Final Report on Parallel Optimizations of Alphabeta Strategy

Group 1: Tiannan Zheng, Ali İhsan Arslanpay, Ahmet Burak Temurer

July 8, 2024

Contents

1 Introduction

This report documents the process of optimizing the Alphabeta strategy using parallel computing techniques. The primary objective was to improve the performance of the algorithm by leveraging multi-core processors through OpenMP.

2 Parallel Optimization

To optimize the Alphabeta strategy, we employed parallel computing techniques using OpenMP. The primary goal was to divide the workload among

multiple threads to speed up the execution.

2.1 Approach

We identified the most time-consuming parts of the algorithm and parallelized the move generation and evaluation processes using OpenMP directives. The key function for this optimization is the `minimax_openmp` function.

Listing 1: Parallelized Minimax Function

```
int MinimaxStrategy::minimax_openmp(int depth, Board board)
{
    int best_evaluation_value = MIN_EVAL;

    MoveList list;
    board.generateMoves(list);
    int total_moves = list.getLength();

    #pragma omp parallel for schedule(dynamic, 1)
    shared(best_evaluation_value) firstprivate(board)
    for (int i = total_moves-1; i >= 0; --i) {
        ...

        #pragma omp critical
        {
            if (eval > best_evaluation_value) {
                best_evaluation_value = eval;
                _bestMove = move;
            }
        }
    }

    return best_evaluation_value;
}
```

2.2 Implementation Details

The `minimax_openmp` function is the core of our parallel optimization. Below, we discuss the key components of the implementation:

Parallel Region We use the `#pragma omp parallel for` directive to parallelize the for loop that iterates over all possible moves. This directive

tells the compiler to distribute the iterations of the loop among multiple threads.

Dynamic Scheduling The `schedule(dynamic, 1)` clause is used to distribute the iterations dynamically among the threads. This allows for better load balancing, as each thread will take the next available iteration when it finishes its current one.

Shared and Private Variables

- **shared(best_evaluation_value):** The best evaluation value found so far is shared among all threads.
- **firstprivate(board):** Each thread gets its own copy of the board to work with, initialized with the value of the board at the start of the parallel region. This prevents data races on the board object.

Adaptive Depth Search Adaptive depth search is a dynamic search strategy commonly used in various computational problems, such as game tree evaluation or AI decision-making processes. Unlike traditional fixed-depth search algorithms that explore all nodes to a predetermined depth, adaptive depth search adjusts the depth based on specific criteria or heuristics. For instance, the depth can be modified according to the remaining time, the complexity of the current state, or evaluation values. This approach allows for more efficient resource allocation, focusing computational power where it's most needed, and can lead to faster and more effective solutions. By dynamically adjusting the search depth, adaptive depth search balances thoroughness with efficiency, improving performance in scenarios where computational resources or time are limited.

Below is the implementation we used for adaptive depth search in our project.

Listing 2: Adaptive Depth

```
1 if (remaining_time_secs < 5) {
2     adaptive_search_depth = 3;
3 } else if (remaining_time_secs < 10) {
4     adaptive_search_depth = 4;
5 } else if (best_evaluation_value < -700) {
6     adaptive_search_depth = 6;
7 } else if (remaining_time_secs > 20
8 best_evaluation_value < -500) {
```

```
9       adaptive_search_depth = 6;
10  } else {
11      adaptive_search_depth = 5;
12  }
```

Critical Section The `#pragma omp critical` directive is used to protect the critical section where the best evaluation value is updated. This ensures that only one thread at a time can update the shared variable `best_evaluation_value`, preventing data races.

2.3 Challenges and Solutions

During the parallelization process, we encountered several challenges, including data races and synchronization issues. These were addressed by carefully managing shared and private resources and using OpenMP directives like `critical` to ensure thread safety.

3 Performance Evaluation

To evaluate the impact of the optimizations, we conducted a series of performance tests comparing the execution times of the sequential and parallel implementations.

3.1 Methodology

The tests were conducted on a single node of SuperMUC-NG, and we compared the effect of number of threads on the speedup.

3.2 Results

Figure 1 shows the results regarding the speedup value along with the number of threads.

The results obtained with 1 thread represent the base case for comparison. The search depth is adaptively updated based on the remaining time and the best evaluation value.

3.3 Analysis

The parallel implementation significantly reduced the execution time, especially for the start and end positions. Though, it can be seen that for the

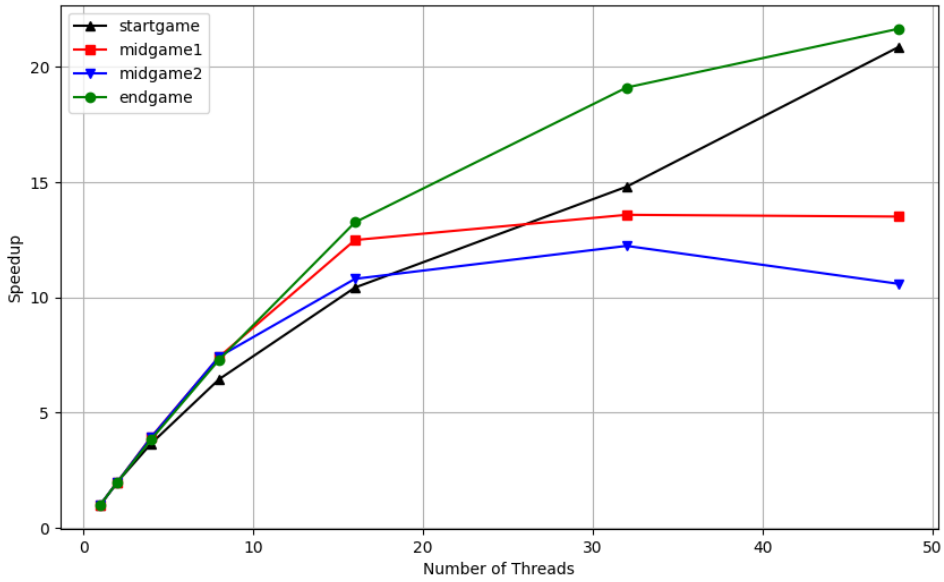

Figure 1: Speedup vs. Number of Threads for Different Game Phases in Abalone

middle parts of the game, we don't see any important difference for higher threads. The results indicate a clear performance improvement, demonstrating the effectiveness of the parallelization.

4 Conclusion

The parallelization of the Alphabeta strategy using OpenMP resulted in a substantial performance improvement. Future work could explore further optimizations and parallelization techniques.