Assignment 8 Report - Group 4

Abalone Implementation

In this task we implemented the search strategy and optimization for determining player moves in the two player board game Abalone. The implementation is based on the Alpha-Beta pruning search algorithm with additional optimization using parallelization, move caching, and iterative deepening.

8.1 Sequential Alpha/Beta

The sequential form of the algorihtm is implemented without further optimizations such as move caching or iterative deepening and is used to create a base case for further optimizations and timings. Alpha-Beta pruning is an algorithm used in adaptive game tree search algorithms that aims at identifying and pruning trivial subsections of the tree, thereby greatly decreasing the calculation time of the tree search. The algorithm maintains the values alpha and beta that represent the minimum and maximum score the maximum and minimum player are at least assured. A move is declared trivial if it falls outside the alpha-beta window and is thus strictly worse than a previous examined move. Such nodes do not have to be examined further as they cannot influence the final decision in any way.

8.2 Optimizations

Caching: PV First

The core idea of PV (principal variation) caching is to reuse the already calculated game tree from the last iteration. The principal variation describes the sequence of moves that was considered best by the search of the last iteration and is therefore expected to be played by a rational agent.
The performance of Alpha-Beta search is directly correlative to the ability to effectively prune large regions of the game tree, which can be achieved by quickly narrowing the available Alpha-Beta window. Firstly evaluating nodes that represent good moves in the game is an easy way to succeed at this task, since the minimum gains for both players are set rather high at the beginning of the search. In the same manner we employ the principal variation of the last iteration to firstly handle nodes in our search that have a high possibility of being one of the better moves in the current game tree.

Predicting the enemy move

In a similar manner to PV caching we tried to use the already calculated principal variation to correctly predict the other player's move. Once correctly predicted we could follow the cached move sequence and only continue searching the tree again when reaching the end of the cached principal variation. This idea was scrapped, however, when it became clear that a globally seen deeper game tree would not guarantee the cached principal variation to be the currently best possible move, and that in fact all other possible moves would have to be inspected with Alpha-Beta search in order to find the best move for the current iteration.

Young Brothers Wait Concept

The Young Brother Wait Concept (YBWC) is often used to optimize the traversal of game trees in Alpha-Beta search. "Brothers" are defined as sibling nodes that are connected to a common parent node. The concept delays the use of parallelism and multithreading in the search algorithm until the subtree of the oldest brother is available and did not result in a pruning of the sibling nodes. By waiting until subtrees are available that are relevant for the final result with a high probability, the algorithm prevents unnecessary resources to be spent on irrelevant subtrees and reduces the overall search overhead.
We use the concept to firstly sequentially traverse along the cached pv nodes of the last iteration until reaching the maximal defined search depth. Parallelism is then enabled for the sibling nodes of the principal variation moves in a recursive manner from the top of the tree to the bottom while maintaining and communicating alpha-beta information to the lower levels.

Iterative Deepening

Iterative Deepening uses the technique to cache already calculated move sequences to reorder the move iteration in Alpha-Beta search in such a way that iteratively incrementing the search depth results in a faster runtime than searching for the given maximal depth immediately. The algorithm start with a maximal depth of 1 and gradually increases the current maximal depth in each iteration, while preserving the principal variation of the last iteration in order to search these first in the current iteration. Besides the benefital move ordering and enhanced search time, the algorithm can thus be used to great effect to manage available computation time per turn, since even the results of the partial search can be accepted as a valid move.
However, in our implementation this technique did not result in an increase of computation efficiency. We propose that the reason for this behaviour could be found in the previously implemented concept of pv caching, which already reorders the nodes of the search tree in a beneficial tree.

8.3 Parallelization

We use the OpenMP task construct to handle parallelism and load balance in the search tree. In order to guarantee thread-safety with the given code, we create a private copy the board and evaluator for each task, thus enabling each spawned thread to use private local instances of the classes.

Initially, we started the parallelization based on the depth of the current search. However, this lead to load imbalance since some nodes may have a shallow search depths compared to its siblings. This Resulted in nodes waiting for children with deeper search depth to finish executing, while having threads of the already finished nodes idling unused. It is also hard to control the resource and memory usage this way because the amount of playable moves that can be taken at each node varies greatly between different tree nodes and espcecially between different depths.
We have experimented on our local machine for parallelization of depth greater than 2 eventually results in segmentation faults due to exhaustion of memory.

Given the need to control the resources and load balance of the parallel threads, it is intuitive to limit the number of simultaneous working parallel tasks by the number of cores we can utilize. In this implementation we first tried to dynamically limit the number of parallel tasks to at most 48 using a shared read-write lock. Any node that was not allowed to execute in parallel would start a serial search until one of the already running tasks would return and make room for a new parallel task. This dynamic allocation of parallel resources did not result in a speedup, however. We presusme this is due to the random nature of the allocation of the resouces, where the high overhead to start and manage the parallelism on not neccessarily beneficial segments of the tree outweights the gain from utilizing the multiple cores.
Instead, we switched to the idea of PV Splitting to control the load balance and parallelism on the search tree by spawning parallel tasks only on already established promising nodes.

PV Splitting

In order to optimize the usage of the available 48 cores and to reduce the overhead of spawning parallel tasks on nodes that are not neccessarily beneficial to the end result of the game tree search, we limit the use of parallelism on the direct children of the PV nodes that are explored first during the Alpha-Beta search. In the same idea as the YBWC we delay the launching of the parallel task until the subtree of the oldest brother has finished evaluating. This concept allows us to gain Alpha-Beta information faster along the considered optimal path and thus efficiently cutting nodes before searching on other potential moves. In order to further reduce the overhead of task generation we limit the generation of parallel tasks to nodes of lower depths in the search tree, that still have to do extensive searching.

Alpha/Beta Synchronization

During the parallel search, we allocate different local Alpha-Beta values for each node and running thread. This results in each of the thread only looking and updating their own alpha-beta values without considering their sibling's alpha-beta window. However, the original sequential version of alpha-beta pruning is intended to prune the younger siblings, given that the older sibling is finished with their parts of the tree. We optimized this by creating a new shared upper alpha variable and a shared lock array at specific synchronization points (principal variation nodes) that can be accessed by all children

during the PV Splitting. Each spawned node compares its current alpha or beta with the already found upper alpha at the synchronization point (nearest upper PV Node) and updates it's window accordingly. If a direct child of the synchronization node returns it's calculation the shared upper alpha value is updated and all corresponding children directly see the updated upper alpha value. Every running node can then terminate its search if it finds out that their part is no longer the most optimal hence further search is not needed.
We limited communication of the Alpha-Beta window to the synchronization points in order to reduce communication overhead on a shared variable.

Parallelize Push Moves

We also tried to parallelize the moves which would take out the opponent's piece (out move in template code) in order to see quickly if the resulting move yields a good final outcome. This way, it is expected that it would also prune the trees much faster since alpha-beta value gained by looking at this potential move is updated earlier. However, during testing we don't see any significant speed up by using this method.

Results

Measured results of the described optimizations, where Table 1 depicts the absolute achieved runtimes when executing the minimax search on the supermuc on one node with 48 threads on a maximal search depth of 6 and Table2 represents the achieved speedup. The sequential runtime is used as a base case for comparison when calculating the speedup. All depicted measurements were averaged from two consecutive moves played by two different players using the minimax search.
As mentioned above, we can see that the dynamic allocation of parrallel ressources introduces a massive overhead to the search algorithm, while PV manages the spawned tasks in a beneficial way. Compared to Loop Iteration simple PV Caching is superior, especially when playing mutliple consecutive moves. The reason for this behaviour might lay in the inability of Loop Iteration to use the previous cached principal variation to it's fullest potential.
Thus, we receive the best runtime on the supermuc when combining Alpha-Beta search with PV Caching, as well as PV Splitting.

Table 1: Comparisons of the achieved runtime in seconds when using the different optimization techniques on the starting and midgame board with a searchdepth of 6.

Optimization	**Start Position (s)**	**Midgame Position (s)**
Sequential	10.2	22.8
No Caching, dynamic Allocation	115.0	$>$ 120.0
No Caching, PV Splitting	3.0	7.0
PV Caching, PV Splitting	2.0	5.8
Loop Iteration, PV Splitting	2.8	6.7
PV Caching, PV Splitting, Push	2.0	5.8

Table 2: Comparisons of the achieved speedup when using the different optimization techniques on the starting and midgame board with a searchdepth of 6.

Optimization	**Speedup Start**	**Speedup Midgame**
Sequential	0	0
No Caching, dynamic Allocation	-11.2	$<$ -5.2
No Caching, PV Splitting	3.4	3.2
PV Caching, PV Splitting	5.1	3.8
Loop Iteration, PV Splitting	3.6	3.4
PV Caching, PV Splitting, Push	5.1	3.8