Note : There are two nodes referred in this document, a tree node and a cluster node. It will be mentioned where not implied.

Some definitions for the problem to be solved and it's Foster's Design :

Use Task Decomposition

Assume a general problem that has -

- A possible solution of the form - an n-tuple (a1, a2, a3, a4 …., an)
- A search tree for systematically listing all such possible n-tuples using depth-first order
- Each non-leaf node in the tree corresponds to a partial solution
- Each feasible solution corresponding to a path from the root node to the leaf node (one-to-one mapping)
- Non-leaf node children of a non-leaf node correspond to more complete possible solutions, i.e., one more variable in the tuple is decided with a specific value.

Partitioning:

In the worst case, the feasibility of a solution needs to be tested for each permutation of the n-tuple.

Tasks to be performed -

1. Testing the feasibility, this needs to be done for each possible complete n-tuple (leaf node)
2. For each possible partial n-tuple (a non-leaf node), EXPAND it by fixing one more variable in the tuple while maintaining the possible feasibility of the solution.

Communication

1. The task corresponding to a tree node is dependant on the computation performed by it's parent tree node i.e., the list of values fixed so far in the n-tuple
2. At termination - Communicate to all other threads when a solution has been found by one of the threads or there is no existing feasible solution

Agglomeration

1. Computation for each list of dependent tasks (a path with nodes from root to leaf), can be on the same node, to reduce amount of communication required.
2. Exception - In case a few threads on a separate node are idle, migrate tasks from a busy node to the node with idle threads. This is done at runtime, and the task grouping changes.
3. ***Randomized*** - Explained in detail in the algorithm and communication pattern.

Mapping

1. Each task corresponding to a node that has already been generated, but is to be expanded, might be assigned to an idle node at runtime.
2. **Initial state** - One thread of one node is assigned to start from the expansion of the root. Other processes are idle at that time.

Detailed runtime explanation in the algorithm and communication pattern followed.

The generic algorithm for solving backtracking :

Stacks will be used to maintain the set of tasks allotted to a thread. This is used to maintain the depth first structure.

The problem is associated with a rooted tree with the internal nodes representing partially solved problems (these partial solutions may or may not be correct).Though this is a general template algorithm, which may be used to solve any backtracking problem, we explain this association with an example:- the sudoku solver. Here the root represents the sudoku puzzle with only prefilled numbers which are part of the question. We select one unfilled box in the puzzle to be filled.This selection can be made with or without any heuristics (our initial runs suggest worst fit choice gives better running time for the sequential algorithm). Worst-fit means choosing the box which can be filled in maximum number of ways. Again once we choose a box to be filled, the root node then spawns several subtrees which correspond to the actual number chosen for this box. It is clear that the number of subtrees will be between 1 and n for a nxn sudoku board, according to the number of non-conflicting entries at that position. Now the root of the subtree is the sudoku puzzle with 1 additional box filled. Note that the problem can be broken down into these partially filled subproblems until we encounter a box which has

no possible numbers which fit, for that state of the partially filled puzzle. This leads us to backtrack and choose another possible filling for some box up the tree. We do this until we reach a leaf node which corresponds to a filling of all boxes in a non-conflicting manner. This is the general method for backtracking. We propose the following algorithm for parallel backtracking.

Initially a single thread is assigned the root of the tree, which is at the top of a local stack of the thread. A thread is called **busy** when it is assigned at least one task node.

In a step, a busy thread executes the following actions:-

1) Expands the node at the top of the stack if it is not a leaf node. If it is a leaf node which gives a valid solution, we are done and all threads are notified and asked to terminate. If it is an invalid leaf node, discard it.
2) Donate work to an idle thread if requested. An idle thread is one that has no task nodes assigned to it and it randomly selects a thread and requests for work to be shared.
3) Donation procedure - Assigns half of the top-level nodes it has (in its stack), to an idle thread only, if requested.

The top level nodes are the task nodes with the minimum level allotted to a thread, that need to be expanded and completed.

The following rules are strictly followed for a donation:-
Only an idle thread can ask for work - Only a busy thread can donate work - A receiving thread can receive from a single thread only.

The process of pairing i.e. busy threads donating to idle threads is random in the sense that the idle threads randomly request a thread for work.

Pseudo Code:

```
F1 = { root };
/* total number of threads available is p */
for i = 2 to p
        Fi = Φ;
while ∃ Fi != Φ
        for i = 1 to p
                if Fi != Φ //Implies thread is busy
                        expand the leftmost node vi in Fi;
                        Fi = Fi - { vi };
                        if Fi is not a leaf
                                Fi = Fi ∪ children of vi;
                Else //Implies thread is idle
                        /* randomly choose one node to request work */

        /* donation step */
        for i = 1 to p
                If busy thread
                        let Gi be the set of top-level nodes and Hi be a subset of Gi such
        that | Hi | = | Gi |/2
                        Gi = Gi - Hi;
                        Send Gi to the requesting idle thread.
                If idle thread
                        If received response from a busy thread, accept and start work
                        Else ===> received response from another idle thread do nothing

                        If received request from another idle thread, respond with a signal
                to notify the it is itself an idle thread
```

An example run of of the algorithm

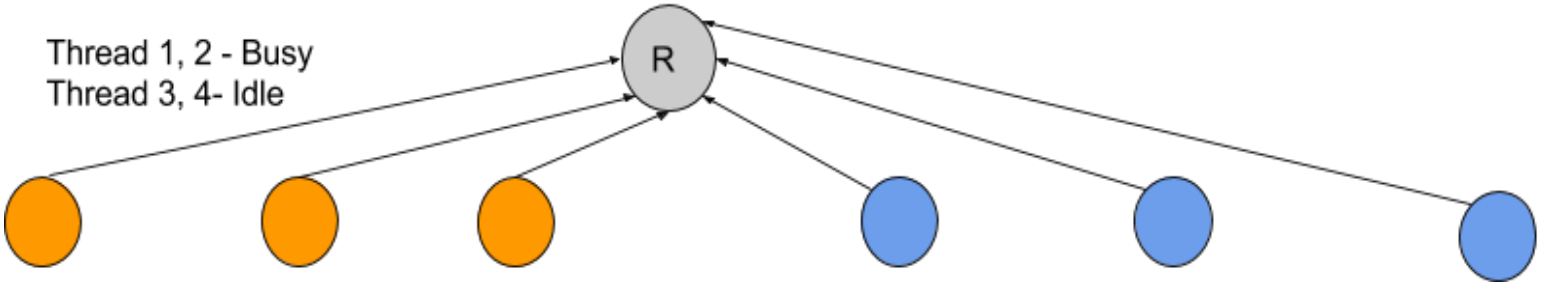
1st iteration - Thread 1 is assigned the task of root node.

Thread 1 - Busy (At least 1 task at hand)
Thread 2, 3, 4- Idle (No task at hand)

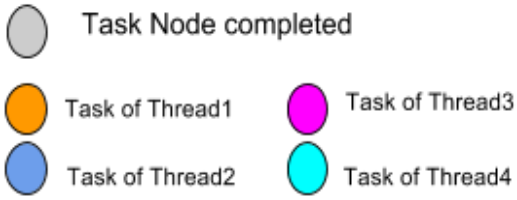
Assume 4 threads in total

2nd iteration - Thread 1 **expands** the root node by filling one variable (a cell in the sudoku problem) in the tuple with all possible feasible values. This is done separately for each empty variable at the node. Task of root completed. Equally share top-level tasks with an idle thread.

Thread 1, 2 - Busy
Thread 3, 4- Idle



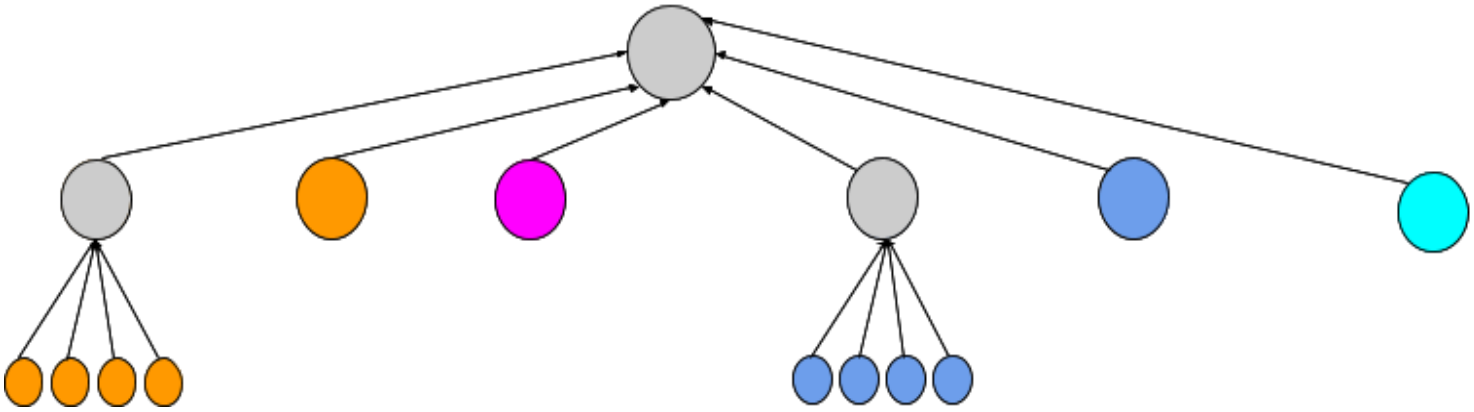
Assume 4 threads in total



3rd iteration - Thread 1 and 2 **expand** the next respective task (in dfs order) at hand. Equally share top-level tasks with idle threads.

Thread 1, 2, 3, 4 - Busy

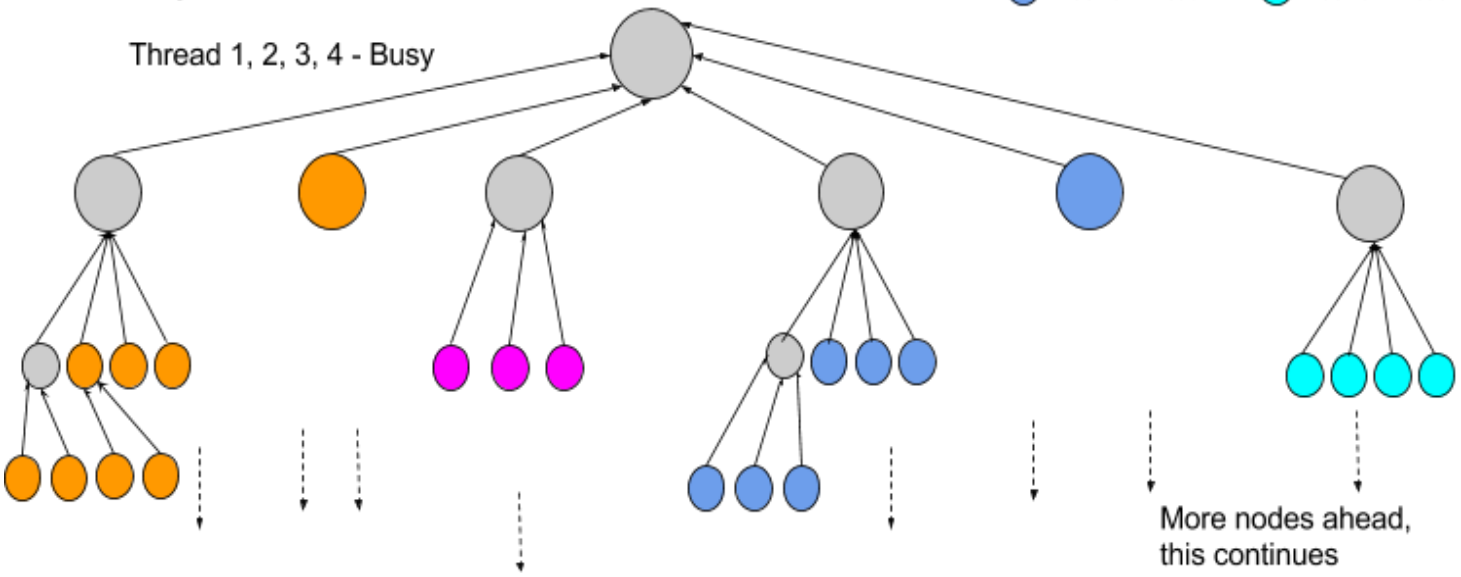
Assume 4 threads in total



4th iteration - Thread 1, 2, 3, and 4 **expand** their respective next node. No more idle threads, each thread runs sequentially till the size of pool of idle threads is 0. Once any thread becomes idle, it will get tasks from a busy node.

Thread 1, 2, 3, 4 - Busy

Assume 4 threads in total



NOTE :
The mapping of an idle thread with a busy thread is randomized as per the algorithm. The mapping shown in the previous illustration is just an example.

Also the number of children from each task node can be varying. For example, in the sudoku problem, the number of ways to possibly fill a cell depends on the constraints put up by row, column and sub-square values that are non-empty. Thus, the task nodes of the problem will generate sub trees with uneven rates of growth.

Possibilities for filling a cell c at a task node in the sudoku grid:-

```
//Assume that rows and columns start range from 0 //to 8
        Let S = {0, 1, 2, 3, 4, 5, 6, 7, 8}
        for i = 0 to 8
                if grid[i][c.col] is filled
                        S = S / {grid[i][c.col]}
                if grid[i][c.row] is filled
                        S = S / {grid[i][c.row]}
        int subCellCol = c.col/3;
        int subCellRow = c.row/3;
        for i = c.col*3 to c.col*3 + 2
                for j = c.row*3 to c.row*3 + 2
                        if grid[i][j] is filled
                                S = S / {grid[i][j]}
//The set S now contains the possible values the chosen cell c can take.
```

Communication pattern : (The most important part)

What is to be communicated?

An idle thread needs to receive half of the partial solutions computed so far by a busy thread that the idle thread is going to share work with. The busy thread is chosen randomly by the idle thread. So, suppose there are **m top-level nodes** in the busy thread and the size of each partial solution will be an **n-tuple** (n is 81 for sudoku). The size of message is then (**m/2)*n**.

The algorithm given assumes that all threads can communicate with all other threads with equal overhead. But the total pool of threads is distributed on separate nodes and nodes communicate with MPI. To reduce the communication overhead, the following structure can be followed for assigning tasks to idle threads. This structure constrains the ***amount of randomness*** as proposed in the algorithm.

There is a master thread in each node handling the work of any communication with other nodes.

The **level of any thread** is the minimum of the set of level of all task nodes. Level 0 starts from the root.

A cluster node is idle, if all threads of that cluster node are idle i.e., no tasks allotted. A cluster node is busy if any thread in that cluster node is busy.

1) If a busy node receives a request from an idle node, the master of the busy thread equally shares the top-level tasks of the thread with minimum level.
2) Within a busy node, each busy thread divides its top-level tasks with an idle thread (if available). This internal division is deterministic and can be done using a binary indexed tree like mapping (to be done in implementation).
3) The master thread of an idle node randomly chooses a node from the cluster and requests work from the master thread of that node. If a busy node was requested, its master will share work as in 1). If another idle node was requested, the idle node will be notified of no work availability and it will again randomly choose another node.

Reason for this communication pattern:- To avoid inter-node MPI communication in the long run. The idea is to divide work first with other idle cluster nodes, so that the process of work division occurs simultaneously in all cluster nodes and number of work

division procedure calls that occur increase **exponentially** to reach a "**no idle thread state"** in **logarithmic** time i.e., sooner. Also, by dividing a huge chunk of work (**half is the best option**) with an idle node, the MPI communication is reduced overall, because there are **lower chances of a node becoming idle again and requesting another node for work**. This is the constraint on randomness.

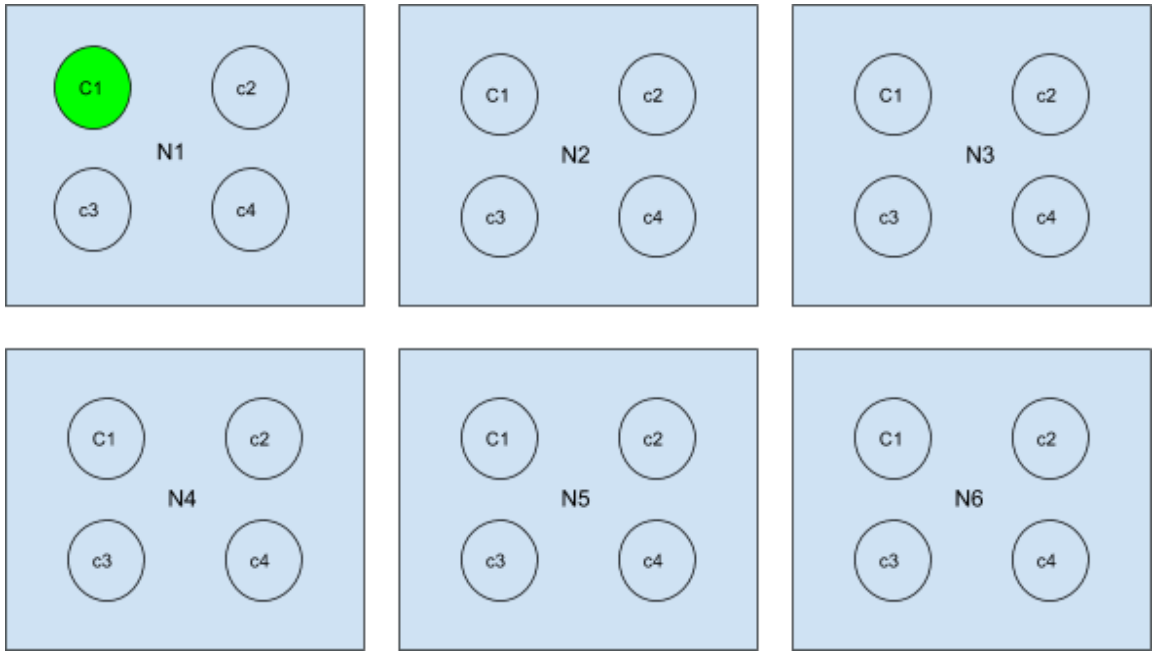
The opposite way? : Choose to share work first with internal idle threads of node and then other idle nodes. The better out of the two methods can be found only after testing.

The number of threads can be maintained as **number_of_cores + 1** or **2*number_of_cores + 1.** The **multiplication with 2** - for the **thumb rule** to keep each core busy. The **+1** is for the master thread, because after some time the master thread will be idle and there won't be any need for a master thread, and all other threads can occupy all cores.

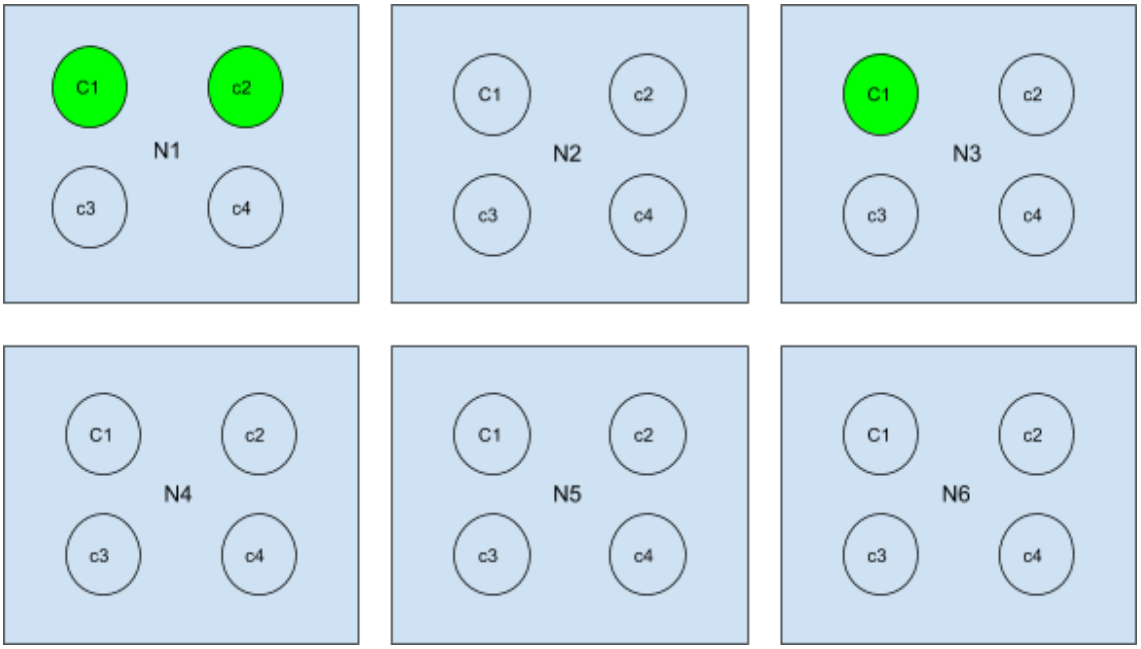
The following flow might be an example to a possible flow communication.

Green ===> busy thread.

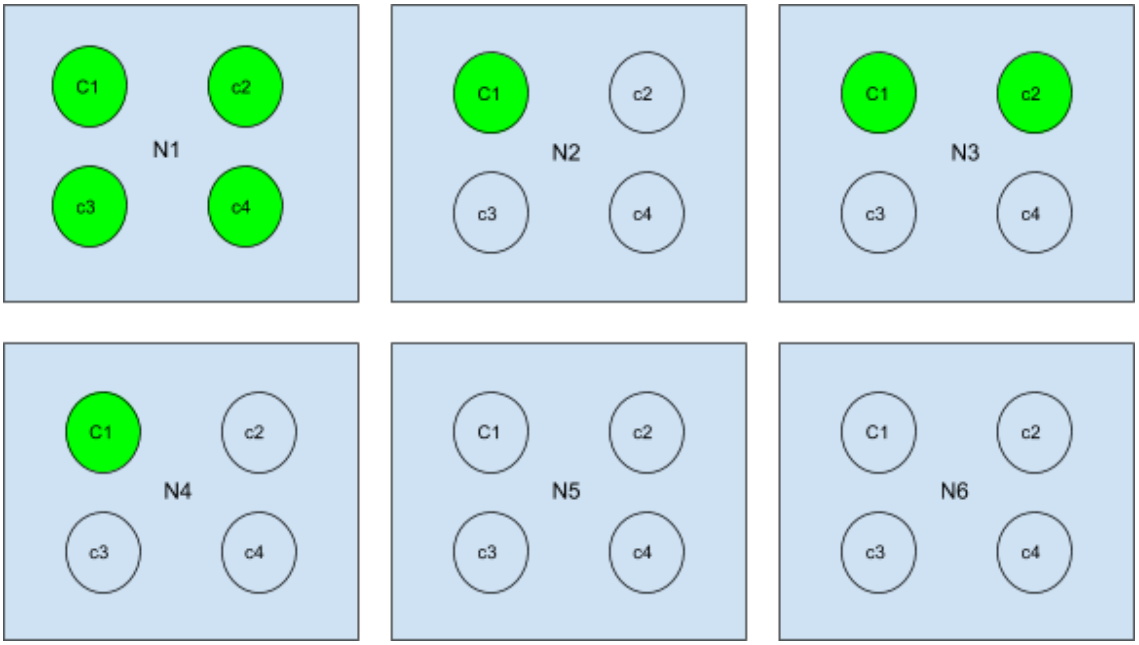
Start with a thread in a node allotted with the root node task.



Iteration 1 : Suppose N6 and N3 request N1 for work. Requests of N2, N4, N5 are wasteful, i.e., they are to other idle nodes (because they are random, they can be useless). Internal sharing of work in N1 is also to occur. A busy node doesn't know about an idle node, until a request is received from an idle node. N3 is served and N6 is not served, due to index priority.



Iteration 2 : Now suppose N2, N4, N6 and N5 change their random choice and N2 requests N1, N4 requests N3 and N5 and N6 request each other. Internal division occurs in N1 and N3 in the meantime.



This process continues till a “no idle thread state is reached”

Problems to be solved using proposed template :

1) Sudoku solver - Given an input puzzle, find the solution to it. (n*n)
2) Knight’s tour - Given a starting position of a knight on a cell, find the knight’s tour. (m*n - such that solution exists)

The general sizes are for testing scalability if the standard grid sizes of the problems are not enough.

Scope:

The two problems chosen will be implemented using a hybrid system. All of the required factors are considered and issues that may arise at the time of implementation are taken care of. The final result will be a cluster system capable of solving the problems mentioned above. The project implementation can be completed by the deadline.