Backtracking

Parallel Computing

Swasti Kakker
Ashutosh Bhatt
Prabodh Agarwal

Term Project

1 Backtracking

Backtracking is a refinement on brute-force approach. Systematically a space of possible values for given problem is searched to generate a solution(s). It is assumed that solution values are available as a vector of values. A permutation of these values is explored in a depth first manner until a possible solution is found.

Initially, the solution vector is empty. At every stage of traversal, the vector is extended with a new value generating a partial vector. If the partial vector can't be extended anymore and the problem is not yet solved, the algorithm *backtracks* to replace last added values until a new value fits in the partial solution, thereby forming a branch in the tree.

Existence of a path from root to solution with decisions based on different values at tree edge and partial solutions at internal nodes, a *backtracking* solution, thus forms a tree.

```
ALGORITHM try(v1,...,vi)
        IF (v1,...,vi) is a solution THEN RETURN (v1,...,vi)
        FOR each v DO
                IF (v1,...,vi,v) is acceptable vector THEN
                sol = try(v1,...,vi,v)
        IF sol != () THEN RETURN sol
        END
    END
    RETURN ()
```

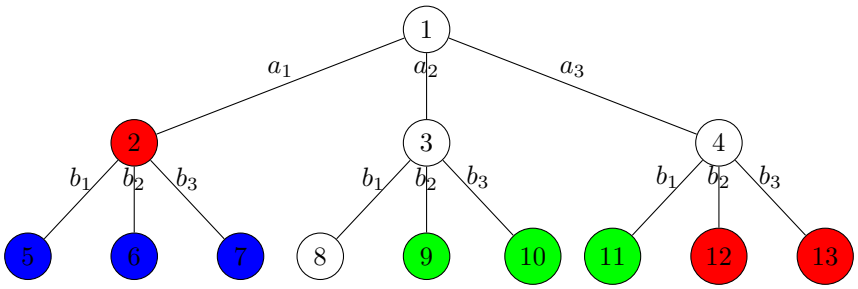

Figure 1: Depiction of a backtracking search tree. Node 2 is not a valid solution thus, nodes 5,6 and 7 are never calulated. Similarly, nodes 12 and 13 are not in solution space. Nodes 9,10 and 11 are solutions. Node 8 is still being calulated and may spawn more branches.

From fig.1, we can see there are *three* possible solutions to the problem in consideration. They are $\{a_2, b_2\}$, $\{a_2, b_3\}$ and $\{a_3, b_1\}$. Thus, each path from root to *green* node is one solution. Also, input a_1 goes to node 2 however, it turns out that this solution is invalid thus nodes 5,6,7 are never formed.

This causes *backtrack* and solution search replaces a_1 with a_2 and proceeds to node 3 and so on. Close analysis of this search tree clearly shows how path formation is independent of one another and that a solution space starting a_1 is independent of that starting at a_2. There is inherent parallelism in this problem solving approach which can be extracted by spawning paths leading to sibling nodes at once in parallel.

For example,
Edges $(1, 2)$, $(1, 3)$ and $(1, 4)$ can be spawned in parallel. Nodes 2,3 and 4 can further spawn their children edges in this manner and so on.

2 Problems

For our term project, we have selected two problems to demonstrate backtracking.

- Nqueens generator
- CNF-SAT solver

Following sections outline in detail for each problem

- Explanation of problem statement
- Extracting parallel structure of problem through backtracking
- Explain a backtracking algorithm
- Designing parallel solution using *Foster's Design Methodology*

3 N-Queens Generator

Problem Statement

Given a NxN chessboard, find a way to place *N queens* on the board in such a way that no queen cross the other. Thus, a solution requires that no two queens share the same row, column, or diagonal. We are aiming to generate all possible solutions for the N Queens Problem.

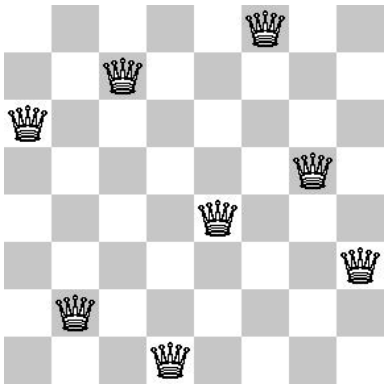

Figure 2: An 8x8 NQueens.

Algorithm

The solution space for the N queens problem would clearly be $\{1, 2, 3, \ldots, N\}$. Thus, trying all vectors $(p_1, p_2, \ldots, p_n)$ implies n^n cases. Noticing that all the queens must reside in different row reduces the number of cases to $n!$. Thus we will try to keep a queen in each of the rows.

Let us take up the 4x4 Queens problem for understanding the algorithm. Fig 3 is a partial graph indicating the tasks required. Thus, at the root node of our tree, we will make the decision of placement

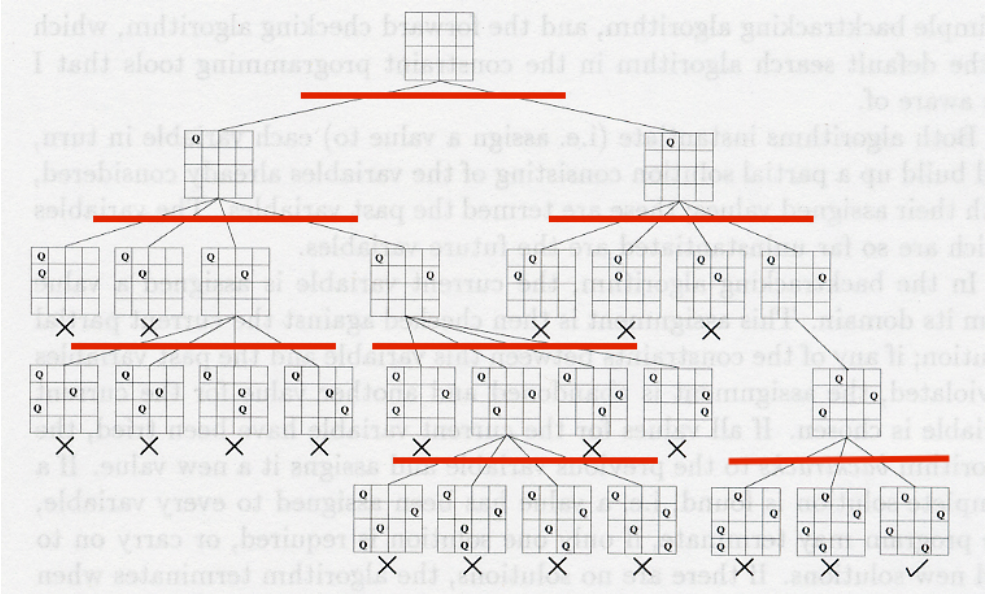

Figure 3: An partial NQueens solution.

of the first queen in the first row. The nodes indicate position where the queen is placed. Each level, indicates the row in which the queen is to be placed in that level.

Clearly, when we place a new queen we check if we can continue placing new queens or should we halt. Checking this would involve if we would be able to place N queens subsequently on the board due to placement of that queen. If this is not the case, we shall abandon going down the tree further. Such nodes are marked by crosses in the partial graph.

```
FOR R=1 to N
        TRY placement of queen for row R in each of the N positions
        (column 1 to N )concurrently.
        IF this placement could lead to a valid board configuration,
        then increment R and repeat.
    ELSE break.
```

Foster's Design

Partitioning

The number of valid positions we can place a queen, will be different tasks. Now for that valid entry, we'll have multiple branches (and hence different sibling tasks) depending on the number of valid positions left for us to place queens. This is how the partitioning is being done. Each of the tasks can execute in parallel. The sibling tasks are independent of each other.

Communication

As earlier stated, the sibling tasks are independent of each other, hence not much communication is required between the sibling tasks. The only communication required is between: parent task and child tasks,i.e. every parent task communicates state of the board to children tasks.

Agglomeration

We can here intuitively realize that a parent task when communicating the board information to its children (say n), is sending the same message n times. To reduce redundancy, we implement a shared memory model. The parent task may send the current board position to all the children tasks. Hence, the number of messages passed will be low but frequent.

Mapping

Given, p processors, and t threads per processor, first p tasks will be sent to individual processors. Subsequent tasks will be kept on a task pool of individual process. The processor picks tasks from this task pool. Now there is high probability that that a particular processor is idle while other processors are busy. Each processor has a decentralized manager, which requests tasks from the task pool of idle processors if the processor is idle.

4 CNF-SAT Solver

Problem Statement

The *Satisfiability Problem (SAT)* is a combinatorial problem. Given a Boolean formula of n variables

$$f(x_1, x_2, \ldots, x_n),$$

values need to be supplied in a manner, the formula evaluates to true.

The *CNF Satisfiability Problem (CNF-SAT)* is a version of the Satisfiability Problem where, the boolean formula is specified as a Conjunction of *clauses*, and each clause is a disjunction of *literals*. A literal is a variable or its negation. It is an AND of ORs. Example:

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_4)$$

An assignment of $x_1 = true, x_2 = false, x_3 = false, x_4 = true$ *satisfies* the above formula. There can be a few more distinct assignments of $x_i s$ to satisfy this formula.

We are aiming generating all possible solutions to any given formula.

Algorithm

Each of x_{1-4} can take possible values of *true* or *false* thereby giving 2^4 possible solutions to test on and a search space of *16* values.

The backtracking solution to CNF-SAT is given by the *Davis-Putnam-Logemann-Loveland*(DPLL) algorithm[1]. The backtracking algorithm builds a tree by choosing a *true* or a *false* value for a literal and then simplifying the clause in a manner as given in DPLL.

```
Repeat
        Select a literal l (some x or x')
        F = Simplify(F, l)
```

```
While F contains a 1-clause l'
        F = Simplify(F, l')
If all clauses removed return SAT
If there is a 0-clause
        Backtrack to last free step and flip assignment
```

Simplify(F, l) works as

- remove clauses containing l
- from all other clause, remove l'

Application of Simplify(F, l) to unit-clause is called *Unit-propagation*.

Following figure 4 shows formation of backtracking tree for given formula by above mentioned algorithm.

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3)$$

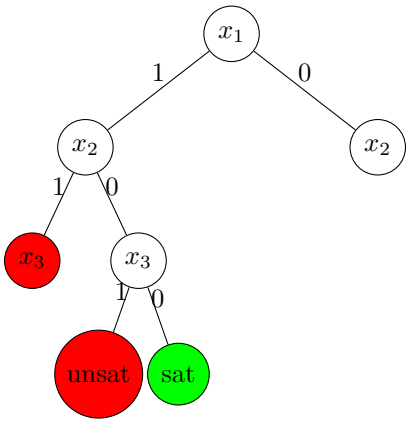

Figure 4: Depiction of a backtracking search tree for CNF-SAT. More solutions can be possible if $x_1 = false$ is explored in a similar manner.

Thus each edge of the tree corresponds to assignment to a particular literal and accordingly, simplification of the formula. Each node is output of the decision indicating if the partial solution can satisfy the formula or not.

A parallel implementation of this algorithm will spawn both tasks of literals being *true* and *false* together. If at a level i, a partial solution of assignments x_1 to x_i causes one or more clauses to become 0, that path is stopped from spawning more tasks. The fact that each path of the tree is independent of sibling paths permits us to construct edges in parallel so as to perform an exhaustive search of the solution space.

Foster's Design

Partitioning

The parallel algorithm selects one literal per task depending on the tree level. Tasks are uniform with each getting a literal depending on state of it parent task and simplifying the clause based on that task. The nature of problem is fine-grained and no partitioning shall be required [2].

Communication

The edges of the tree are the tasks and also depict a communication direction as each task communicates the formula to spawned tasks and info on literals still left to be assigned. Communication is uniform and

a 2-time affair.

- First when newly formed task receives from parent task
- When a task communicates the formula to children tasks.

Agglomeration

Due to each task communicating its state to children tasks, and tasks themselves being fairly easy to compute, instances of communication would be high. Same data shall be communicated by a task to two children tasks. This shows involvement of three tasks on same data, thus, all three of these tasks should implement shared memory to reduce message passing communication, and thereby keep time spent in message passing in check.

Mapping

Given, p processors, and t threads per processor, first p tasks shall be sent to individual processors. Subsequent tasks on a given processor shall be kept on a task pool. The processor shall pick tasks from this task pool. As depicted in 4, leftmost task finishes earlier and there is high probability that that particular processor is idle while other processors are busy. Each processor has a decentralized manager, which requests tasks from the task pool of idle processors [2].

References

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[2] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.