CS3211 Project 2

Yasen Petrov A0153226M

April 2017

Introduction

In this document I provide a description of my approach to the problem, the techniques I used to improve my agent and a discussion of the efficiency of these techniques. The focus of the discussion will be on parallelisation - how I have solved problems that arise when parallelising the algorithm and the effects parallelisation had on performance.

Before starting parallelising, I created a serial agent - there were two good reasons for this - firstly, I did not have to debug parallel code while smoothing out the kinks in my initial implementation and secondly, the sequential code serves as benchmark to measure the efficiency of the parallel implementation. Sections 1 and 2 describe the most important components of the system - how we search the game tree and how we evaluate boards. I have kept the discussion short as the focus of the report is parallelisation. Section 3 contains a description of how I went about parallelising my OthelloX agent and a discussion of how effective it was. Section 4 analyses the performance of the algorithm across different board sizes, different number of processes and implementation-specific parameters. Section 5 discusses ways to improve the performance of my agent. This can easily be the longest section in the report but I have tried to keep it short in order to focus on describing what I *have* done.

1 Search strategies

As OthelloX is a two-player zero-sum deterministic game in a fully observable environment, me can deterministically generate the next nodes in the game tree an apply the Minimax algorithm. We build the game tree recursively by generating all the possible moves for the current player given the state of the board and then applying those moves to generate the possible next states.

Minimax

The fact that OthelloX is a zero-sum game means that both players(if they are rational) will try to maximize their score and in so doing minimize the

opponent's. Therefor, to find the optimal move in a game between two rational players(call them MIN and MAX), we can alternate between choosing the move that will bring the maximum score for MAX (MAX nodes) and the one that will bring the minimum score for MAX(MIN nodes). Plain Minimax search does a DFS exploration of the game tree, alternating between MIN and MAX node at each ply.

Alpha-Beta pruning

The number of nodes Minimax has to evaluate grows exponentially with the depth of the search(the exponent being the average branch factor, which, for regular Othello is around 7) which means that plain Minimax is not terribly efficient in exploring the game tree. Alpha-Beta pruning is a very efficient technique for reducing the search space without sacrificing the correctness of the algorithm. The effects of pruning can be seen on Fig/1, which compares the performance of the sequential algorithm with and without pruning.

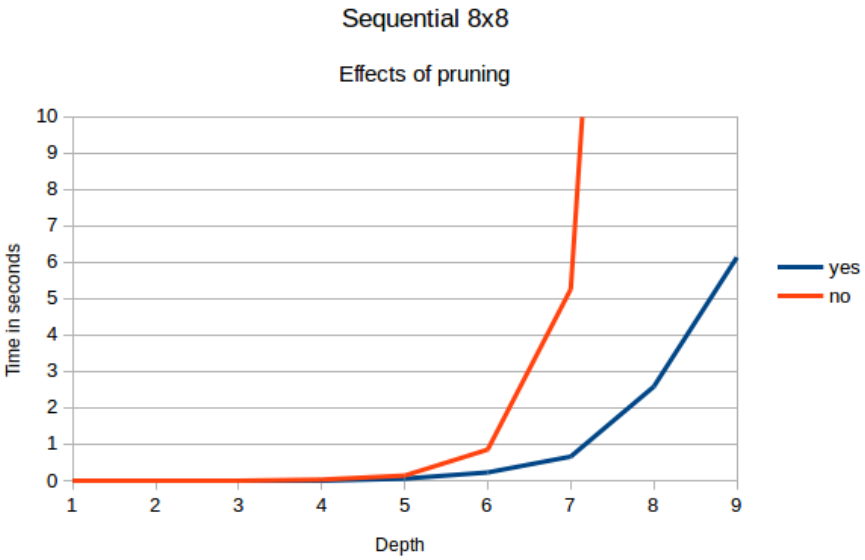

Figure 1: Pruning effects

Consider a situation where we are at a MIN node and assume we know that our parent MAX node has got a score of 10 from evaluating another child. Now assume the evaluation a move at the current(MIN) node returns a value $v \leq 10$. This means that the current node will choose a move with a score at most 10. Its parent(MAX) will choose a move with a score at least 10 since it has already gotten a value of 10 from another branch. Therefore, evaluating the rest of the

possible moves at the current node is pointless, so we safely "prune" them and return v. This means that the best value found by a MAX node so far puts a lower bound on the values returned by its next MIN children. We call this value α. Its analogue for MIN nodes we call β. The process of choosing not to evaluate certain nodes based on this criterion is called Alpha-Beta pruning and is very efficient - note that when exploring the game tree of a game with an average branching factor of b up to a depth of d, pruning a single node at depth k means we have to evaluate, on average, b^{d-k} less nodes.

Negamax

We note that minimising the score for the other player is the same as maximising the negated score for them. Negamax search is nothing but a simplification of the implementation Minimax search algorithm that makes use of that observation. Instead of alternating between choosing the next move with the maximum and minimum score, Negamax chooses the maximum out of the negated scores for the next nodes.

Move ordering

When using Alpha-Beta pruning, evaluating the best child for the current node first can lead to a drastic improvement in performance since many more nodes will be pruned. There is, however a trade-off between computation cost and the quality of the prediction when choosing the next best move. A computationally cheap technique is to use a simple heuristic. For instance, since corner and edge squares are valuable in Othello(more in the discussion of stability in Section 2), if the opponent plays a piece to a square adjacent to an edge or a corner square, see if we can capture that edge or corner square. Another way of ordering moves is preforming shallow searches. This is the technique I chose and has a couple of drawbacks - it was too expensive even at a depth of 2 with my more complicated evaluation function(More in section 2) so its effect was detrimental to the performance when using that function. The other problem is that it is often the case that shallow searches do not provide a very good approximation of how good the move is several moves ahead. I have had minor performance improvements when using move ordering with my static evaluation function.

2 Board evaluation

When we are at a terminal state, we can easily determine how "good" a board is - its score is the difference between MAX and MIN pieces. However, since we cannot explore the whole game tree in the early and middle stages of the game, we need to be able to assign a score to a board in a non-terminal state when we reach a cut-off depth. There is also a trade-off when writing an evaluation function - easy, cheap solutions do not capture some of the details that entail

how good a board is and encapsulating more complex ideas in your evaluation function is expensive. I have implemented two evaluation function - one with static weights that I also have a parallel implementation of(More in Section 3) and a more complicated one that I only have a sequential implementation of.

Board representation

In the final implementation my board is represented as a one-dimensional vector of signed characters. I had initially implemented it as a nested vector, which meant I had to pass the data row by row. Passing the whole board in one message improved performance by about 10%. This representation is still far from optimal - more discussion in Section 5

Complex evaluation

The more complicated of the two evaluation function makes use of several known heuristics for playing Othello[5], computes values in the range $[-100; 100]$ for each and produces a weighted sum of those as the utility for the board. If the board is in a final state, the evaluation function only calculates the difference of the number of pieces and increases its absolute value by a 100 to mark it as more reliable than an evaluation of a board in an intermediate state. The heuristics are described below.

Parity The parity score takes in account the number of pieces each player has on the board. This by itself is a rather naive heuristic, especially early in the game.

Mobility A popular heuristic for playing the game is to maximise the number of moves you have and minimise the opponents. My function only accounts for the number of moves players have available in the current state. A more advanced version would estimate the mobility in future states.

Stability Stability is probably the most reliable of the three heuristics. Pieces are called stable if they cannot be flipped anymore. The first stable piece will occur in a corner and the player who captured that corner can build stable pieces along the edges. Pieces not on any of the edges are stable if there are only stable pieces in the space between them and a corner. The evaluation function counts stable pieces for both players by starting in each of the four corners.

This evaluation function is computationally heavy and difficult to parallelise compared to the one described in the next subsection because the calculation of the mobility and Stability scores requires knowledge of the rest of the board.

Static evaluation

A much simpler approach to evaluation is to assign fixed weights to each square on the board(Corners are very desirable, squares next to them are bad if we don't have the corner, edges are good and inner-squares are OK) and compute the element-wise product of the current state(MAX pieces are 1, MIN pieces are -1) of the board and the weight matrix. This requires less computations, it is easy to parallelise and leads to a big performance boost.

3 Parallelisation

The two major components of the agent are the search and evaluation algorithms. I have made parallel versions of both the search algorithm and the static evaluation function. I have not explored in depth the idea of running parallel search with parallel evaluation.

Parallel search

The search algorithm was what I was the more challenging to parallelise. The basic idea is to generate some of the nodes of the search tree, let different processes evaluate subtrees starting at different nodes and then aggregate the results they got. I have identified two major issues to be tackled. I have a basic implementation which deals with the problem of load balancing across processes. The other issue is the propagation of $\alpha - \beta$ values - assigning lower-level nodes to different processes will inevitably reduce the efficiency of Alpha-Beta pruning since when distributing sibling nodes among processes, we would not know if some of them have to be pruned until we get a result back from at least one of them. Moreover, the $\alpha - \beta$ window will have to have been update by the precious "uncle" node, which in turn has to know the cousin nodes' scores. I have not addressed this issue in my implementation.

Load balancing

An obvious way to assign nodes to processes is to generate one node for each. The problem with this approach has to do with the fact that the number of boards that the process has to evaluate grows exponentially with the distance of the node from a leaf node(or a node at the cut-off depth). Obviously, if we have N processes, there is no guarantee that we will have a ply in the game tree with exactly N nodes. This means that the height of the subtrees assigned to different processes will often differ by at least 1, which would mean some nodes will have to evaluate 7 times more nodes(on average). Moreover, processes that are assigned more promising nodes are more likely to prune a larger part of their subtree. All of this means that this method of assigning work to processes leads to poor load balancing.

Model How I have tackled this problem is by employing the Worker pool model. A pool of game-tree nodes with a size several times bigger than the size of the process pool(I call the ratio of work pool size to number of processes the *load factor*) is generated. Processes get a node from the pool, explore it, return the result and get a new node. This means that when a process gets a subtree that is much cheaper to evaluate, it gets a new subtree instead of having to be idle until the most heavily loaded process finishes its search. With a big enough load factor, this leads to the load being spread evenly across processes.

Trade-offs However, when the load factor is very big, even though we will get a good spread of the load across processes, the nodes we send to the processes will be at a bigger depth, which will reduce the effect of Alpha-beta pruning. That would also mean putting a heavier load on communication. The smaller effect of pruning can be seen in Fig. 3 - the number of nodes evaluated on the same problem size grows as the number of processes and the pool size grows - as expected. The improvement in load balancing that comes with increasing the load factor is visualized in Fig.2

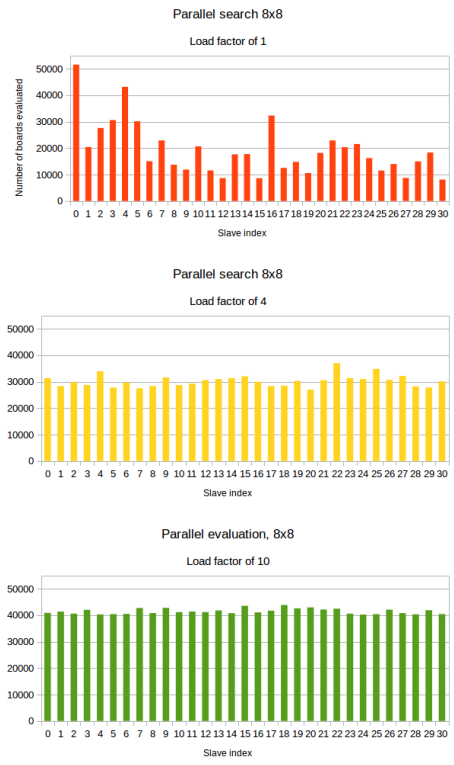

Figure 2: The effect of pool size on load balancing

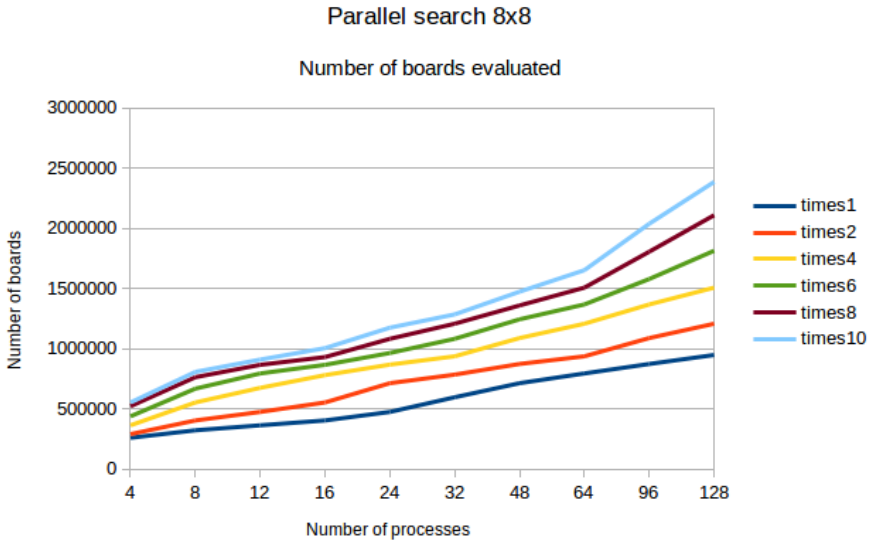

Figure 3: Boards evaluated by parallel search

Implementation One of the process is designated as a master process and is responsible for filling the work pool and distributing the work amongst the other processes, which I call slave processes.

Master When filling the work pool, the master stores all of the generated nodes, starting from the root in an array. Each node generated stores the best score so far, its type(MIN or MAX), its depth and the index of its parent node in the array. This information is then used to propagate the scores returned from the slaves back to the root. The process of generating nodes involves the aforementioned node array and a queue and goes as follows: we store the root in the node array and enqueue its index. Then we repeatedly dequeue a n index, generate the nodes for that index, store them in the array and enqueue their indices until we have enough nodes in the queue to fill the pool. If we cannot generate more nodes, we stop. The master then proceeds to send a job to each slave. A job consists of a node's board, type(MIN or MAX) and its index in the array, which is used as a unique identifier for this job. In the next stage, the master waits for a results from slaves. Along with the score for that subtree, the result contains the job id. The master uses that to store the score in the node array. Then it dequeues an index, creates a new job and sends it to the slave it just got results back from.

Slave The slave process is quite simple - each slave runs an infinite loop, first waiting for a flag from the master, which indicates whether a next job is

to be sent. If the slave is told no jobs will be sent, it terminates. Otherwise, it receives a job and runs the $negaMax$ algorithm with the node received as root. Then it sends the score to the master process and loops back to the waiting for the flag. When the master has received results from all slaves, it sends a flag, indicating no more work is to be done to all slaves.

Corectness To assert the correctness of my implementation, I modelled the process in CSP, using PAT. The *.csp* file is included in my submission.

Communication/computation Graph charts graphs charts

Parallel evaluation

In the more complex evaluation function described above, when evaluating mobility and stability, the evaluating process needs knowledge about other parts of the board, which means an distributed algorithm for performing that evaluation would be more complicated and would have a heavily load on communication, which, for the small board sizes we are dealing with, will probably render the algorithm useless. Therefore, I restricted the parallelisation of evaluation to the much simpler static evaluation function.

Model The model, apart from being embarrassingly parallel, is embarrassingly simple. One process, which I call *master*(maybe in dissonance with the correct terminology in this instance), which the search algorithm runs in, distributes the work evenly amongst all of the processes(including itself). Every process performs the same computation on the sub-problem - it multiplies the elements it received by the corresponding weights and produces a sub-score. The sub-scores are then gathered and summed by the master process.

Implementation, load balancing Every process has to be sent a part of the board and we want the distribution to be as even as possible. To do this, we compute offsets from the beginning of the board(its flattened representation, rather) to ensure that every process gets $\lfloor \frac{NM}{P} \rfloor$ elements and the first $N \, mod \, P$ processes get an extra element, where N and M are the board dimensions and P is the number of processes. Theses offsets are computed by every process before the search starts. The offsets are then used by the master and slave processes in an MPI_Scatterv() call to distribute the data. The slave then uses them to get the right elements from the weights matrix, which is also initialized for every process before the search starts. As in the parallel search algorithm, each slave runs an infinite loop, waiting for a message telling it whether it will get more work. A flag indicating no more work is to be done is broadcast to all slaves when the search terminates. When the master is done with its own computation, it gathers the sub-scores from the rest of the processes, and returns the sum of all sub-scores.

Performance Unfortunately, at this stage, for reasonable board sizes, this approach does not lead to an increase in performance, as demonstrated in Fig4, comparing the boards evaluated per second over the number of processes.

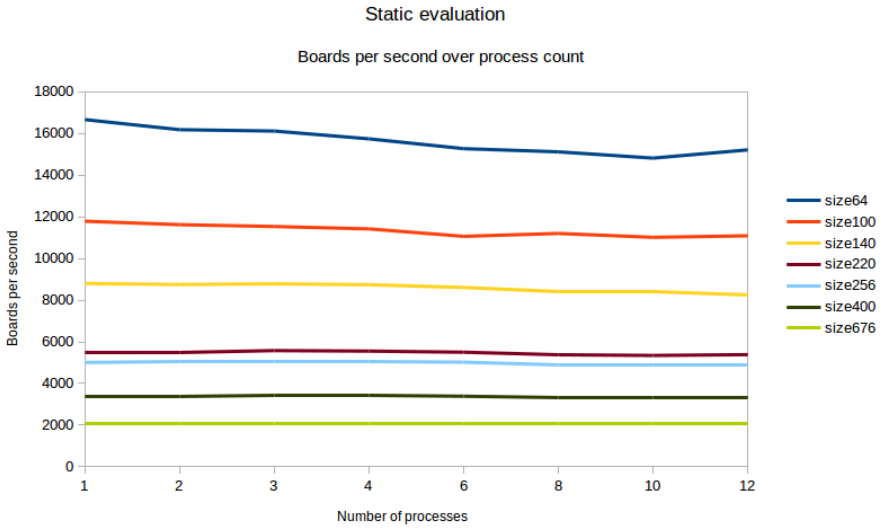

Figure 4: Parallel evaluation

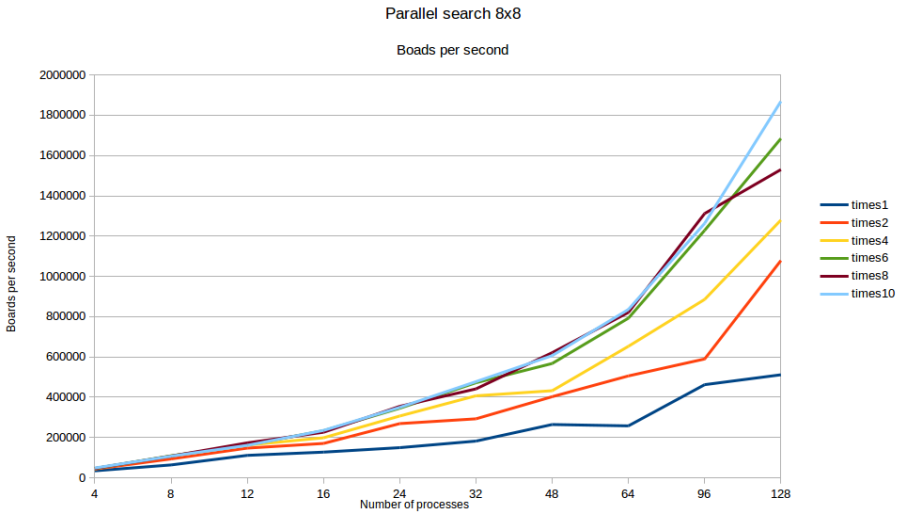

Figure 5: Parallel search BPS

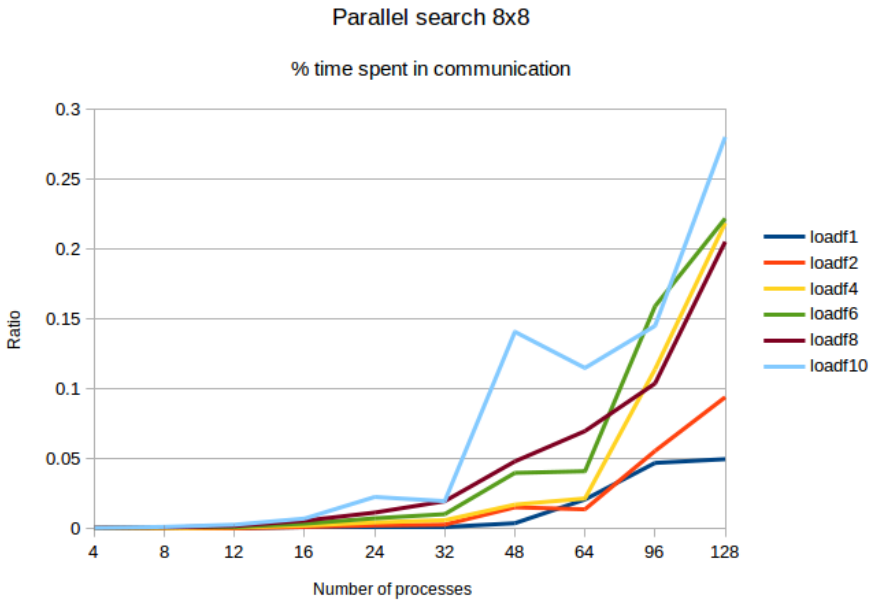

Figure 7: Parallel search communication/computation

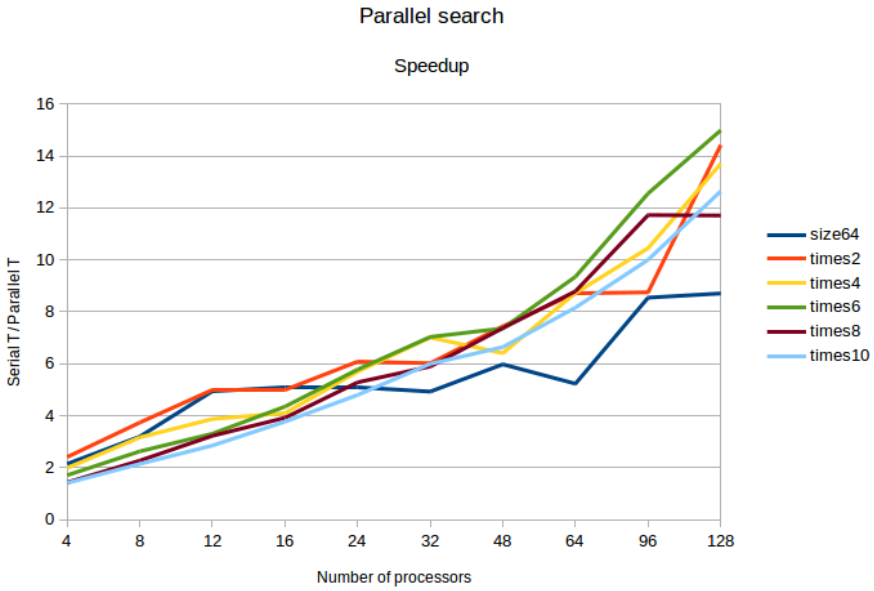

Figure 6: Parallel search speedup

4 Comparing performance

Figure 6 shows the speedup evaluating an 8x8 board from an initial position over different numbers of processors and load factors. We note that we achieve sub-linear speedup. We can compare it to Fig. 5 which shows the increase in boards evaluated per second. The latter increases more steadily because of the aforementioned reduced efficiency of pruning means that many of these evaluated boards could have safely not been evaluated. Looking at Fig.5 and comparing the BPS for different pool sizes, we can analyze the effects of job pooling since we have abstracted away the effects of pruning by measuring boards per second. We notice that for the same number of processors, bigger pools generally lead to better performance, meaning job pooling works well to distribute the load between processes. However, we notice that when the pool size gets big enough, that effect seems to vanish. We can attribute this to the increase in waiting time for processes. This problem would be solved by sending several jobs at a time and queueing them locally for each slave, as suggested in lectures. This can be confirmed by looking at Fig7 which shows the portion of the time slaves spend on communication over their lifespan.

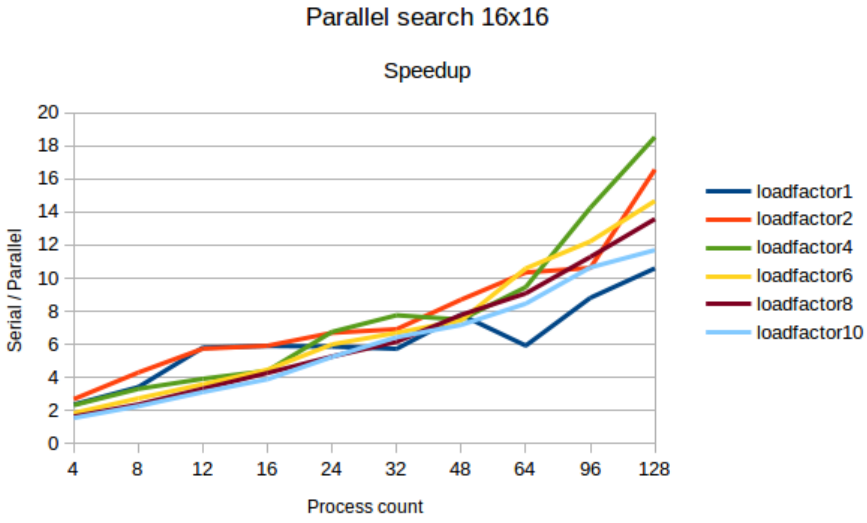

Figure 8: Parallel search speedup

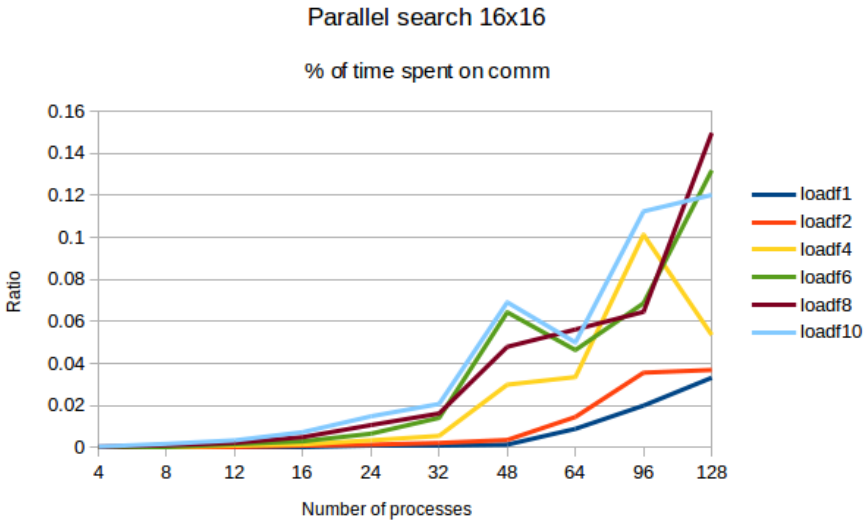

Figure 9: Parallel search communication/computation

Efficiency Fig 10 shows the efficiency of our algorithm measured in increase in boards evaluated per second over the number of processes and Fig.11 shows it measured in speedup over the number of processes.

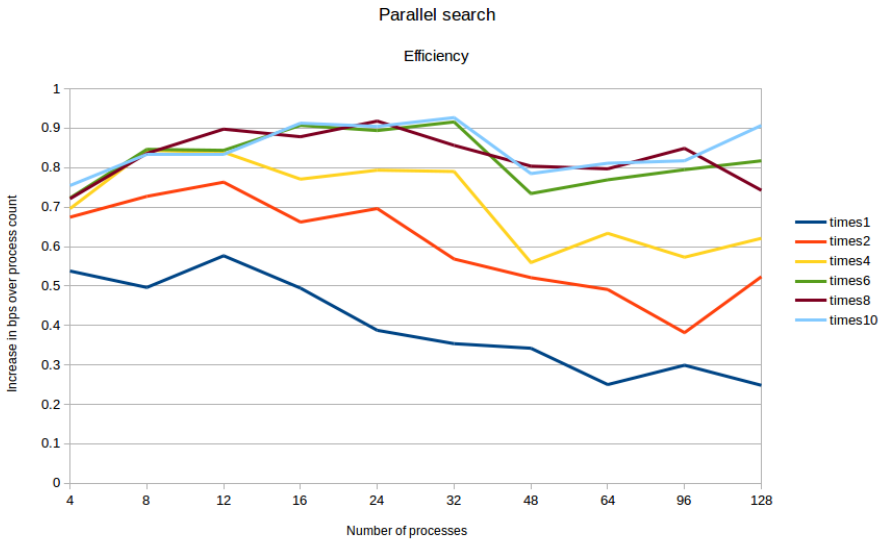

Figure 10: Efficiency of parallel search w.r.t. boards per second

5 Potential improvements

Search

Efficient move ordering is the next improvement to be made on this agent. Some of the techniques in this literature[3] such as Rank-Cut, Probe-Cut and IDS can also be explored. Probabilistic methods for pruning are also an option.

Parallelisation

For future work on this agent, I intend to explore and article[1] on parallelising game tree search that I found only too late and attempt to implement some of the techniques mentioned there as well as in the lecture on tree search, like sending several jobs to a slave at once and queueing them up. A place to start would be communicating the Alpha-Beta window between processes. Improving

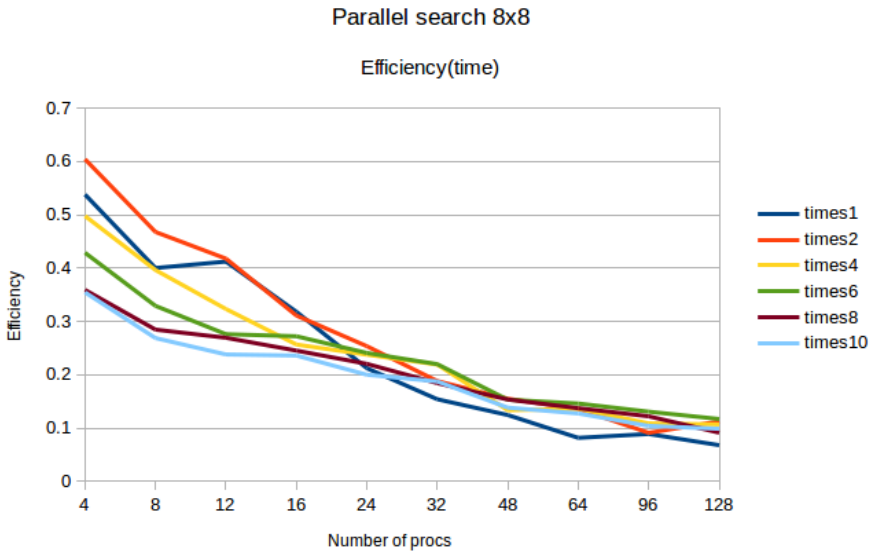

Figure 11: Efficiency of parallel search

the board representation should also cut down on communication times, leading to better performance of the distributed search. Another interesting possibility is to explore how to best make use of the topology of the network.

Board evaluation

Much more can be done both for the computational efficiency and the accuracy of my evaluation functions. The most important thing to be done is getting "good" weights for the functions. The most obvious way to do that is machine learning[4]. Adaptive weights for different stages of the game are also a must in Othello.

Board representation Currently, our board takes up $M * N$ bytes, M and N being the dimensions of the board. In functions where it's passed by value, like applying a move, for example, and when it's sent to other processors, a more efficient representation can lead to big performance gains. Pretty much all of the successful board game agents use bitboards for representation[2] and this would be an obvious next step to take in developing this agent.

Memory and learning

Transposition tables When exploring the game tree, especially in the endgame, we re-evaluate boards we have already seen. Transposition tables store

the boards found so far along with the best moves for them and the search depth the results came from(often in a hash table). We can then query the table for our current state. This can also be used for efficient move ordering. I have implemented Zorbist hashing for our board but have not implemented a full transposition table. This is a very important next step in developing the agent and will also pose interesting challenges in implementing the distributed agent[1].

Machine learning Learning the best weights for the evaluation functions and how good certain heuristics are should result in a dramatic improvement of the agent's performance in a tournament. It should also lead to more efficient pruning, increasing performance[4][2].

6 Conclusion and future work

One out of my two parallel implementation has led to a significant speedup, which has been tabulated and analyzed. In hindsight, I should have spent less time on the sequential implementation and trying to improve that. However, I think the result of this project can serve as a basis for future development of a high-performance parallel Othello agent. My future plans are to use AWS instances to run my algorithm on a distributed system and implement and asses the potential improvements listed in the previous section.

References

[1] R.Feldmann P.Mysliwietz B.Monien. *Game Tree Search on a Massively Parallel System.* URL: https://pdfs.semanticscholar.org/12c8/0362791b9727165cf7d6d63c19033ed7ece2.pdf.

[2] Jack Chen. *Applications of Artificial Intelligence and Machine Learning in Othello.* URL: https://www.tjhsst.edu/~rlatimer/techlab10/Per5/FourthQuarter/ChenPaperQ4-10.pdf.

[3] LIM YEW JIN. *ON FORWARD PRUNING IN GAME-TREE SEARCH.* URL: http://cs.uno.edu/people/faculty/bill/ML-Othello-Heuristics-ACM-SE-Regl-1991.pdf.

[4] Machine Learning of Othello Heuristics. *Applications of Artificial Intelligence and Machine Learning in Othello.* URL: http://cs.uno.edu/people/faculty/bill/ML-Othello-Heuristics-ACM-SE-Regl-1991.pdf.

[5] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. *An Analysis of Heuristics in Othello.* URL: https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf.