Parallelizing a Chess Engine

Atharv Belagali
Information technology
National Institute of Technologyy
Karnataka
Surathkal,India
181IT208

Gagandeep KN
Information technology
National Institute of Technologyy
Karnataka
Surathkal,India
181IT215

Prithvi raj Patil
Information technology
National Institute of Technologyy
Karnataka
Surathkal,India
181IT234

***Abstract*—Computer AI is now ruling the 21st century and the centuries yet to come. Since parallel computing is in such a trend , it is natural that it complements heavy computation like AI. In this paper we implement an application of AI using parallel computing .We implement a parallel search algorithm named root-splitting.The main idea of this technique is to ensure that each node except for the root, is visited by only one processor.**

I. Introduction

Chess programming is regarded as one of the most classical challenges ever encountered in AI. It may seem easy at first glance but when we look profoundly, there exist 10 to the power of 120 possibilities. Which is a lot to comprehend and frankly difficult to enumerate and execute them.

Fortunately for us, there exist an algorithm which gives computers the ability to play chess, that is the Minimax algorithm. The minimax; from a given node in the search tree, computes all its children (that is the set of all the possible moves from the current configuration) and visits each child one by one, computing each time an evaluation function that gives away a kind of performance of that child. We repeat that recursively until we reach the leaves. At the end, the algorithm chooses the branch with the highest evaluation. However, the search tree being too large, we generally stop at a maximum depth, otherwise it would be computationally heavy or worse: impossible.

By parallelizing the search, we can compute independent branches simultaneously, gaining a considerable time that we could use to visit an extra layer.

Instead, we focused on optimizing it by parallelizing the computation of independent branches. We also use a bit of python script to display the difference between the sequential and parallel search graphically.

II. Objectives

- To understand and implement a chess engine sequentially and parallelly.
- To implement the chess game using a specific parallel search algo.(Root-splitting).
- To compare and prove the fact using visual graphics that parallel runs faster than sequential.

III. System Requirements

- Software requirements:
  Windows/Linux /macOS
- Hardware Requirements:
  i3 processor
  4GB RAM

IV. Literature Survey

1. Parallel Alpha-Beta Pruning of Game Decision Trees ;A Chess Implementation - Kevin Steele:

An AI powered program that plays chess either with itself or with an use. The most efficient method to traverse a minimax game tree is the alpha-beta pruning algorithm. This algorithm effectively prunes portions of the search tree whose leaf node scores are known a priori to be inferior to the scores of leaf nodes already visited. Unfortunately for parallel computing enthusiasts, the alpha-beta algorithm and its variations are inherently sequential. In fact, if great care is not given in parallelizing the algorithm, parallel performance can degrade well below the performance of the serial algorithm.

2. Massively Parallel Chess - Christopher F. Joerg1 and Bradley C. Kuszmaul:

Implementation of Chess game with different types of algorithms. Uses a lot of algorithms: Negmax search without pruning, Alpha-beta pruning etc.. And gives their analysis. Since the algorithms are so many , the debugging is going to be a problem and since parallel and sequential almost use the same code.

V. METHODOLOGY:

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!

Given the time allowed for this project and the relative simplicity of implementation, we came to choose this algorithm for our project. We used the openmp library to do multithreading for its simplicity and efficiency. The full implementation of the root splitting algorithm can be found on the repository. We'll just illustrate the key (changed) parts of the original C code here.

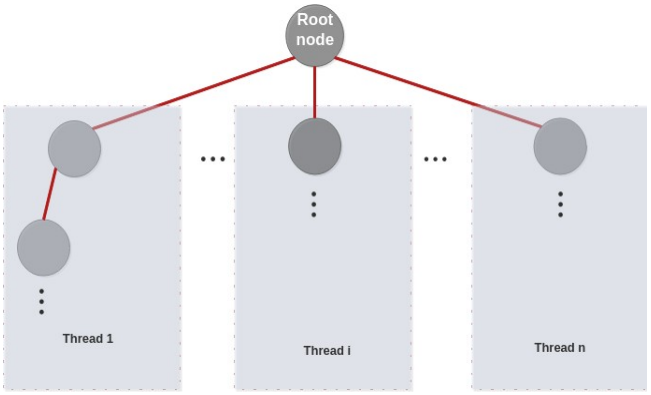

Figure 5.1 Root splitting (dividing the nodes into threads(or clusters))

We said that we parallelize at the root level, so for that we parallelize the for loop that iterates over the children of the root node for calling the minimax function on them. for that, we use the following two openmp directives:

#pragma omp parallel private (/*variables private to thread here */) #pragma omp for schedule (dynamic)

The two above directives need to be put right before the for loop to parallelize. In the first one we declared the variables that must be private to each thread. In the second directive we specified a dynamic scheduling between threads, meaning that if a thread finishes his assigned iterations before others do, he'll get assigned some iterations from another thread, that way making better use of the available threads.

Then we must ensure inside our for loop that after each minimax call returns, the thread enters in a critical section in mutual exclusion to compare (and modify) the best evaluation of the root node. We do that using the following directive:

#pragma omp critical { // Code here is executed in mutual exclusion }

All the code written inside this directive will be run by at most one thread at a time, ensuring thus the coherence of the value of our best evaluation variable.

VI. RESULTS :

We display two graphs that compares sequential and parallel parts : first graph plots every move on the chess board and the other displays total time using a bar graph respectively.

As the graph suggests the time taken by serial player at each step is significantly more than parallel player and also graph in Fig 6.2 and 6.5 Shows total time taken by serial is much more compared to parallel

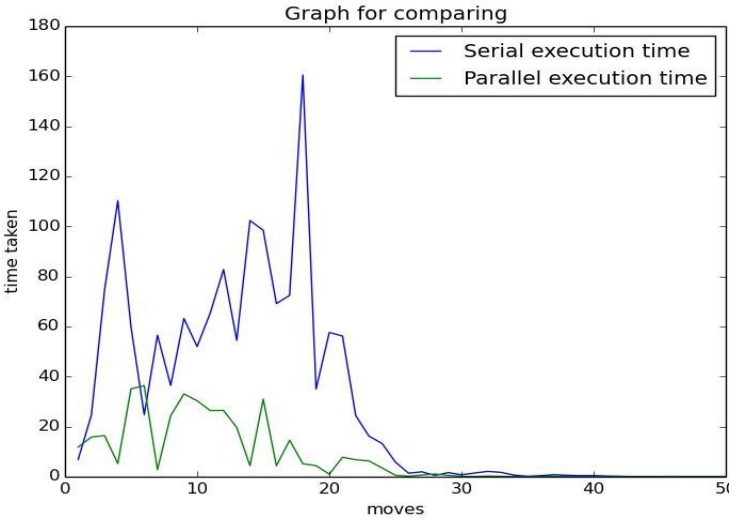

Fig 6.1 Graph showing the time taken by the player for every move till the end

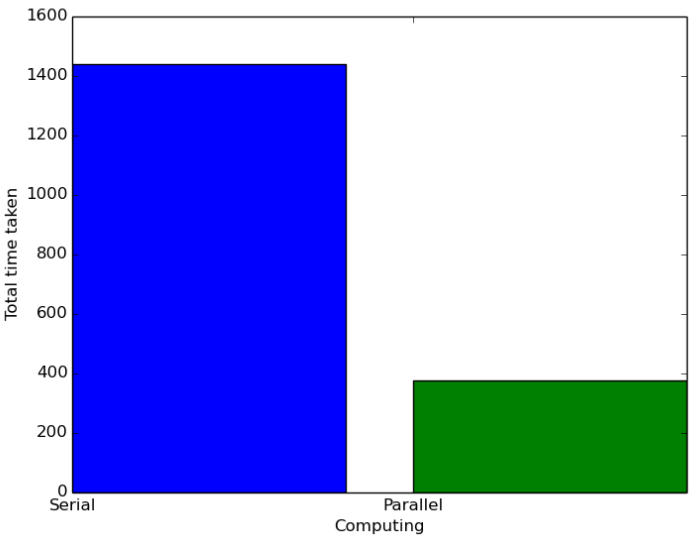

Fig 6.2 Bar graph indicating total time taken by each player by the end of the game

Now we have a display of how the pawns in the chess board are being played and in the end the sequential player wins respectively:

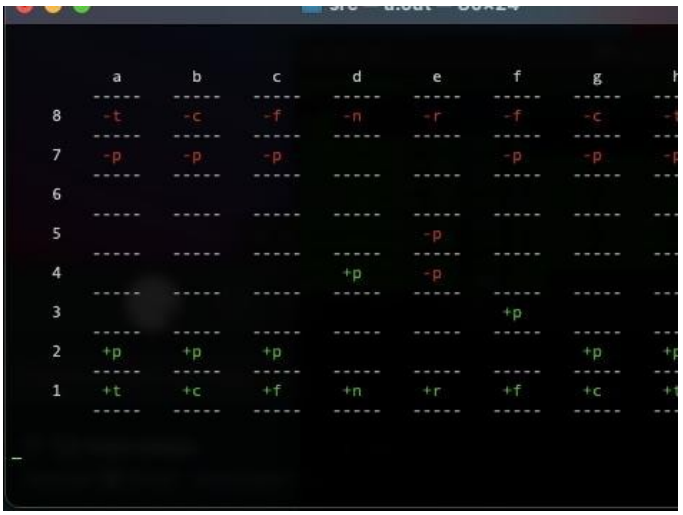

Fig 6.3 Screenshot of the game progress

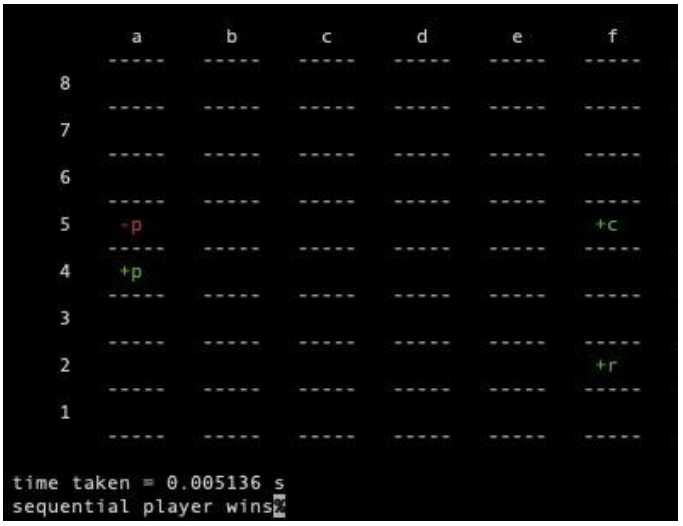

Fig 6.4 Screen shot of the output at the game declaring the winner

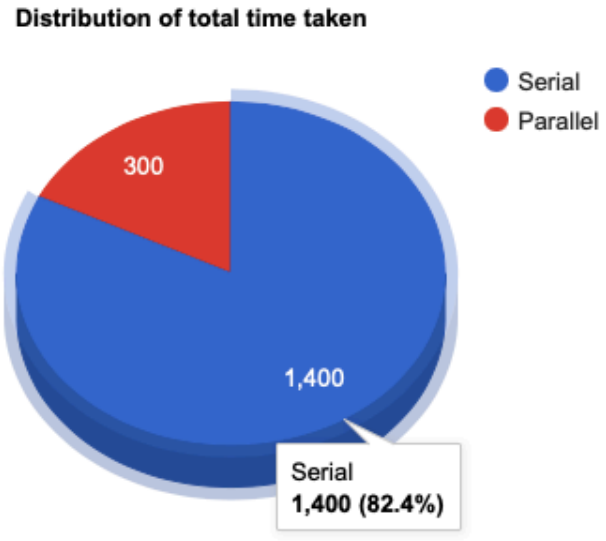

Fig 6.5 pie chart displaying the distribution of time in the game

VII.CONCLUSION

In this paper we have successfully implemented a chess engine using alpha beta pruning technique and compared and displayed the sequential and parallel times using visual graphs. But this engine only plays with itself (AI vs AI), we can improve upon it by adding a user vs AI in the future.

VIII.INDIVIDUAL CONTRIBUTION

- Gagandeep KN: Worked on dividing the children nodes into clusters to assign them to different threads and displacement functions, Minimax algorithm
- Atharv Belagali: Worked on handling the evaluation function in the parallel part and maintaining the local score and count of individual thread, Alpha-beta Pruning
- Prithvi: Worked on the the mutual exclusion part of the program for comparing the obtained results at the root node from all the threads to take the next best step, Plotting graphs and tables

IX. REFERENCE

1. http://supertech.csail.mit.edu/papers/dimacs94.pdf (paper[1])
2. https://students.cs.byu.edu/~snell/Classes/CS584/projectsF99/steele/report.html (paper[2])
3. Wikipedia (https://www.chessprogramming.org/Parallel_Search)
4. Minmaxalgo.(https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1- introduction/ #:~:text=Minimax%20is%20a%20kind%2 0of, %2C%20Mancala%2C%20Chess%2C%20etc.)