Master’s Internship

Efficient Programming of Multi-Core Systems and Supercomputers

Final Report – Abalone

Submitted by: Rajpreet Singh

Matriculation Number: 03742970

MSc. Computational Sciences and Engineering

Table of Contents

1. Introduction

The code implements a Minimax strategy with alpha-beta pruning for Abalone game using OpenMP for parallel processing. The MinimaxStrategy class inherits from the SearchStrategy base class and is designed to search for the best move in a game by evaluating possible moves to a certain depth, recursively considering opponent's responses, and using parallelism to improve performance.

The MinimaxStrategy class inherits from the SearchStrategy class and overrides the searchBestMove method to implement the Minimax algorithm with alpha-beta pruning and OpenMP parallelism. The core of the implementation is the 'doMinMaxSearch' function, which performs a recursive Minimax search with alpha-beta pruning and OpenMP parallelism. The 'doMinmaxSearchSeq' function performs the Minimax search sequentially. It evaluates all possible moves, recursively calls itself for the subsequent depth, and performs the pruning. The 'doMinMaxSearch' function is the parallelized version of the Minimax search. It utilizes OpenMP tasks to parallelize the recursive search process.

2. Some key Feature about the implementation:

- **Move Generation and Evaluation:** The function generates all possible moves and evaluates them.

- **Alpha-Beta Pruning:** Implements alpha-beta pruning to cut off branches that do not need to be evaluated.

- **Parallelism with OpenMP:** Utilizes OpenMP tasks to parallelize the search. The #pragma omp task directive is used to create tasks for each move at certain depths.

- **Locks for Synchronization:** Uses OpenMP locks (omp_lock_t) to ensure safe updates to shared data like upperAlphaArray and the best move list (mList).

3. Parallelization Strategy:

- **Task Parallelism:** The algorithm employs task parallelism by creating tasks for evaluating moves at certain depths. This allows different parts of the game tree to be explored concurrently.

- **Granularity Control:** The createThread boolean controls whether a new thread (task) should be created based on the depth of the search and the proximity to the maximum depth.

- **Synchronization:** OpenMP locks are used to prevent race conditions when updating shared variables such as maxEval, alpha, and beta.

4. OpenMP Parallelization:

The MinimaxStrategy class utilizes several key OpenMP directives to parallelize the Minimax search algorithm. Here, #pragma omp parallel creates a team of threads, and #pragma omp single ensures that only one thread executes the doMinMaxSearch function call, but the tasks within that function can be parallelized.

```
void MinimaxStrategy::searchBestMove() {
    // Initialization code
    omp_set_num_threads(48); // Set the number of threads

    #pragma omp parallel
    {
        #pragma omp single
        test = doMinMaxSearch(0, *_board, *_ev, newMList,
-15000, 15000, 0, false, SearchStrategy::_maxDepth);
    }
    // Finalization code
}
```

The core parallelism is achieved using the #pragma omp task directive which defines a task to be executed by a thread.

```
if (createThread) {
    #pragma omp task firstprivate(m, depth, board, maxEval, evaluator, depthOfPv, pushParallel)
    {
        Move newTaskMList[10]; // Children write here
        board.playMove(m);
        int eval;
        if (depth + 1 < curMaxdepth) {
            eval = -doMinMaxSearch(depth + 1, board, evaluator, newTaskMList, -beta, -alpha, depthOfPv,
pushParallel, curMaxdepth);
        } else {
            eval = evaluator.calcEvaluation(&board);
        }
        board.takeBack();

        // Synchronization and update logic
        omp_set_lock(&(lockArray[depthOfPv]));
        if (eval > *maxEval) {
            *maxEval = eval;
            newTaskMList[depth] = m;
            foundBestMove(depth, m, eval);
            for (int d = depth; d < curMaxdepth; d++) {
                mlist[d] = newTaskMList[d];
            }
            if (depth == 0) {
                _currentBestMove = m;
            }
        }
        omp_unset_lock(&(lockArray[depthOfPv]));

        // Alpha-beta updates
    }
}
```

The task synchronization is done using #pragma omp taskwait, which waits for the completion of all tasks created by the current task. This directive ensures all tasks to complete before proceeding.

To handle synchronization and prevent race conditions when accessing shared resources, OpenMP locks are used as shown below.

```
omp_lock_t lockArray[10];

for (int i = 0; i < 10; i++) {
    omp_init_lock(&(lockArray[i]));
}

// Inside the parallel region
omp_set_lock(&(lockArray[depthOfPv]));
// Critical section
omp_unset_lock(&(lockArray[depthOfPv]));

// After parallel region
for (int i = 0; i < 10; i++) {
    omp_destroy_lock(&(lockArray[i]));
}
```

Locks are initialized before the parallel region, used to protect critical sections of code, and destroyed after the parallel region.

5. MPI Parallelization

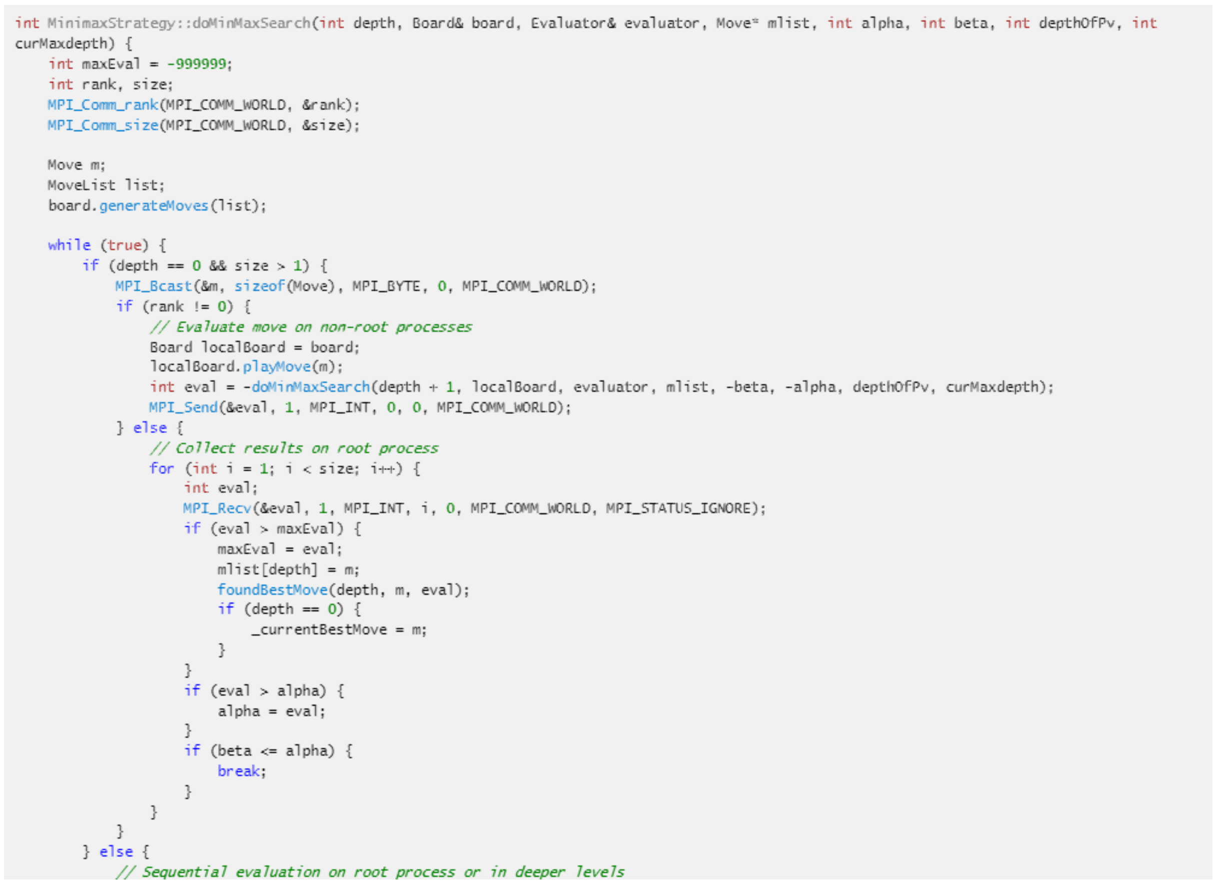
In another implementation, The MinimaxStrategy class leverages MPI to parallelize the Minimax search algorithm, distributing the workload across multiple processes in a distributed memory environment. MPI is initialized and finalized using MPI_Init and MPI_Finalize, typically outside the class definition but necessary for any MPI-based application. The rank of the current process and the total number of processes are obtained using MPI_Comm_rank and MPI_Comm_size. MPI_Bcast is used to broadcast a move from the root process (rank 0) to all other processes.

```
MPI_Bcast(&m, sizeof(Move), MPI_BYTE, 0, MPI_COMM_WORLD);
```

MPI_Send and MPI_Recv are used for point-to-point communication, allowing non-root processes to send their evaluation results back to the root process as shown below.

```
MPI_Send(&eval, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&eval, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

The parallel Minimax search function distributes the work across multiple processes.



```
int MinimaxStrategy::doMinMaxSearch(int depth, Board& board, Evaluator& evaluator, Move* mlist, int alpha, int beta, int depthOfPv, int
curMaxdepth) {
    int maxEval = -999999;
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    Move m;
    MoveList list;
    board.generateMoves(list);

    while (true) {
        if (depth == 0 && size > 1) {
            MPI_Bcast(&m, sizeof(Move), MPI_BYTE, 0, MPI_COMM_WORLD);
            if (rank != 0) {
                // Evaluate move on non-root processes
                Board localBoard = board;
                localBoard.playMove(m);
                int eval = -doMinMaxSearch(depth + 1, localBoard, evaluator, mlist, -beta, -alpha, depthOfPv, curMaxdepth);
                MPI_Send(&eval, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            } else {
                // Collect results on root process
                for (int i = 1; i < size; i++) {
                    int eval;
                    MPI_Recv(&eval, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    if (eval > maxEval) {
                        maxEval = eval;
                        mlist[depth] = m;
                        foundBestMove(depth, m, eval);
                        if (depth == 0) {
                            _currentBestMove = m;
                        }
                    }
                    if (eval > alpha) {
                        alpha = eval;
                    }
                    if (beta <= alpha) {
                        break;
                    }
                }
            }
        } else {
            // Sequential evaluation on root process or in deeper levels
```

The searchBestMove function initiates the search, differentiating behavior based on the process rank.

```
void MinimaxStrategy::searchBestMove() {
    reachedBottom = false;
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        if (mList[0].type != Move::none)
            _inPv = true;

        Move newMList[10];
        int maxEval;
        maxEval = doMinMaxSearch(0, *_board, *_ev, newMList, -15000, 15000, 0, SearchStrategy::_maxDepth);
        printf("maxEval = %d \n", maxEval);
        _bestMove = _currentBestMove;

        printf("Output Principal Variation:\n");
        for (int i = 0; i < SearchStrategy::_maxDepth; i++) {
            mList[i].print();
            printf("\n");
        }
    } else {
        while (true) {
            Move m;
            MPI_Bcast(&m, sizeof(Move), MPI_BYTE, 0, MPI_COMM_WORLD);
            if (m.type == Move::none)
                break;

            Board localBoard = *_board;
            localBoard.playMove(m);
            int eval = -doMinMaxSearch(1, localBoard, *_ev, nullptr, -15000, 15000, 0, SearchStrategy::_maxDepth);
            MPI_Send(&eval, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
}
```

6. Performance Considerations:

- Scalability: The implementation is designed to scale with the number of available threads. By setting the number of threads with omp_set_num_threads, it allows efficient utilization of multi-core processors. MPI excels in distributed memory systems, allowing the application to scale across multiple nodes in a cluster. This can provide significant performance benefits for large-scale problems where shared memory parallelism is insufficient.
- Load Balancing: The use of OpenMP tasks helps in dynamically balancing the load. Tasks are created for deeper levels of the tree where more computation is required, thus ensuring that all threads are effectively utilized. Effective load balancing in MPI requires careful distribution of tasks. Imbalanced workloads can lead to idle processes and inefficient utilization of computational resources.
- Synchronization Overhead: While locks ensure data consistency, they can introduce overhead. The design minimizes lock contention by only locking when necessary (e.g., updating the best move and alpha values).
- Memory Usage: The implementation uses arrays to store moves and evaluation results, ensuring efficient memory usage. Care is taken to avoid excessive memory allocation and deallocation within the recursive search.

7. Usage and Output:

The searchBestMove method initializes the search, sets up OpenMP locks, and starts the parallel search using the #pragma omp parallel and #pragma omp single directives.

Output:

- Best Move: The best move found is stored in `_currentBestMove` and `_bestMove`.
- Principal Variation: The principal variation is printed after the search completes.

8. Conclusion:

The OpenMP implementation of the Minimax algorithm in the MinimaxStrategy class effectively leverages parallelism to improve performance. The use of task parallelism, synchronization mechanisms, and careful management of shared data ensures both efficiency and correctness. This approach is suitable for high-performance game-playing AI on multi-core systems, allowing for deeper and faster exploration of the game tree.

9. Further Work:

- **<u>Dynamic Task Creation</u>:** Further improvements could include dynamic adjustment of task creation thresholds based on runtime metrics.
- **<u>Hybrid Parallelism</u>:** Combining OpenMP with other parallelization techniques (e.g., MPI) could further enhance performance on distributed systems.
- **<u>Optimizations</u>:** Additional optimizations like move ordering and advanced pruning techniques could be integrated to reduce the search space and improve efficiency.