Moore bound are good here!
- Periodically exchange load information with neighbors in this topology
  - Load ≈ size of queue, for example
  - Or a more refined metric, if you have a heuristic for judging the load of each task
- Send task seeds to neighbors if you detect “above threshold” imbalance
- This method can, in practice, to better than work stealing alone
  - And can be combined with work-stealing (from a random processor globally)

Divide and Conquer

- Steps:
  - Divide the work: create tasks (chares in Charm++) for sub-pieces
  - Combine solutions: send messages to parents
  - Judiciously set grainsize : below which, execute task sequentially

State Space Search

- Example: N-Queens:
  - Place N queens on a NxN chessboard such that no two of them attack each other
  - Attack: are in the same row, column, or diagonal
- General form:
  - Start state is given,
  - Goal state is either given or described (e.g. no 2 queens attack each other), and
  - A set of permissible moves (operators) is defined

State Space Search: Find All Solutions

- If you are looking for all solutions, it is basically the same as divide-and-conquer
  - Send solutions up to the parent
    - When root get responses from all its children, you are done
  - Or collect them using some other mechanism (print, store locally)
    - But in that case, you need to know when your search is finished
    - This can be done via “Quiescence Detection” algorithms, which we will skip

State-Space Search

Finding The First Solution Faster

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign

Search for any feasible solution

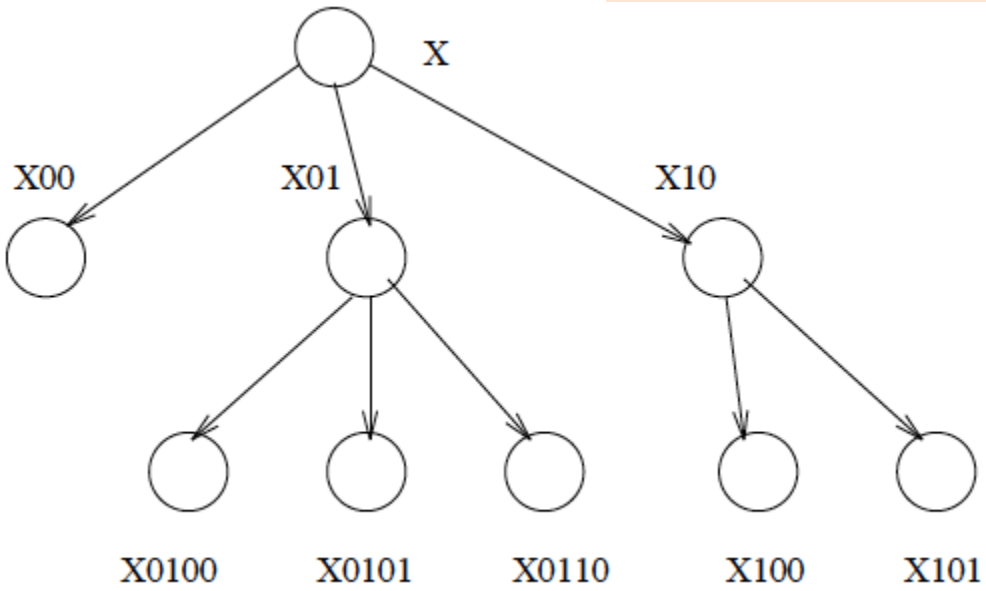
- This can be thought of as divide-and-divide
  - No combining step
- Question: how to stop search after a solution is found?
  - One solution: Broadcast and then exit or purge the queues..
  - It is a more interesting problem than that, in general, but we won't go into that.

Finding first solution faster

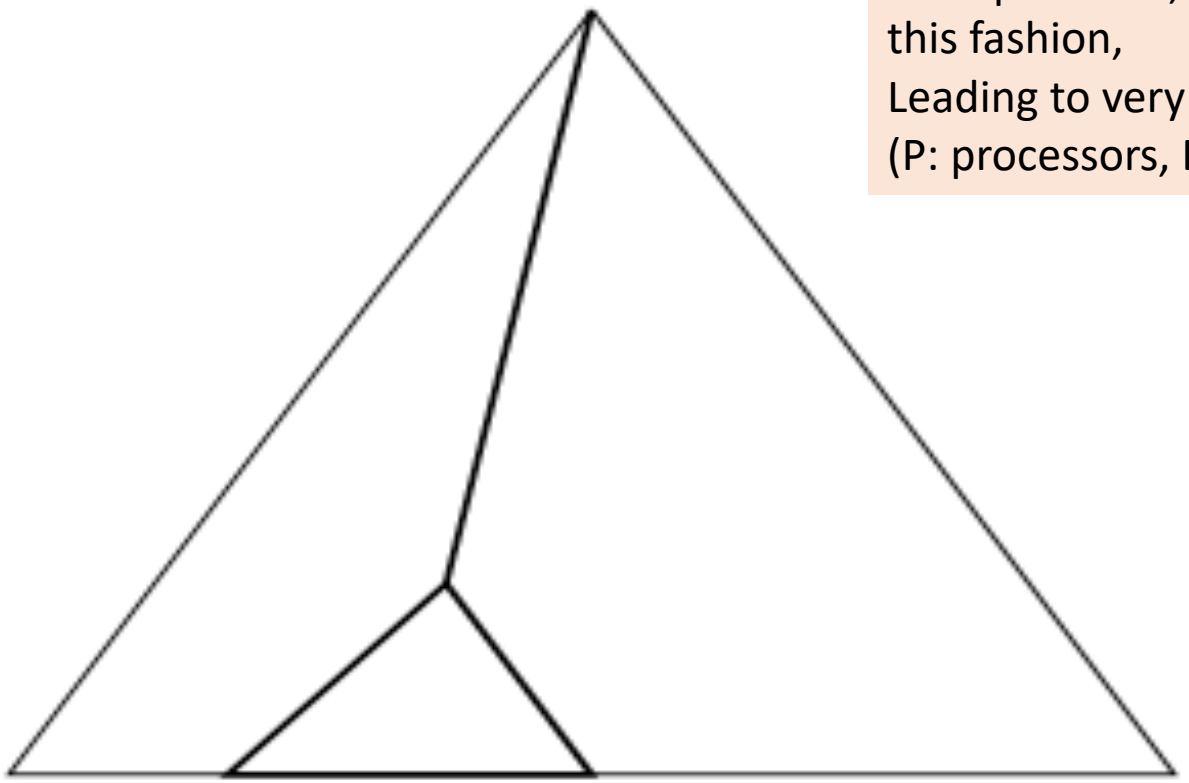
- The search tree is huge, how to quickly home in on a solution?
  - Various heuristics can rank children
  - Also, how to be consistent?
    - Search anomalies: more processors may slow it down and vice versa
    - We want monotonically increasing speedups, and consistency from run to run
  - Memory usage can be a big problem: exponential in depth
  - Left-to-right prioritization of the search tree

Left to right (prefix) prioritization

Priority is a bit-vector



Broom-sweep search



With priorities, search tends proceed in this fashion,
Leading to very low memory usage: P +D
(P: processors, D: depth)

Graph Coloring

- Given a bi-directional graph of V vertices and E edges
  - Find a way to color each vertex such that no two adjacent vertices are of the same color
  - Using no more than K colors
- A very rich problem in terms of parallel issues
- Basic parallel formulation:
  - Search tree
  - Each state:
    - Vector of vertices which have been colored
    - Optional/heuristic: # of colors available for each uncolored vertex
    - The graph itself is a replicated data structure (available on all nodes), not part of a state.

Heuristics

- Which vertex to select next? (e.g. one with fewest alternatives)
- Which color to color it with?
  - Try all available ones, of course, but in which order?
  - E.g. with a color that limits the neighbors the least
  - This leads to priorities among children, so left-to-right bitvector prioritization is useful
- Once its colored, can we simplify the graph or prune the subtree?
  - Any node with k-1 neighbors:
    - Remove from the graph,
    - Repeat!
  - Impossibility: A vertex with k or more neighbors with distinct colors.. Prune the tree
- Has the graph become partitioned?
  - I.e. two parts, and vertices that separate them are all already colored
  - If so, start 2 separate search problems and combine solutions later

Vertex removal : 4 coloring

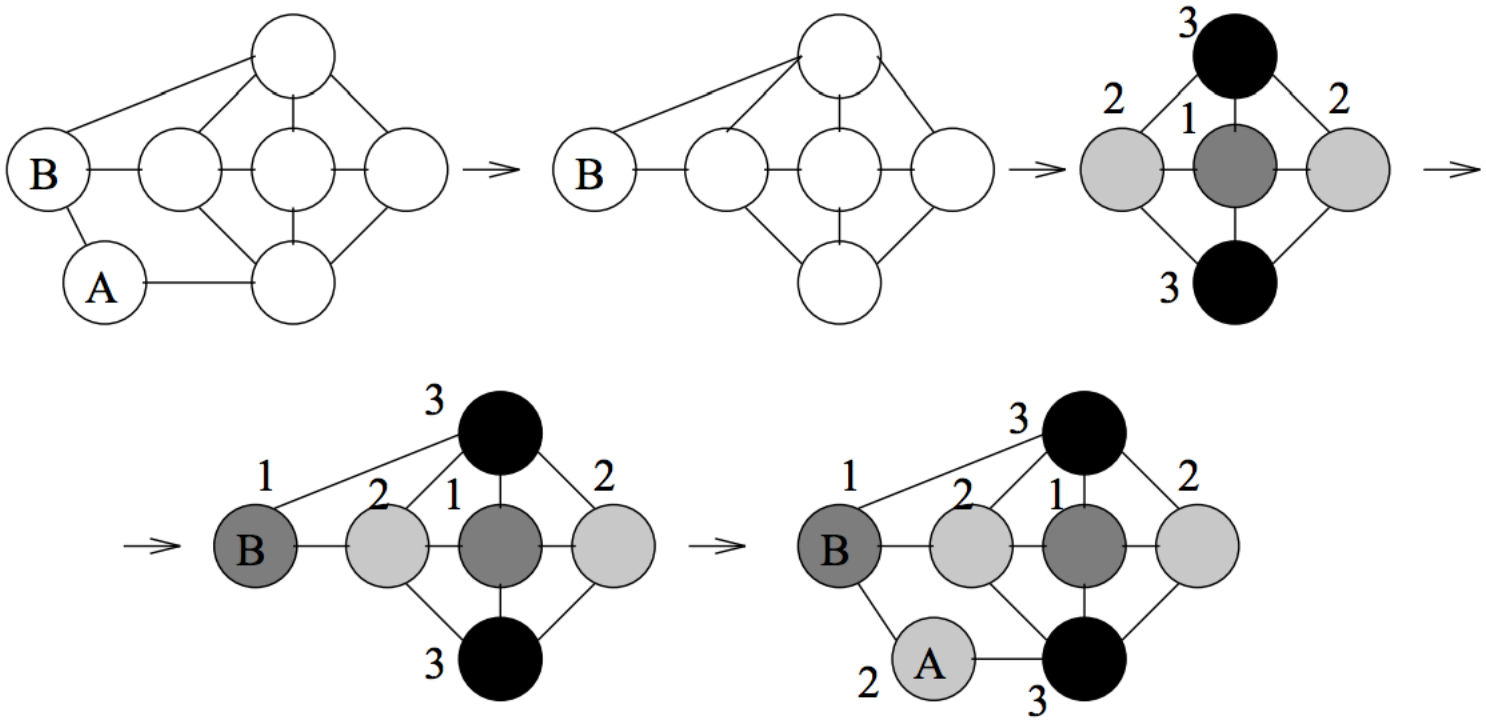


Fig. 1. The vertex removal process.

Heuristics

- Which vertex to select next? (e.g. one with fewest alternatives)
- Which color to color it with?
  - Try all available ones, of course, but in which order?
  - E.g. with a color that limits the neighbors the least
  - This leads to priorities among children, so left-to-right bitvector prioritization is useful
- Once its colored, can we simplify the graph or prune the subtree?
  - Any node with k-1 neighbors:
    - Remove from the graph,
    - Repeat!
  - Impossibility: A vertex with k or more neighbors with distinct colors.. Prune the tree
- Has the graph become partitioned?
  - I.e. two parts, and vertices that separate them are all already colored
  - If so, start 2 separate search problems and combine solutions later

Some more combinatorial search domains

- A* seach and IDA* (Iterative Deepening A*) search
  - 15-puzzle
- Bi-directional search:
  - 15-puzzle, rubik's cube
- Branch and bound:
  - Use bounds as integer priorities
- Graph coloring and
- AND-OR trees
- Game trees

Common *System* Issues:

- Duplication detection in search trees
- Scalable prioritized load balancing

Duplication Detection

- Same state may appear multiple times in the tree
  - (e.g. in 15 puzzle)
- Solution?
  - Store active states in a distributed table
  - Before firing a task, check the table
    - First hash function (applied to the state) finds the processor where info about that task may be found
    - Second hash function finds the local table entry, if there is one

Scalable Prioritized Load Balancing

- Continuous Balancing among queues across processes can balance loads, but what if each task has a priority?
- This becomes a more complex task
- A solution:
  - Hierarchical organization, with a manager for each subgroup maintaining the priority queue for that subgroup
  - Managers exchange high-priority task seeds among themselves