Parallel Chess AI Engine

Priyanshi Garg, Teddy Liang
May 04, 2024

1 Summary

Our project focuses on the development of a parallel chess AI engine, leveraging C++ and the OpenMP framework to accelerate the decision-making process in chess. We achieved 5.6x speedup on 8 cores.

2 Background

The main purpose of a chess engine is to be able to analyze a given chess board and find the best move. This is done by considering different board positions that are multiple moves into the future. If we want a chess engine to evaluate the best move by looking 4 or 5 moves deeper into the future, the number of possible board states grows exponentially as the depth increases. Thus the goal of a chess engine is to explore a vast array of potential moves and their resulting positions in a reasonable amount of time.

This task is managed by the combination of the minimax algorithm and alpha-beta pruning techniques. Minimax is a decision-making algorithm used in two-player games such as chess to find the optimal move by minimizing the maximum possible loss, assuming optimal play from the opponent. It explores a game tree by recursively evaluating potential future moves to a certain depth. Alpha-beta pruning enhances this by reducing the number of nodes evaluated in the minimax algorithm. It does this by introducing two values, alpha and beta, which represent the minimum score that the maximizing player is assured and the maximum score that the minimizing player is assured. If it becomes clear that a move will not improve these bounds, i.e. the maximizing player has a better option elsewhere, further exploration of that move can be cut off early (see Figure 1). This significantly reduces the search space and improves efficiency without affecting the outcome of the decision.

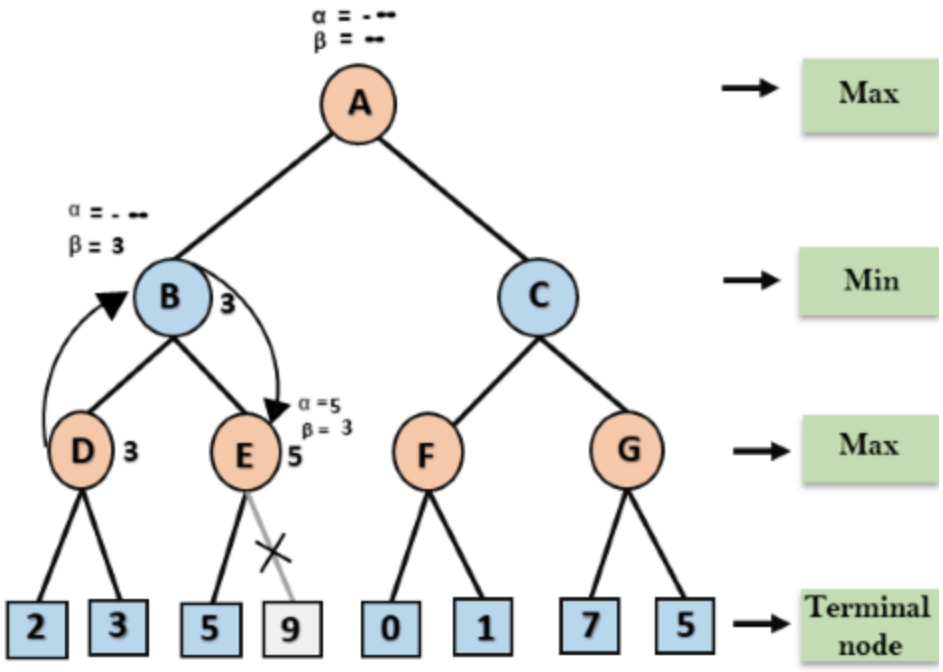


Figure 1: Alpha-Beta Pruning (Javapoint)

Key Data Structures: Aside from the bitboards utilized by the chess engine, the searching algorithm relies on essential data structures such as move lists and search stacks to traverse the game tree efficiently. Move lists store legal moves generated for a given position, facilitating move ordering and pruning strategies. Search stacks maintain information about the current ply and aid in managing recursive function calls during the search process.

Key Operations: The core operations of the the chess engine involve move generation, move ordering, and evaluation. Move generation constructs a list of legal moves for a given position, enabling the exploration of potential move sequences. Move ordering prioritizes moves based on heuristic criteria, such as captures and promotions, to optimize alpha-beta pruning efficiency. Evaluation computes the desirability of board states based on positional factors and material balance, guiding the search toward more promising branches of the game tree.

The main opportunity for parallelism is in the search for the best move. Board states are represented as position objects which contain all the information about the game state. The game tree, representing potential moves and their outcomes, is inherently expansive. Our approach involves dissecting this tree into multiple branches at the root level, each assigned to a separate thread for exploration. This division allows simultaneous traversal of

different parts of the game tree, significantly enhancing the breadth and depth of the search within the same computational time frame. By parallelizing the search, we dramatically increase the number of positions analyzed within a given time, directly contributing to more strategic and informed decision-making by the AI. This can significantly reduce the time needed to make an informed decision. It can also allow the chess AI to look farther into the future in the same amount of time compared to a sequential version, allowing it to choose better moves.

Because the value of a parent node cannot be determined until all its children are evaluated, the dependency of child nodes on parent nodes complicates parallelization, introducing synchronization points that can limit parallel efficiency. Furthermore, different branches of the game tree can have vastly different complexities and depths, leading to divergent execution paths among parallel threads. Some threads might quickly reach a pruning point or a shallow depth, while others might need to explore deeper, more complex branches, resulting in workload imbalance. This leads to challenges with mapping the workload effectively so that threads have similar amounts of work.

3 Approach

We utilized C++ and openMP to build our parallel chess engine. Since our central aim was studying parallelism in the search algorithm, we utilized a chess library “Libchess” to outsource complicated tasks like move generation and game state representation so we could focus on specifying the evaluation and search. We also utilized their testing suite to build our baseline sequential engine.

We employed two main approaches to parallelize the search algorithm:

3.1 Shared Hash Table

The shared hash table approach in parallel search involves utilizing a common hash table, also known as a transposition table, which can be accessed by multiple processes or threads concurrently across multiple processor cores or processors. This approach leverages shared memory to enable efficient communication and data sharing among the parallel search instances. Shared hash tables are implemented as dynamically allocated memory treated as a global array, with each entry storing information about positions and their corresponding evaluations.

Concurrency in accessing the shared hash table introduces challenges related to synchronization and data consistency. Concurrent writes and reads to the same memory address may lead to corrupt data, resulting in significant issues for the search process. To mitigate such problems, synchronization mechanisms such as locks are commonly employed. These locks ensure that only one thread can access a particular hash table entry at any given time, preventing data corruption and ensuring data integrity.

Various techniques have been proposed to implement shared hash tables efficiently in parallel search algorithms. One approach involves using atomic locks to synchronize access to hash table entries, ensuring mutual exclusion and preventing concurrent modifications. Another approach is to implement lock-less algorithms, such as the XOR technique, which eliminates the need for locks by exploiting bitwise operations to ensure data consistency.

In our implementation, we utilize a Transposition Table (TT), a global memory structure designed to store and retrieve information about previously encountered game positions. Each thread in our parallel search algorithm is tasked with exploring a distinct subset of moves from the current game position. Utilizing OpenMP we orchestrate concurrent execution of these threads, enabling them to independently traverse the game tree and evaluate potential moves. Critical sections are employed to synchronize access to shared resources, particularly the Transposition Table. This ensures that concurrent read and write operations do not compromise data integrity or lead to race conditions. Furthermore, when updating critical variables such as the best score or principal variation (PV), only one thread is permitted to modify these shared entities at a time. During execution, each thread recursively invokes the search function to explore deeper levels of the game tree from its assigned subset of moves. As threads complete their exploration, the results are collated to determine the optimal move and score, incorporating contributions from all threads.

We experimented with various scheduling policies as well as chunk size granularities and in our iterative procedure aimed to reduce the critical sections to reduce overhead due to synchronization especially when writing to the TT.

3.2 Root Splitting

This approach to parallelizing the search for the best move is implemented by dividing the decision tree’s root among several processors. This method leverages the power of multi-core processors to speed up the search process by exploring multiple branches of the tree concurrently, using algorithms like alpha-beta pruning to assess each move's implications efficiently. For example, one thread will work on the branch stemming from one move while another thread will work on the branch stemming from a different move.

This isis also achieved using openmp and C++ on the ghc machines. The algorithm would first generate moves from the given position object. These moves would then be be distributed to different processors in a dynamic schedule by openmp. Each of these cores would independently evaluate the board with its new move, performing the alpha-beta pruning algorithm. This method would allow for simpler implementation but at the cost of less speedup.

4 Results

4.1 Performance Metrics

For this project, we measured the performance of our parallel chess search algorithms primarily through two metrics: parallel speedup and nodes per second.
Parallel Speedup here is the ratio of the execution time of the search algorithm on a single thread to its execution time on multiple threads. This helps us understand the efficiency gains from parallel execution.
Nodes per second measures the number of positions or nodes the search algorithm explores in one second which gives us a good indication of search speed.
We also considered the evaluation score of the resulting move which tells us about the quality of the move suggested by the algorithm.

4.2 Experimental Setup

We conducted our experiments on GHC machines from 1-8 threads. We also experimented with using the PSC machines up to 64 threads. We utilized 20+ different initial positions representing a variety of different chess board configurations for testing.

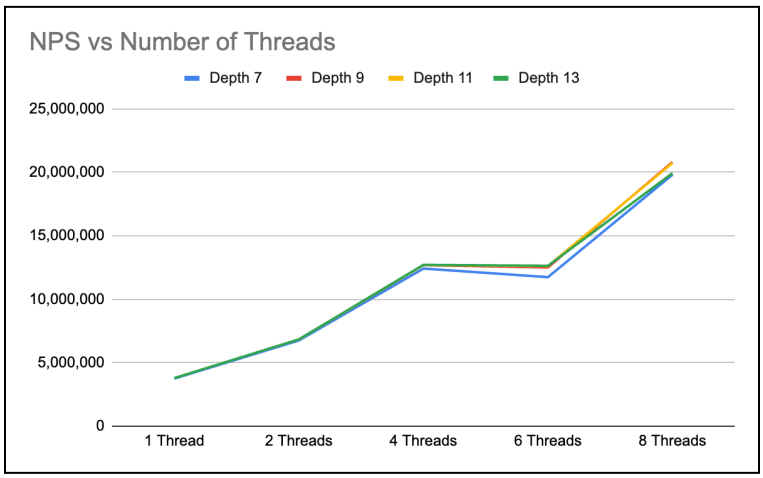
Problem Size
Problem size in our situation would be the depth of the search tree so we tested our configurations and solutions with varying depths. For example, a depth of 4 would mean in total 2 moves from the white side and 2 moves from the black side.

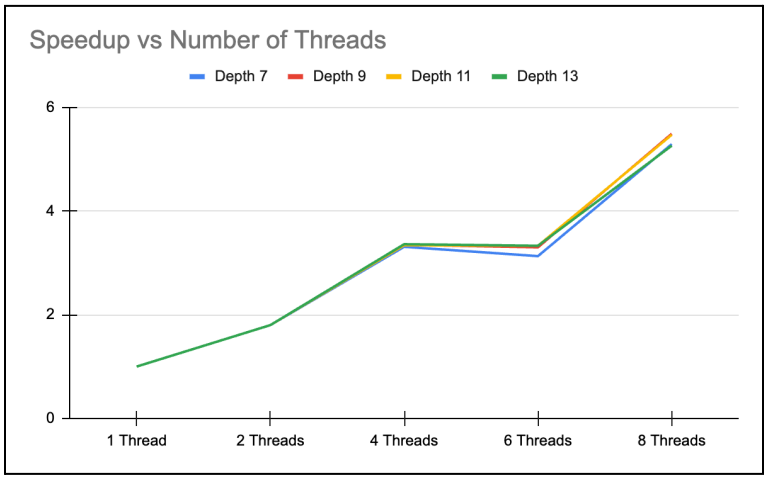
Chunk Size
For dynamic assignment policies, we experimented with varying chunk sizes testing how different task granularities affected performance.

4.3 Performance Analysis

4.3.1 Shared Transposition Table:

Results on GHC:





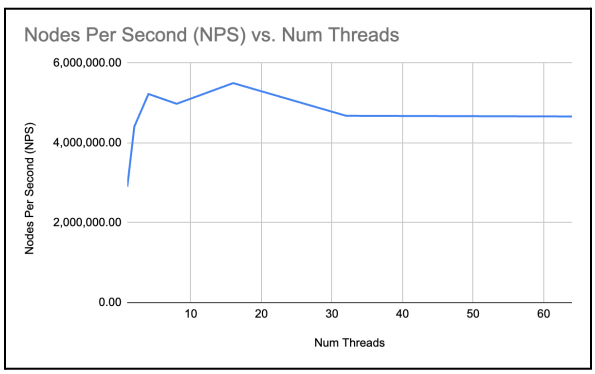
	1 Thread	2 Threads	4 Threads	6 Threads	8 Threads
Depth 7	1	1.8	3.31	3.13	5.29
Depth 9	1	1.8	3.35	3.3	5.49
Depth 11	1	1.8	3.34	3.33	5.47
Depth 13	1	1.8	3.36	3.33	5.26

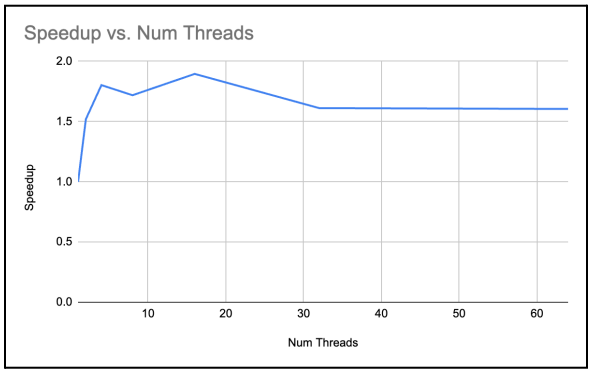
The performance analysis of the system as the number of threads increases shows a clear pattern of improvement in both Nodes Per Second (NPS) and speedup ratios. With the initial increase from 1 to 4 threads, there is a substantial boost in NPS and speedup, indicating that the system is efficiently leveraging the additional computational resources. This improvement suggests that the parallelization strategy effectively minimizes overhead from communication and synchronization among threads.

As the thread count further increases to 6 and 8, NPS continues to rise, reflecting the system's capability to handle higher loads and execute more operations per second. However, while the speedup also improves, the rate of increase in efficiency begins to plateau. This trend indicates that, although additional threads contribute positively to processing more nodes, the relative gain in speedup starts to diminish. This marginal reduction in speedup gains could be attributed to increased coordination costs and potential bottlenecks that become more pronounced at higher thread counts. Despite this, the overall performance metrics affirm that the system scales well with increased threading, consistently enhancing throughput and computational efficiency.

While testing with varying depths, we found that the performance does not change significantly, but there is a slight decrease in speed-up as we increase the depth. This observation suggests that the algorithm's efficiency remains relatively stable across different depths of search. However, as the depth increases, the workload becomes more complex, potentially leading to increased coordination overhead and reduced parallel efficiency.

Testing on PSC



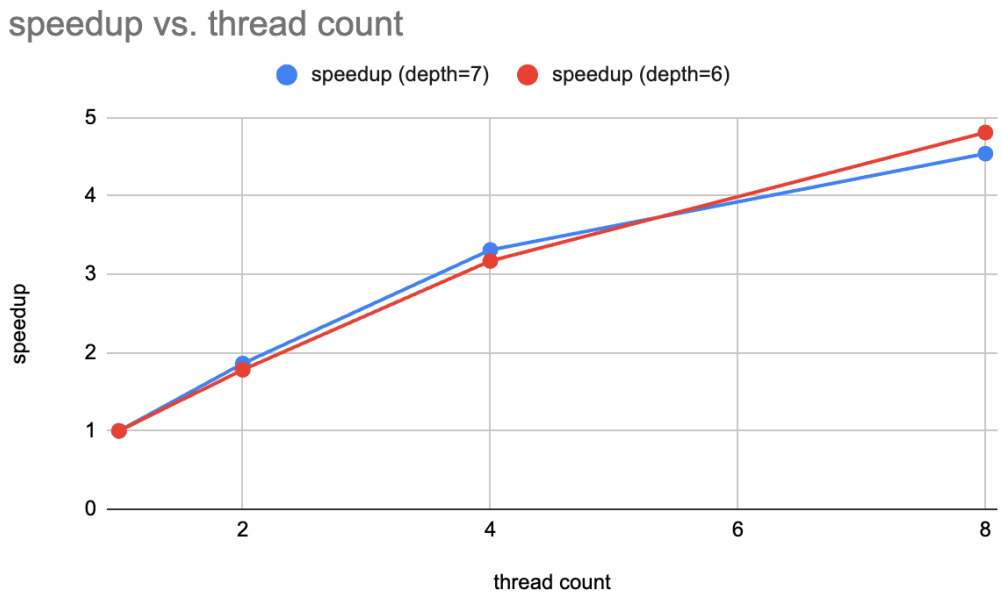


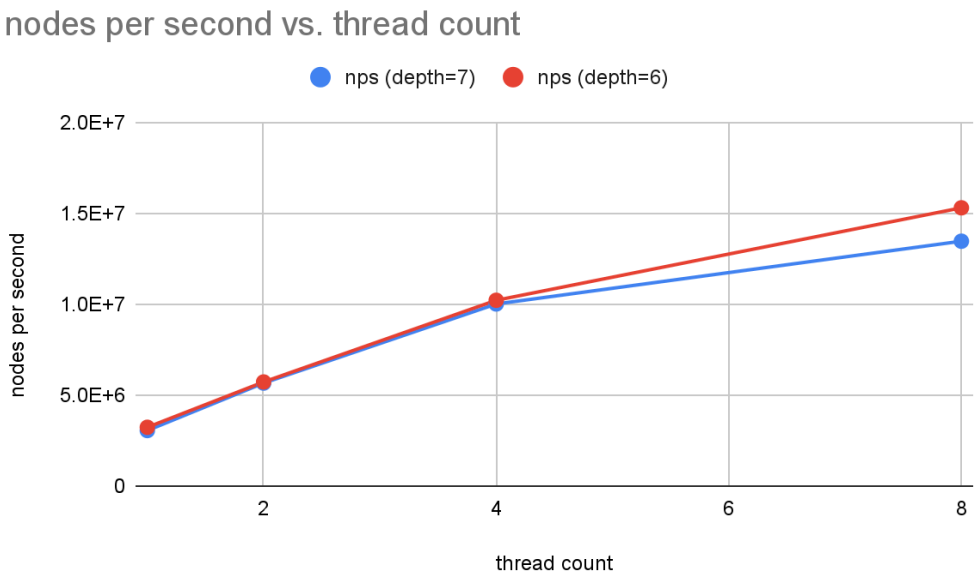
The performance analysis of the program on PSC machines reveals interesting trends in speedup as the number of threads increases. Initially, the speedup increases significantly from 1 thread to 4 threads, indicating effective parallelization and utilization of additional

computational resources. However, beyond 4 threads, the speedup experiences fluctuations, with a peak observed at 16 threads. Subsequently, the speedup decreases slightly as the number of threads continues to increase to 32 and 64. The poor performance on PSC below 8 threads is unexpected and is likely an issue with differences in hardware architecture or high load contention on PSC.

To identify bottlenecks and areas for optimization in our shared transposition table implementation we used "perf". The analysis reveals that a substantial portion of execution time, approximately 36.77%, is attributed to functions within the libgomp library, indicating the overhead incurred by managing OpenMP threads. Additionally, another 25.42% of the time is spent within libgomp which implies that the key reason for poor performance is parallelization overhead and thread management.

4.3.2 Root Splitting

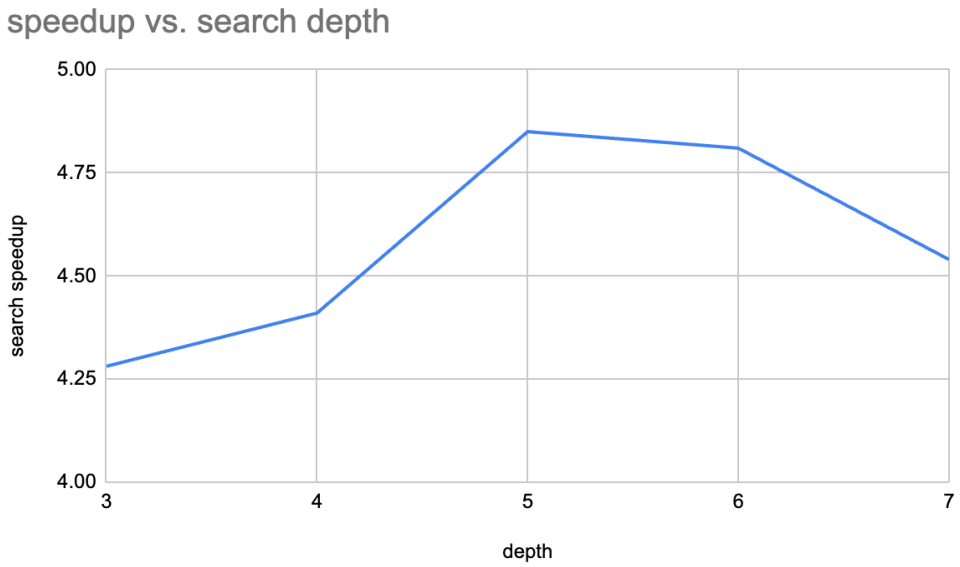




From the speedup graph, we can observe that both curves, representing depths 6 and 7, show an increase in speedup as the number of threads increases from 1 to 8. This trend is consistent with the expected behavior in parallel computing, where utilizing more threads can significantly reduce computation time by distributing the workload across multiple cores. The nodes per second graph also follows this trend which is in line with the results seen from the speedup. As more threads are used, more nodes are being processed at the same time which means higher nodes per second.

The graphs increase more noticeably when going from 1 to 4 threads. However, the rate of increase in speedup and NPS slows down slightly as more threads are added beyond 4. One reason for the diminishing speedup can be attributed to the overhead of managing more threads by OpenMP. Another reason for non-ideal speedups is root splitting inherently does not utilize as much parallelism as other techniques like the shared transposition table. Once processors get their new boards from the root, they work independently on their branches, without more parallelism.

We see that for 8 threads the NPS for a depth of 6 is slightly higher than the nps for a depth of 7. This is in line with the fact that the speedup for using a depth of 6 is slightly higher than the speedup for a depth of 7. This is likely because with a depth of 7, the effects of pruning the tree become more prominent. There are more opportunities for pruning which can lead to a larger load imbalance between the processors, leading to less speedup.



We also tested changing the search depth while keeping the thread count constant at 8 threads. From depths 3 to 5, an increase in speedup is seen, but from depth 5 to 7 it starts to decrease. The decrease in speedup past the depth of 5, as explained earlier, can be attributed to the effects of pruning being more prominent, leading to more load imbalance. The main reason for the increase is that at a depth of 3, the number of nodes being evaluated is exponentially smaller than the number of nodes being evaluated at higher depths. Therefore, a higher percentage of the is spent on thread management. With more nodes, there is more time being spent on the actually search and evaluation.

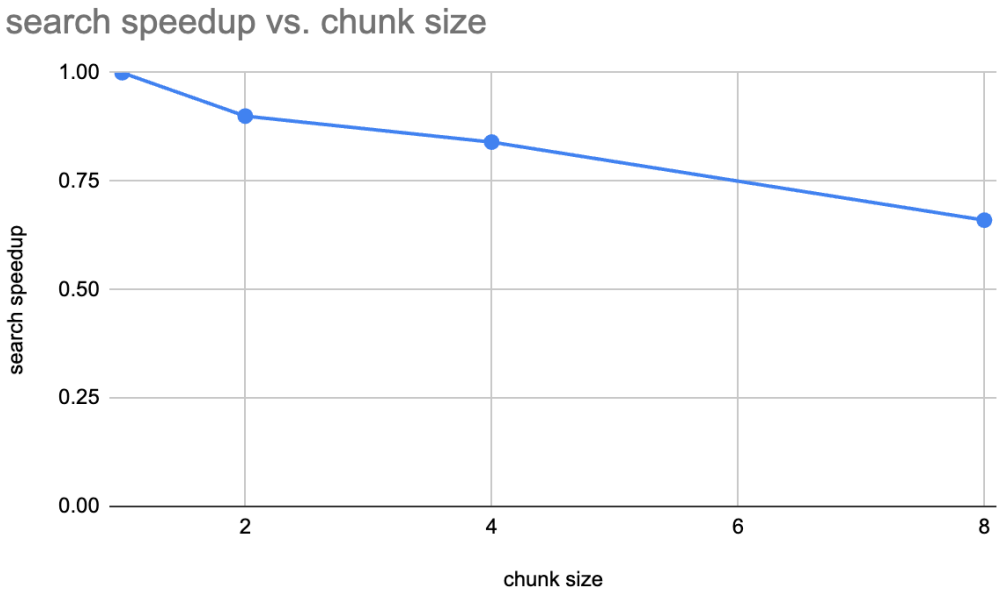
Perf report output from depth=3:

```
21.33%  timing-tests  timing-tests          [.] libchess::Position::make_move
20.95%  timing-tests  libgomp.so.1.0.0      [.] 0x00000000000207b8
15.60%  timing-tests  timing-tests          [.] eval::evaluate
 8.02%  timing-tests  timing-tests          [.] search::qsearch_impl
```

Perf report output from depth=5:

```
27.67%  timing-tests  timing-tests          [.] libchess::Position::make_move
19.65%  timing-tests  timing-tests          [.] eval::evaluate
10.81%  timing-tests  timing-tests          [.] search::SearchGlobals::stop
 9.59%  timing-tests  timing-tests          [.] search::qsearch_impl
```

We see that for depth=3, thread management (libgomp.so.1.0.0) is the second most time consuming function at about 21%, while in the output from when depth=5, the thread management is not taking up a significant portion of the total time as it is not in the top 4 functions.



Since this implementation used a for loop with a dynamic schedule, we experimented with different chunk sizes. We found a steady decline in search speedup as chunk size increased. This is likely due to a higher load imbalance. Most chess positions have between 20-40 legal moves, so changing to a larger chunk size does provide many benefits in terms of decreased thread management. It leads to more load imbalance since for example if there were 4 threads, 40 legal moves, and chunk size 8, one thread could be assigned 8 positions to work on at the end while the other threads sit idly. Thus, we stuck with a chunk size of 1.

5 Discussion

In our parallel chess AI engine, we explored two search parallelization strategies: Shared Transposition Tables and Root Splitting. Each approach provided unique benefits and challenges. Shared Transposition Tables were effective in reducing the search space and improving data reuse across threads. However, ensuring data integrity impacted scalability. The decision to use multi-core CPUs for our Parallel Chess AI Engine was strategically sound, given the nature of the application. This method showed effective speedups even at higher depths. Root splitting simplified implementation by distributing the decision tree's root across processors allowing independent branch evaluations. Both strategies had significant performance improvements with multiple threads with TT offering higher speedups due to more efficient parallelism in complex trees. Root Splitting is beneficial due to its lower complexity and is suitable for shallower trees.

CPUs excel in managing complex, interdependent tasks such as the ones required by our chess engine, which utilizes parallelism extensively to handle the decision trees and synchronization crucial for the minimax algorithm and alpha-beta pruning. CPUs offer superior control over task scheduling and concurrency, which is essential for efficient thread management and synchronization in our application. Unlike GPUs, which are optimized for independent parallel tasks, CPUs are better suited for applications like ours where tasks are highly interdependent and require frequent communication between threads.

6 References

- Kruskal, C., Rudolph, L., & Snir, M. (1988). Efficient Synchronization on Multiprocessors with Shared Memory. ACM TOPLAS, Vol. 10, No. 4.
- Hyatt, R., & Mann, T. (2002). A lock-less transposition table implementation for parallel search chess engines. ICGA Journal, Vol. 25, No. 1.
- Maier, T., Sanders, P., & Dementiev, R. (2016). Concurrent Hash Tables: Fast and General?(!). arXiv:1601.04017.
- https://www.javatpoint.com/ai-alpha-beta-pruning
- https://github.com/Mk-Chan/libchess
- https://github.com/Mk-Chan/LibchessEngine

7 Work Distribution

50 -50
Shared Transposition Tables - Priyanshi
Root Splitting - Teddy