Parallel Backtracking

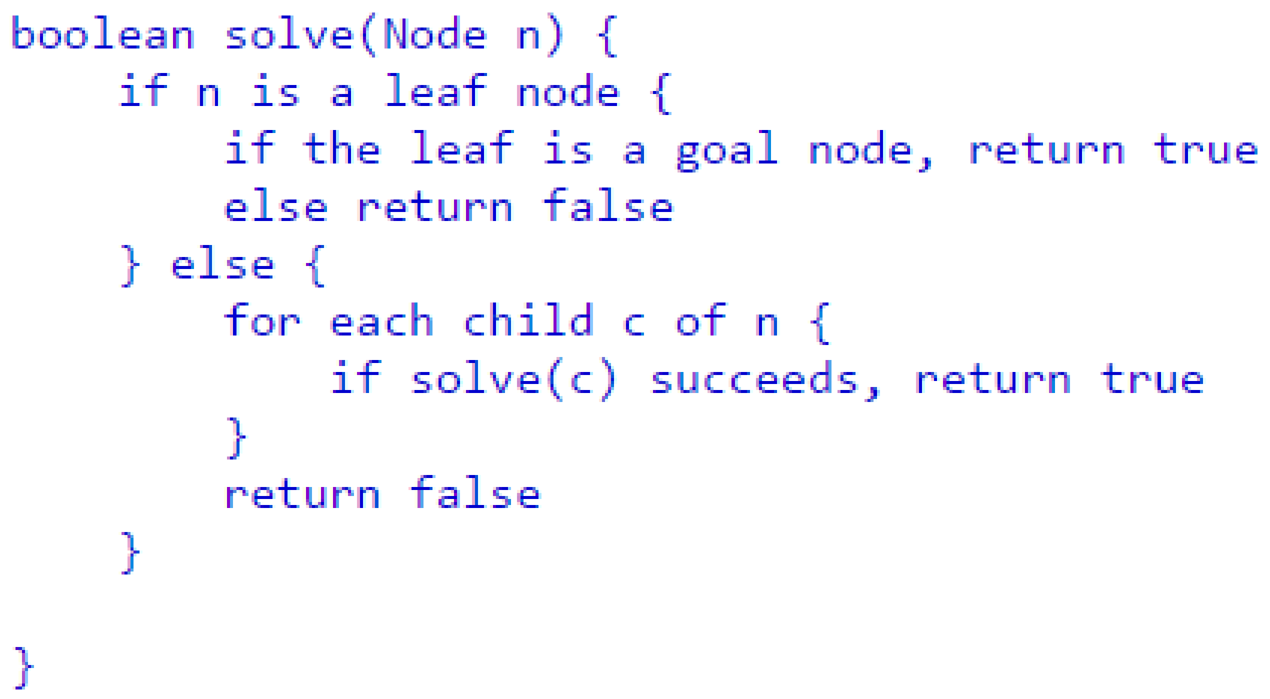
By:

Aakriti Jha
(2012B1A7840P)

Radhika Pasari
(2012B4A7445P)

Backtracking

- Traverses through possible search paths to locate solutions or dead ends



```
boolean solve(Node n) {
    if n is a leaf node {
        if the leaf is a goal node, return true
        else return false
    } else {
        for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
    }

}
```

Scope and Requirements

- Size constraints of cluster, knowledge of cores and hyper-threading supported
- Can be used to search solution in large (exponential) set of possibilities in a systematic way
- Can be used to build algorithms for hard problems

Example problems of Backtracking

- Solving Crosswords, Sudoku and other puzzles
- Used to solve queries in logic programming like Prolog
- Combinatorial optimization problems such as parsing and the knapsack problem
- Solving N-queen problem , m coloring and Hamiltonian Cycle
- Rat in a maze problem

Parallelism in Backtracking

- A parallel algorithm for backtracking would traverse (i.e. construct) multiple paths in parallel.
- As soon as one path leads to a correct solution exploration of all paths can be stopped.
- It could suffer from unbalanced load
- Scheduling has to be dynamic because paths may evolve at uneven rates.

Partition Phase

- When a large solution space has to be explored, search space is structured as a tree (or DAG)
- Each branch of the tree is a parallel task (as each path can be computed independently of another)
- Multiple branches can be explored in parallel
- Tasks may not be of equal size (unbalanced load)
- Order of number of tasks is more than processors to ensure flexibility

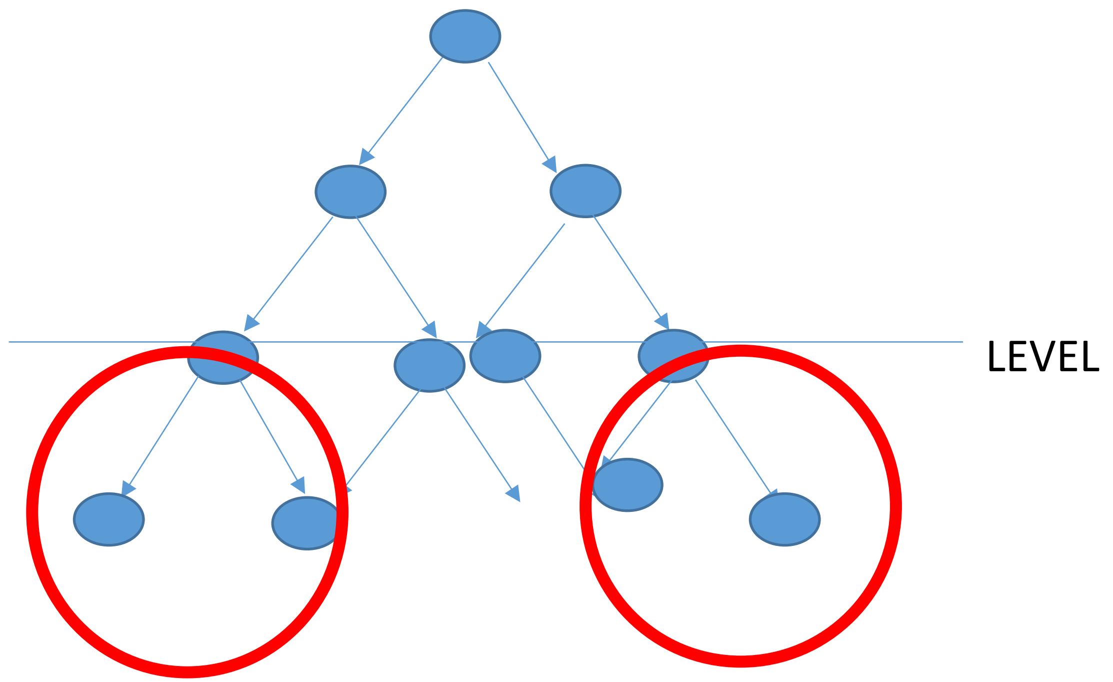
Communication

- Parallel Backtracking is expanding paths in the tree by DFS
- So, only information about search node along the path from current search node to root has to be stored (Structured)
- Asynchronous communication to all nodes when the result is found (Unstructured , Global, Dynamic)

Agglomeration

- If for every new recursive call we create a new task, all tasks do very little work in this implementation
- Overhead of creating the task will be high
- Each task must have a private copy of the state configuration, there can be a large memory usage
- So, we are creating a new task for each child node but only to a certain depth
- Total number of tasks are taken to be a multiple of number threads available

Agglomeration



- How to decide the LEVEL
- P processors ,c cores we want number of tasks to be a multiple (m) of p*c
- At level, the number of child nodes (tasks) should be close to m*p*c

Mapping

- To maintain locality, we assign different nodes to initial levels of search tree as the state of search information to be passed onto the child node is small
- As the depth of tree increases, we keep sub-tasks on different cores of same node to utilize shared memory locality

Dynamic Scheduler

- Some branches grow at uneven rate, so dynamic Scheduling has to be done
- We can maintain a pool of tasks and use manager worker structure to assign pool of tasks to nodes
- Type of manager for assigning tasks (Further reading)
- On demand work stealing to minimize processor's idle time (Further reading)