Title: Distributed Chess Engine

Team No: 39, Members : Siddharth Gupta- 2021201082, Pulkit Gupta- 2021201037, Vishal Pawar- 2021201025

Problem Statement

The limitations of sequential chess engines, such as limited processing power, longer analysis time, and reduced scalability, pose a significant challenge to their ability to provide accurate and efficient move recommendations. As the size of the game tree grows exponentially with the number of moves, sequential chess engines are limited in their ability to analyze the game state to an optimal depth. This can result in suboptimal move recommendations and an inefficient use of resources.

The problem statement, therefore, is to develop a more advanced approach to chess engine development that can leverage the power of distributed computing to overcome these challenges and provide more accurate and efficient move recommendations. The objective is to design a distributed chess engine that can distribute the computational load across multiple processes, allowing the system to process larger amounts of data in parallel. This will enable the engine to analyze the game state much faster and provide more accurate and timely recommendations for the next move.

The challenge is to develop an efficient algorithm that can coordinate communication between the different processes and analyze the game state to an optimal depth. The resulting distributed chess engine will provide users with the best possible move recommendations for any given game situation and enable a seamless and highly responsive experience.

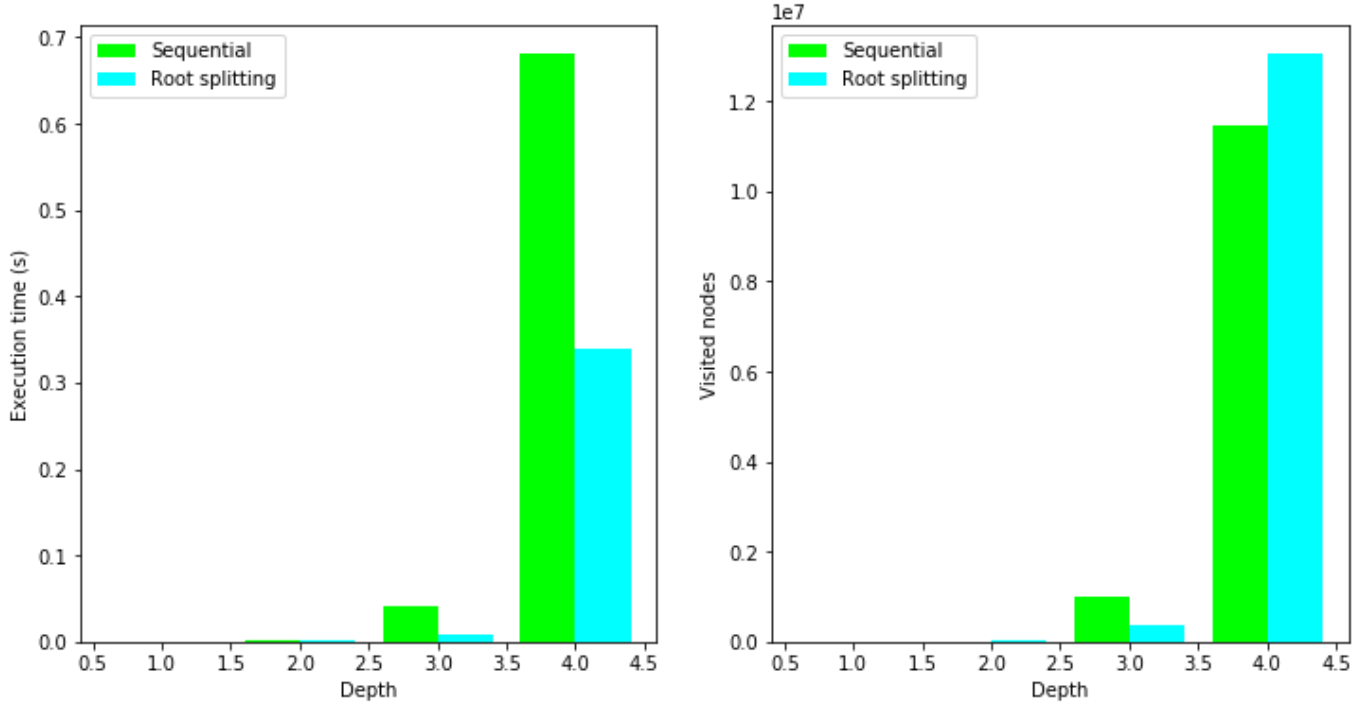
Root splitting

The main idea of this technique is to ensure that each node except for the root, is visited by only one processor. To keep the effect of the alpha beta pruning, we split the children nodes of the root into clusters, each cluster being served by only one thread. Each child is then processed as the root of a sub-search tree that will be visited in a sequential fashion, thus respecting the alpha beta constraints. When a thread finishes computing it’s subtree(s) and returns its evaluation to the root node, that evaluation is compared to the current best evaluation (that’s how minimax works), and that best evaluation may be updated. So to ensure coherent results, given that multiple threads may be returning to the root node at the same time, threads must do this part of the work in mutual exclusion: meaning that comparing to and updating the best

evaluation of the root node must be inside a critical section so that it's execution remains sequential. And that's about everything about this algorithm

Simulation (Seq vs Distributed) - Full games

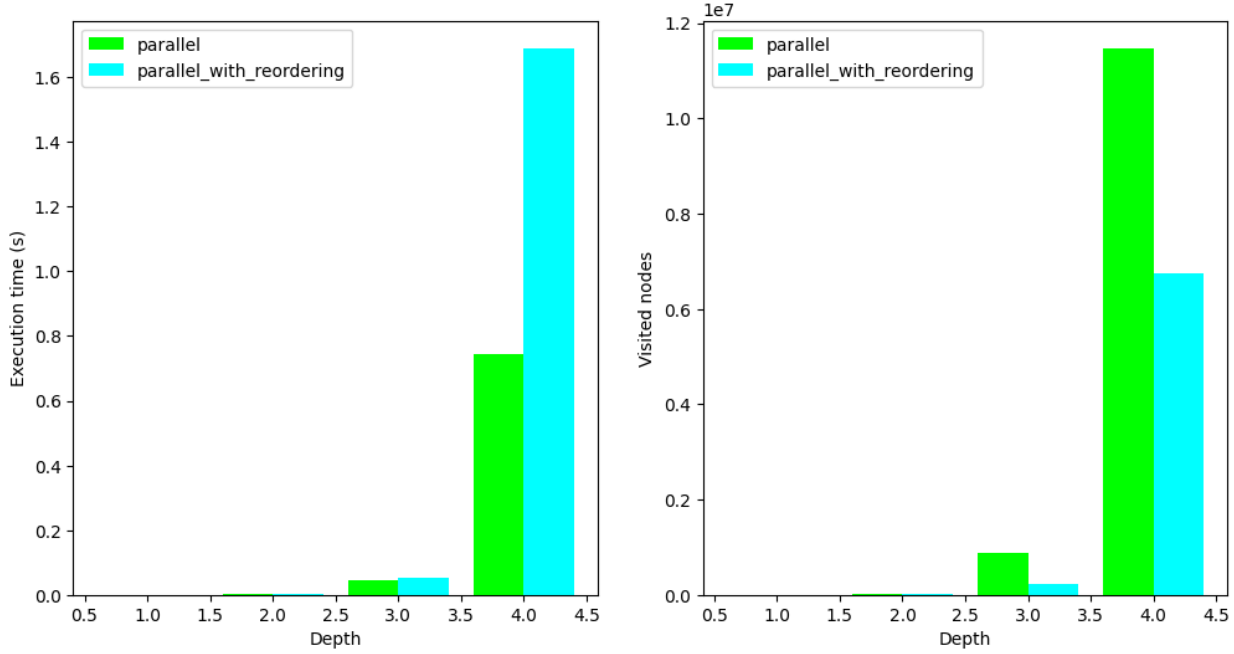
This simple program makes the old engine play against the distributed one at different search depths. For each depth, we make them play n games, we record the time taken for each move (in chess terms: each ply) of the game for both players (separately, of course!) and then we compute the mean of all the moves for all the games. In addition to execution times, we store the number of nodes visited to compare the search spaces explored by both players. The machine used has Intel I5 4th generation processor and 8 GB of RAM.



The distributed algo is clearly faster at depths 3 and 4 also even if the number of visited nodes is slightly higher for the distributed engine.The distributed algorithm can divide the work among multiple processors or machines, allowing it to explore the search tree in parallel. This can lead to significant speedups, especially for large search trees.

Move Reordering:
Move reordering is a technique used in computer chess algorithms to improve their efficiency. It involves evaluating and ordering the possible moves for a given position based on their expected effectiveness, rather than processing them in a predetermined or random order. By evaluating and prioritizing the most promising moves early on in the algorithm's search process, move reordering can reduce the number of nodes that need to be explored, thereby increasing the algorithm's speed and efficiency.
The sooner we prune the better, as cutting off a node eliminates its entire descendance from the search tree. Reordering the children of a given node in a way to start exploring the most promising branches (a priori) generates more cut-offs.

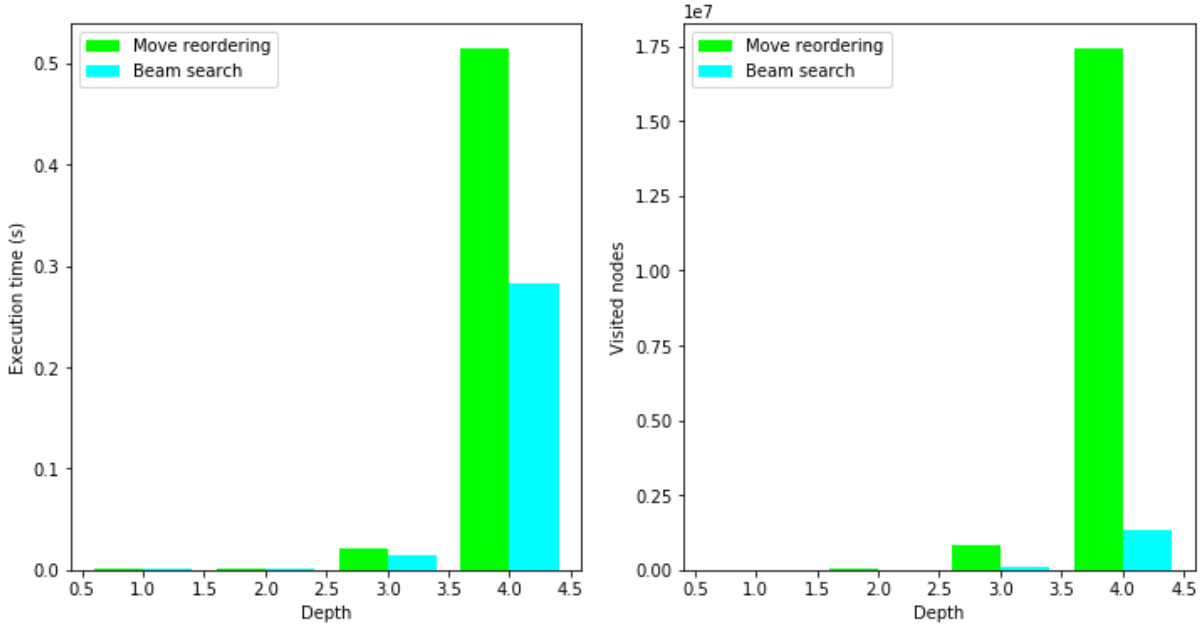
Simulation (Distributed Move reordering vs Simple Distributed)



Move reordering involves evaluating the heuristic function for each child of the current node and then sorting them based on the estimated value. Although this approach can result in more efficient pruning, it may also result in additional overhead due to the computation and sorting of each child node. In other words, move reordering can reduce the number of nodes visited but may increase the time required to evaluate each node.

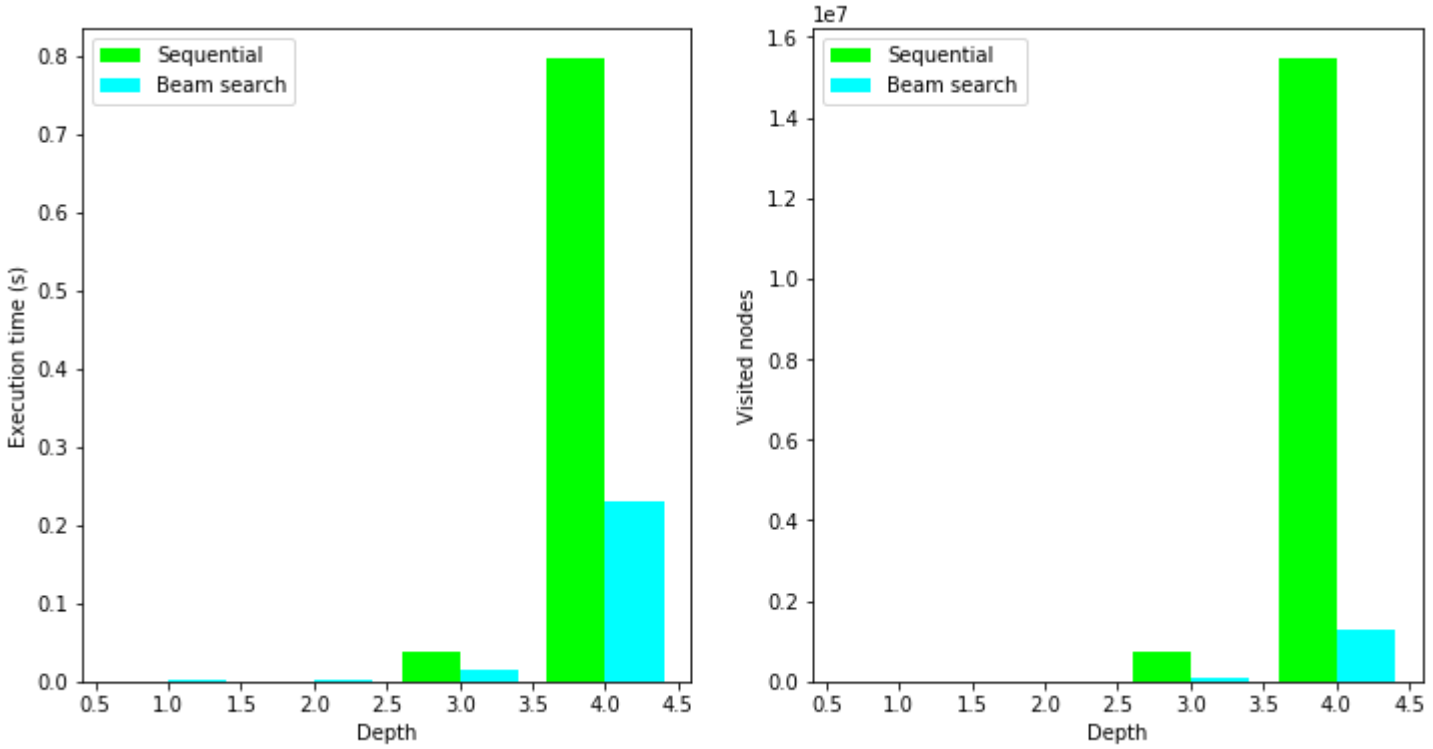
Beam Search:
The beam search method utilizes move reordering to focus only on the most promising nodes of the search tree. By sorting the children of a node according to their estimated potential, we can limit our exploration to a subset of the most promising child nodes while ignoring the less promising ones, which can speed up the search. However, the downside of using this approach is that we may miss exploring parts of the search tree that could lead to better moves.

Simulation (Beam Search vs Move reordering)



Using beam search technique with move reordering can reduce the execution time of the search algorithm by exploring only the most promising nodes in the search tree, while ignoring the less promising ones. This approach may lead to missing some good moves if the estimation function is not accurate. It is important to note that the estimation function must be good for this technique to work effectively. In our experiment, we observed a significant reduction in the number of visited nodes, as we only considered half of the child nodes in each minimax call.

Simulation (Beam search vs sequential)

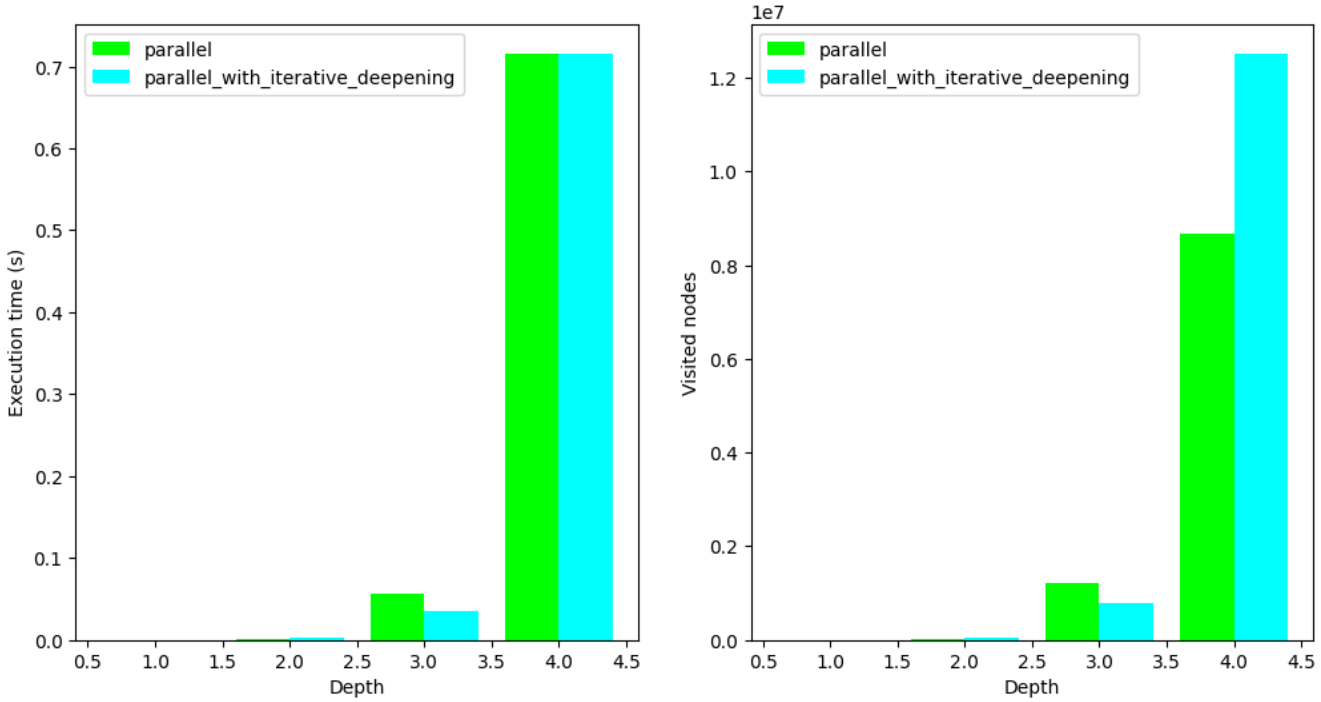


Beam search is a useful technique to explore the most promising nodes while reducing the number of nodes visited, but it heavily relies on the accuracy of the estimation function. A good estimation function should provide reliable evaluations of the board positions, so that the most promising moves are selected for further exploration. However, if the estimation function is inaccurate or biased, it may lead to suboptimal or

even bad moves being selected, which can be detrimental to the overall performance of the algorithm. Therefore, it is important to carefully design and test the estimation function to ensure its reliability and accuracy.

Iterative deepening:
Iterative deepening is a modified version of the minimax search algorithm that involves exploring all depths from level 1 up to a specified depth limit "d". After each exploration, the child nodes are reordered based on the value obtained from that exploration. This process may seem repetitive and time-consuming as it involves exploring the same nodes multiple times. However, the cost of expanding the same nodes repeatedly is negligible when compared to the exponential growth of the search tree.

Simulation (Iterative deepening vs distributed iterative deepening)



Although the execution time is clearly better, I am mainly referring to the number of explored nodes. It looks like the iterative deepening explored slightly less nodes than the regular minimax search.

Working of MPI in for Various Algorithms for Distributed Chess Engine :-

We have an initial chess board configuration given.
MPI enabled communication between processes or nodes in a parallel system, allowing these processes to coordinate their work and share data as needed.
Here are the steps involved in how MPI works:-

1. **Initialization**: In MPI, a program starts by initializing the MPI environment. This involves calling the MPI_Init function, which sets up the necessary data structures and resources for the MPI system to function.

2. **Process Creation**: Once the MPI environment is initialized, the program creates a number of processes or nodes, which will execute concurrently. Each process is assigned a unique rank, which is used to identify it during communication.

3. **Data Distribution**: The program then distributes data , which is initial Chess Board configuration , among the processes. This involves broadcasting the initial configuration to various slave processes by master process, which are then assigned to different processes for processing.

4. **Communication**: The processes then communicate with each other using a variety of MPI communication routines. These routines include point-to-point communication, where one process sends a message to another process, and collective communication, where a group of processes coordinate to perform a particular task.

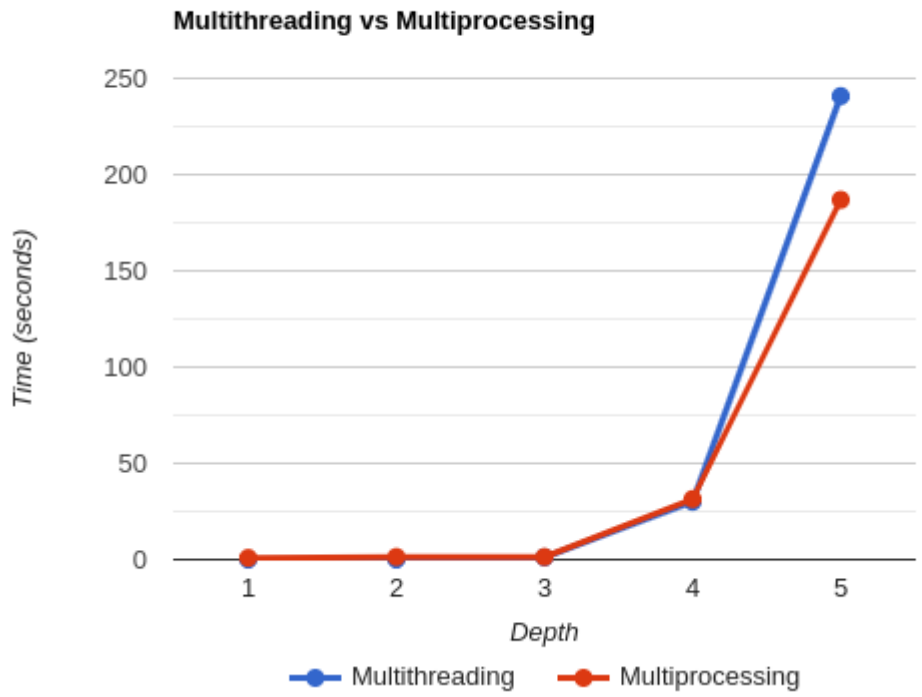
5. **Computation**: Each process performs computations on its assigned config.
   Here incase of Chess Engine, each process having the Chess board configuration processes it (Applies MinMax Algorithm and its Variations according to requirement).
   On processing their Chess config, they are now equipped with some game score value and corresponding optimal configuration (based on whether mode of play was MIN or MAX).
   The results of these computations are be stored locally with the processes.

6. **Synchronization**: As the processes communicate their respective scores and config to master process, for the purpose that to ensure that they are synchronized properly and lock, which allow processes to coordinate their actions and avoid race conditions.
   All the slave process sends their game score values along with corresponding Chess config to the Master Process.
   Master process then compares all these games score values and selects and saves best score value.
   This best score's corresponding Chess config is saved and again broadcasted to slave processes for further processing.

7. **Termination**: Once the computations are complete, the program terminates the MPI environment. This involves releasing any resources used by the MPI system and cleans up any remaining processes.

   In summary, MPI works by initializing the MPI environment, creating a set of processes, distributing config among these processes, performing computation and synchronization, and finally terminating the MPI environment when done.
   These steps allows Distributed Chess Engine to harness the power of parallel processing to solve this time consuming problem more quickly and efficiently

Comparison with Multithreading :



It appears that for depths 1 to 3, the parallel and distributed implementations have similar execution times, with the parallel implementation sometimes being slightly faster. However, for depth 4, the distributed implementation seems to perform slightly better than the parallel one.

For depth 5, there is a significant increase in execution time for both implementations, but the distributed implementation still appears to be faster than the parallel one, despite the longer execution time for both.

Overall, it seems that the distributed implementation has a clear advantage over the parallel one at larger depths, which is to be expected as it can distribute the workload among multiple machines, whereas the parallel implementation can only use multiple threads on a single machine.