2. Lab Report for Abalone Parallelization

Group 7: Jinwen Pan, Mingshuai Li, Jingtian Zhao

2.1 Parallelization Strategy and Scheme

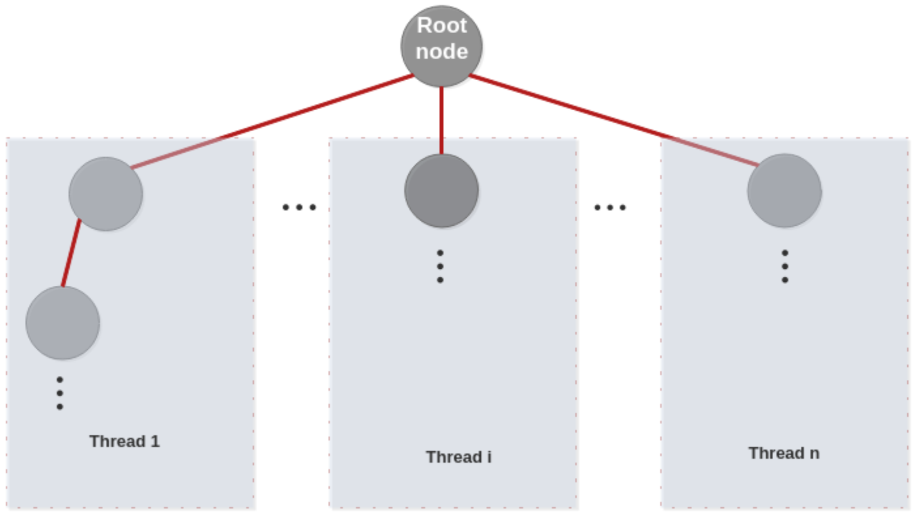


Figure 2.1: Parallelization Strategy of Alpha-Beta Search[1]

OpenMP parallelization is exclusively utilized in the program. OpenMP is suitable for shared memory systems. When there is only one node, using OpenMP allows for utilizing multiple threads on the node to execute tasks in parallel, thereby improving performance. On the other hand, MPI is a standard for communication and parallel computing among multiple nodes, designed for distributed memory systems. On a single node, compared to OpenMP, using MPI does not result in significant performance improvements, and the more complex communication functions in MPI may even introduce additional overhead. Therefore, using the fork-join model in OpenMP, which is more consistent with the structure of the search tree, not only benefits from parallelization but also largely avoids the need for program restructuring when using MPI, as well as the more complex partitioning of the problem domain.

As illustrated in Fig 2.1, when it comes to the movement in each turn, the various possible moves are assigned to multiple threads, and each thread performs a sequential alpha-beta pruning search on the assigned sub-tree. Specifically, `#pragma omp parallel for` parallelizes the for loop in the sequential program that iterates over each possible move in every turn. We fully utilize the 48 cores available on a single node and disable the automatic adjustment of the number of threads during program execution. This approach also avoids potential

[1]https://github.com/hennimohammed/parallel_chess_engine

performance losses that may be caused by hyper-threading. Regardless of the parallel strategy employed, we cannot explicitly know the specific workload of each iteration in advance. However, it can be determined that due to pruning, the workload for each iteration is likely to be different. Therefore, instead of static scheduling, dynamic scheduling with a chunk size of 1 is used to avoid unnecessary synchronization waits that may arise from workload imbalances. Since, on average, there are around 60 possible moves per turn, 1 is almost the only viable choice for the chunk size. Otherwise, in the initial round of iteration scheduling, some threads would remain idle. Additionally, the entire layout of the board before each move in every turn is passed as a private variable to each thread to avoid data racing. Similarly, `#pragma omp critical` ensures that shared variables related to moves and evaluations are updated exclusively by individual threads when necessary. Finally, `reduction` is used to aggregate the total number of evaluations from each thread.

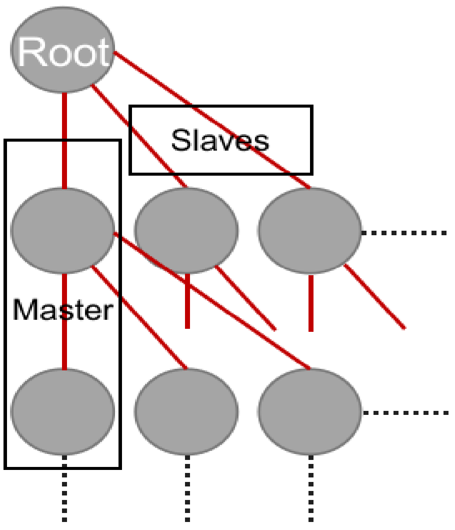


Figure 2.2: Master-Slaves Parallelization Strategy

However, the drawback of such a parallelization strategy is obvious. Because the alpha and beta factors cannot be propagated through the root node to other branches, it results in certain branches that should have been pruned being additionally evaluated. We have also investigated a theoretically superior parallelization strategy, known as the master-slaves strategy, as shown in Fig 2.2. The master thread initially searches the leftmost branch of the tree to obtain better initial alpha and beta bounds. Afterward, the master thread primarily remains idle and is responsible for distributing work to the slave threads at each level of the tree. Theoretically, this scheme makes better use of the propagation of alpha and beta factors throughout the entire search tree, resulting in additional pruning. However, maybe due to its more complex implementation, it does not provide significant performance improvements for our application. Therefore, we opt to continue using the parallelization strategy mentioned earlier.

2.2 Basic Underlying Search Algorithm

2.2.1 Alpha-Beta Pruning

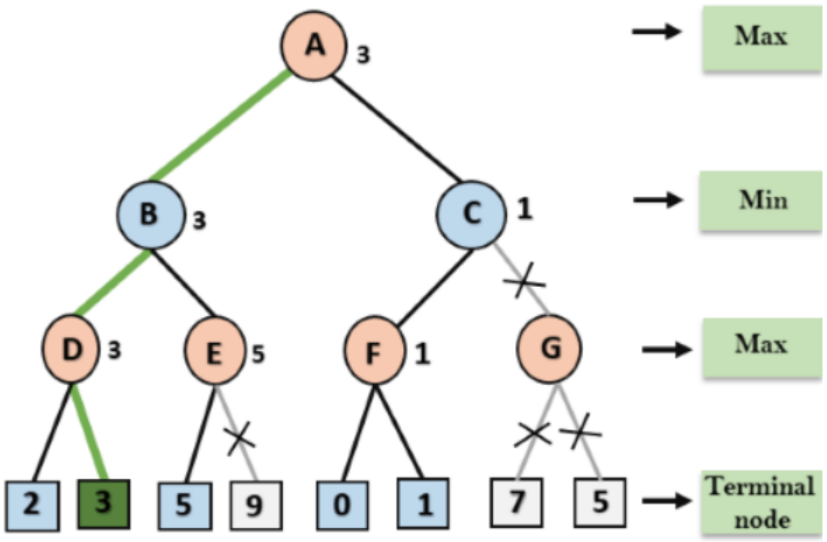


Figure 2.3: Alpha-Beta Pruning Search Algorithm[2]

Unlike Tic-Tac-Toe, where it is sufficient to generate a complete game tree once at the beginning of the game, Abalone, due to its complexity (an average game consists of approximately 80 turns, with around 60 possible moves per turn), can only generate game trees of depth around 5 at each turn. In the beginning, we examined both sequential and parallel implementations of the minimax algorithm. Although based on our parallel strategy it balances the workload across threads, at specific depths, it will traverse all possibilities, introducing a significant amount of additional evaluations. This incurs a considerable time overhead, particularly for a large search tree.

In addition to the evaluation value for each node, the alpha-beta algorithm introduces alpha and beta factors for each node. The alpha factor represents the highest-value choice we have found so far at any point along the path of the Max node. On the other hand, the beta factor represents the lowest-value choice we have found so far at any point along the path of the Min node. As shown in Fig 2.3, which represents a search tree of depth 3 (although the number of branches at each level is much smaller than the actual search tree in the Abalone game), similar to the minimax algorithm, the node values are propagated from leaf nodes to the root node. However, in the alpha-beta algorithm, these values are also used to update the alpha-beta window of each node. This allows us to determine certain branches that are definitely not worth exploring further, based on being worse or better, respectively, for Max and Min nodes.

In terms of the accuracy of the search results, there is no difference between the alpha-beta

[2]https://www.javatpoint.com/ai-alpha-beta-pruning

algorithm and the minimax algorithm because the pruned branches are certain to not be chosen. Furthermore, if we disregard computation time and consider only the perspective of winning, the alpha-beta strategy is already nearly optimal and guarantees at least not losing. In other words, for such a strategy, the only way to increase the winning rate in a timed competition is to complete the deepest possible search in the shortest possible time.

2.2.2 Monte-Carlo Search

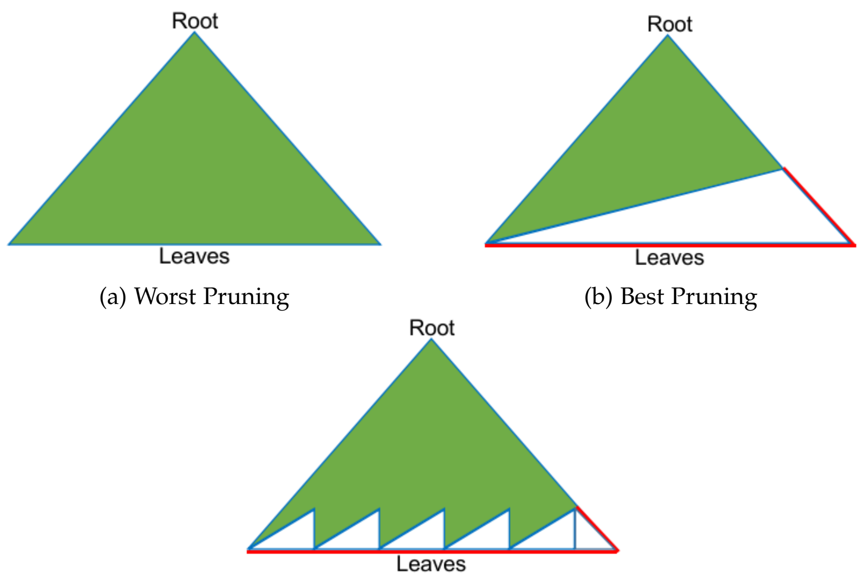
We also explore one of the most famous search algorithms, the Monte-Carlo search algorithm. Unlike the exhaustive approaches of minimax and alpha-beta algorithms, the Monte-Carlo search relies on probability and statistics to determine each move in Abalone. It simulates a certain number of games starting from the current board situation, evaluates each possible move based on the simulation outcomes, and ultimately selects the move with the highest winning probability.

However, for our application, it is highly unlikely for the Monte-Carlo search to outperform the alpha-beta search. We consider several reasons for this. One potential explanation is the randomness inherent in the Monte-Carlo method. While randomness can explore a broad range of possibilities, it lacks the cut-off strategy used in the alpha-beta search, making it less efficient. Additionally, the alpha-beta algorithm can utilize heuristics to make smart decisions about which branches of the search tree to explore first, thereby reducing the search time. In contrast, the Monte-Carlo search may have to simulate many unnecessary moves due to the lack of heuristic guidance. Lastly, while the alpha-beta algorithm can provide an exact minimax decision for a perfectly modeled game, the Monte-Carlo search gives an approximate solution, which can be less accurate, especially in complex games with deep search trees.

2.3 List of Optimizations Investigated

2.3.1 Move Ordering

For the alpha-beta pruning search, the order in which potential moves are generated (or, in other words, the order in which the depth-first search explores potential moves) is crucial. The performance of the alpha-beta algorithm is optimal when the tree is minimized, or when the total number of visited nodes is minimized. As shown in Fig 2.4a, assuming the depth-first search starts from the leftmost branch of the tree, in the worst-case scenario, the best moves are positioned at the end of the search, i.e., the rightmost branches of the tree, the alpha-beta search will not prune any of the tree's leaf nodes. This leads to the alpha-beta search tree having the same structure as the minimax search tree. In terms of performance, it may even be slower than the minimax algorithm due to the additional computations introduced by the alpha and beta factors. In the ideal scenario, however, good moves are concentrated on the left side of the tree, resulting in more and more leaf nodes being pruned as the search progresses as shown in Fig 2.4b. This demonstrates the performance advantage of the



(a) Worst Pruning (b) Best Pruning

(c) Our Pruning

Figure 2.4: Pruning of Different Move Orderings

alpha-beta pruning algorithm, without sacrificing any accuracy in the search, compared to the minimax algorithm.

According to the heuristics of the Abalone game, the ranking of different move types from best to worst can be as follows:

1. 3 stones, pushing out;
2. 2 stones, pushing out;
3. 3 stones, pushing;
4. 2 stones, pushing;
5. 3 stones, inline;
6. 2 stones, inline;
7. 3 stones, board side;
8. 2 stones, board side;
9. 1 stone.

The underlying idea is that in the Abalone game, moving more stones at once implies making better use of each turn's opportunity. Simultaneously moving multiple stones increases the potential for stronger attacks or opportunities to push opponent stones out of the board, thus maximizing the scoring potential. In addition, inline moves are more advantageous for scoring through offensive strategies, while side moves are mostly used for defensive purposes or as an escape mechanism.

In the original program, the logic for generating possible moves is as follows: first, scan each of our own stones; for each stone, scan its six directions; for each direction, add the possible moves to the list in ascending order of the number of stones to be moved. This is almost opposite to the aforementioned guideline based on experience, which leads to fewer pruning opportunities and results in a tree structure close to Fig 2.4a, leading to poorer performance. If we were to arrange the generated possible moves globally according to the previous heuristic, it would require a complete rewrite of the generation logic and could be difficult to ensure compatibility with other parts of the program. Therefore, a compromise approach is taken. Instead of changing the original logic, in each turn, a depth-first search of possible moves is performed starting from the end of the generated list. This approach results in a zigzag pruning structure, as shown in Fig 2.4c (the number of zigzags in the figure may not necessarily match the actual scenario). This balances the improvement in performance with the effort required for code modifications.

2.3.2 Adaptive Depth

While it is difficult to quantify the contribution of the search depth in each turn to the overall winning rate, it is evident that different depths have significant variations in their time requirements for the search. The logic for winning in a timed Abalone competition is to either win within one minute or to use up the game time later than the opponent. Therefore, continuously adjusting the preferences for search accuracy and efficiency during the game is crucial. Although this process is largely based on experience, it can improve the overall winning rate from a strategic perspective.

We dynamically adjust the search depth for each turn based on the evaluation score and remaining time. The basic idea is to explore the search tree as deeply as possible to establish an advantage in the early stages of the game or reverse a disadvantage when the evaluation score of the current board layout is low. On the other hand, when there is little remaining time, the search depth is adjusted to be small. This is because, at this point, both players have few remaining marbles, the board layout is simple, and the game may be in a stalemate. Therefore, a low-precision search is sufficient, and frequent moves are advantageous in breaking the stalemate and increasing the likelihood of the opponent running out of time before we do. Additionally, if a deeper search is adopted when time is running out, there is a high risk of the game ending during one of the searches. In the worst case, this would almost waste the time spent on a complete search with a larger depth. In addition to the experience gained from multiple experiments, we also refer to the average search time requirements for different depths and map them proportionally to divide the game time into 60 seconds. The final depth adjustment strategy is as follows:

- if the remaining time is less than 5 seconds or 10 seconds, the search depth is adjusted to 3 or 4 respectively;
- if the best evaluation score from the previous round is less than -600, the search depth

is adjusted to 6;

- if the best evaluation score from the previous round is less than -400 and there are more than 20 seconds remaining, the search depth is also adjusted to 6;
- in all other cases, the search depth is adjusted to 5.

Iterative deepening is also investigated in the experiment. Iterative deepening means making full use of the limited time by iteratively increasing the depth of the search from 1 within the time allotted for each move until the time runs out. This allows for better utilization of move ordering advantages, but it requires the introduction of additional per-move search timers, checks for the remaining time, and determining optimal preset time limits. Additionally, with each depth increase, the search has to repeat previous searches, resulting in significant time overhead, particularly at deeper depths. Therefore, we have ultimately decided to only use the aforementioned adaptive depth strategy.

2.3.3 Board Caching

Similar to the data locality principle in computer cache, it is highly likely for the same board layout to occur repeatedly in adjacent moves, especially in the later stages of the game when both players have fewer stones and the strategies are almost identical. By leveraging this pattern, in addition to the current board layout, we also store the board layouts from the previous two turns. Before starting the search in each turn, we check if the current board layout matches the board layout two turns ago. If a match is found, we reuse the previous search results to make the move (which will obviously be consistent with the search result for the current turn), adjusting the current layout to match the layout of the previous turn. This avoids the time overhead of redundant searches.

2.3.4 Evaluation Function

For any board game, implementing an evaluation function that quantifies the advantage or disadvantage of a board layout is complex yet crucial. It reflects the subjective preferences of the developer and often depends on experience in many cases. A slight modification to the provided evaluation function can lead to significant changes in the game situation. We first enhance the evaluation weights of the three rings closer to the center of the board from (45, 35, 25) to (70, 60, 40) because of our conservative gaming preference. However, an unacceptable situation occurred. When the program is one step away from cornering the opponent at the edge of the board, thus leading to victory, our strategy refrains from making it due to the strong attraction of the board center. Instead, it opts for an inward move, giving up the advantage and consequently losing the match. We attribute this oddity to the low weight 0 assigned to the outermost ring, resulting in incorrect decision-making. To rectify this, we elevate the weight of the outermost ring from 0 to 1. In the end, the modified version of the evaluation function consistently outperforms the previous version, whether using X or O.

2.3.5 Heuristic Opening

In the experiments, we observe that deploying deeper search tactics during the opening phase notably improves the overall winning rate. However, at the game's onset, both players have a substantial number of stones, resulting in a multitude of potential moves, which invariably leads to extended search times. Our goal is, therefore, to identify optimal solutions within the shortest feasible timespan to maximize our chances of winning.

To minimize the time consumed by the first few searches, we propose to use a fixed starting strategy. At the beginning of a game, the conflict between our stones and the opponent's is virtually negligible, implying that some movements can be strategically determined based on prior experience regardless of how the opponent moves, much like chess. For example, our test has revealed that employing a search depth of six during the game's commencement takes approximately three seconds per move. By applying pre-determined, optimized opening sequences for the initial two moves, we can secure satisfactory outcomes while saving six seconds (roughly one-tenth of the total time). This time saved can often be reallocated to deeper searches during later game stages, thereby enhancing our win probability.

Besides deploying pre-determined, experience-based opening sequences, we've also contemplated an interesting opening strategy. It involves foregoing any searches at the game's onset (especially when playing second) and directly mimicking the opponent's moves in a symmetrical direction. The advantage of this tactic lies in achieving almost identical efficacy to the opponent's moves while significantly reducing search time.

2.4 Final Setup

The final setup utilizes OpenMP parallelization at the root node level of the search tree and employs a sequential alpha-beta algorithm on each thread. Additionally, all optimizations from the previous section are utilized. The relevant parameter settings have also been described in detail.

In addition to that, we have also investigated compiler-related choices. Because the platform for compiling and running the application is processors based on Intel architecture, using the Intel compiler can take advantage of specific hardware optimizations, thereby improving performance. Additionally, the Intel compiler has stronger support and optimization for parallel computing. Therefore, we ultimately choose to use the Intel compiler instead of the more general GCC compiler (although in our experiments, there is almost no performance difference between the two).

We have naturally selected the highest optimization level, `-O3`, and also investigated the `-ipo` and `-xHost` compilation options. `-ipo` is a technique for global optimization during the compilation process that can optimize multiple functions together to improve overall code performance and efficiency. However, it does not significantly improve the performance or win rate for our application. Instead, it often unpredictably alters the default game strategy. Additionally, because our application may not significantly benefit from vectorization, the `-xHost` optimization targeting the host processor may have little to no effect. In the end, we only utilize the `-O3` optimization and `qopenmp` for enabling OpenMP.

2.5 Results

2.5.1 Pruning Effects

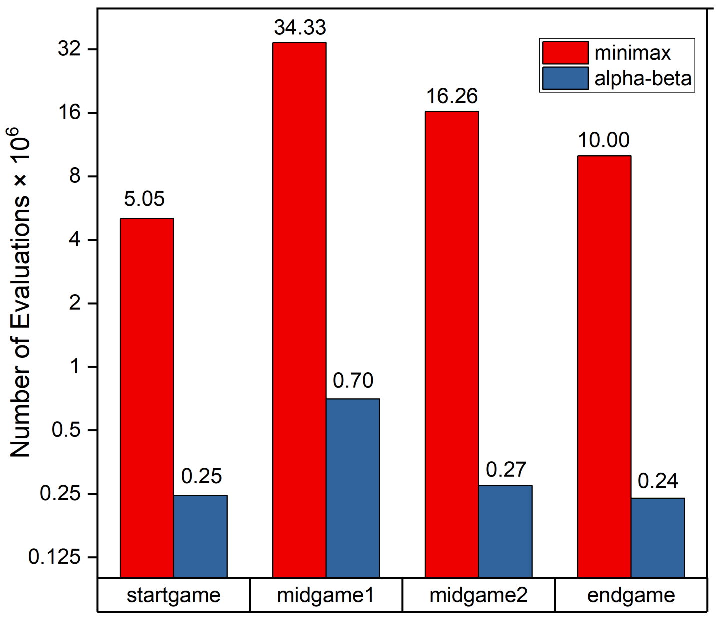


Figure 2.5: Pruning Effects of Alpha-Beta (Depth 4, One Move)

The original implementation of the minimax algorithm, developed in the first Abalone assignment, is used as a baseline for comparison with the final alpha-beta implementation. The final application utilizes adaptive depth, where a depth of 5 is chosen for the majority of the game. On the other hand, the minimax application utilizes a customizable fixed depth. However, in the later stages of the game, when the minimax application reaches a depth of 5, it requires a significant amount of time for searching, which is not feasible for measurements. Therefore, a compromise is made by selecting a depth of 4 (for measurement purposes, the final application is adjusted to use a fixed depth of 4). Additionally, only one move is performed at different positions.

As shown in Fig 2.5, the number of evaluations is indicated on top of each bar. It is evident that the pruning effect of the alpha-beta algorithm is very significant (a logarithmic scale is used on the y-axis, and the actual effect is even more pronounced), with an average pruning rate of approximately 97%. For depth 5, we have only tested the number of evaluations in the starting position of the game, and it drops dramatically from 283,320,928 to 1,605,372, resulting in a reduction of approximately 99.5%. We define the reduction rate of the number of evaluations, which is equivalent to the number of leaf nodes, as the pruning rate. The reduction rate of the total number of visited nodes is also a good indicator, but the former is sufficient to demonstrate the effectiveness of the alpha-beta algorithm.

2.5.2 Scaling Runs

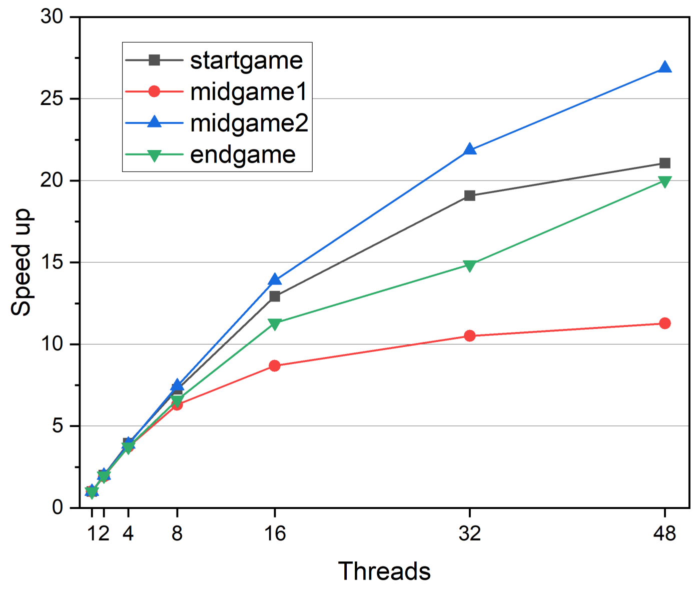


Figure 2.6: Abalone Scaling Runs (Adaptive Depth, One Move)

As shown in Fig 2.6, scaling runs are performed on one node for the final Abalone application with adaptive depth and only one move at different positions. The calculation of speedup is based on the number of evaluations per second, with the case of using a single OpenMP thread being treated as the baseline.

For different positions, the speedup steadily increases as the number of threads used increases. Especially, when using 48 threads at the position of midgame2, the speedup achieves approximately 27×. When a higher number of threads are used, there is a significant difference in speedup among different positions. For example, at the position of midgame1, the speedup only reaches about 12× in the end. This could be due to the varying degree of workload imbalance among different threads during the parallel searches at different positions, and the increasing number of threads used amplifies this difference. On the other hand, when a lower number of threads are used, there is less variation in speedup among different positions. This could be due to multiple dynamic scheduling with a chunk size of 1, which minimizes workload imbalance on each thread as much as possible. Furthermore, even when fully using one node, the speedup curves have not saturated yet. This opens up the possibility of utilizing MPI to accelerate Abalone on distributed memory systems. Nevertheless, the larger variations in speedup among different positions due to the increased number of processes and the more complex implementation based on MPI need to be considered.