CS3211 Project 2: OthelloX

Tan Soon Jin, A0112213E
a0112213@u.nus.edu

April 16, 2017

1 Introduction

To generate the best move for a game of othello, I have implemented a **minimax with alpha-beta pruning** as the sequential game tree search algorithm. In addition, some enhancements are made such as using **transposition table** to store best moves for a particular board configuration and **killer heuristic** to consider more promising move first.

I have experimented with 2 board representations: the first being a **1D array** to store the board configuration which is more efficient than a 2D array because of data locality. Secondly, **bitboard** which is a bit array data stucture that is commonly used in board game is also implemented. Leveraging on bitwise operations which can be done in parallel, faster move generation and board evaluation can be achieved.

For the parallelization of game tree search, this paper primarily look at **Principle Variation Split**, **Young Brother Wait Concept** and **Dynamic Tree Splitting**. The paper will further elaborate on the strategies used to achieve optimal *load balancing*, *communication / computation ratio* and *aggregation of tasks*.

2 Sequential game tree search

2.1 Minimax game tree search

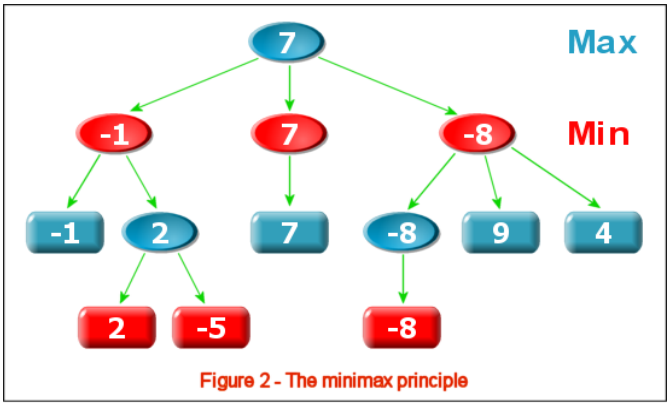

Figure 1: Minimax game tree. http://www.hamedahmadi.com/gametree/minimax6.png

Looking into various computer games like chess, checker and otheelo, minimax algorithm is the standard solver which is easy to implement and generates relatively good result. Minimax algorithm effectively is trying to minimize the possible loss for the worst case scenario with the assumption that the opponent will take the most optimal move. As show in Figure 1, it works by first generating all possible moves at an increasing depth with a depth-first search strategy.

```
01 function negamax(node, depth, color)
02     if depth = 0 or node is a terminal node
03         return color * the heuristic value of node

04     bestValue := −∞
05     foreach child of node
06         v := −negamax(child, depth − 1, −color)
07         bestValue := max( bestValue, v )
08     return bestValue
```

Figure 2: Negamax: simplified variant of minimax

For this project, negamax which takes advantage of the relationship $max(a,b) = min(-a,-b)$ which reduces to one subroutine instead of 2 subroutines for the original minimax approach. With this each evaluation is performed with respect to the current player's point of view. Provided in Figure 2 is a simple pseudocode for negamax:

2.2 Alpha-beta pruning

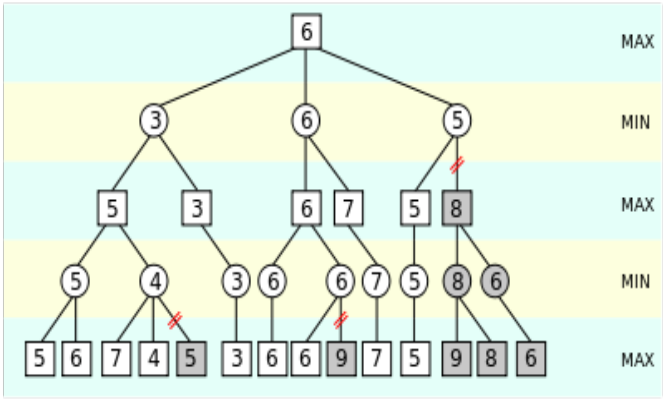

Figure 3: Alpha-beta pruning: greyed out nodes are pruned

Alpha-beta pruning is an optimization on the common minimax algorithm that prunes the search tree by removing subtrees of moves which will definitely not be searched. It works as shown in Figure 3 by maintaining an α: maximum lower bound and β: minimum upper bound. Any node in the search tree that does not fall between these bounds will be pruned, and thus reducing the search space.

2.3 Alpha-beta enhancement

2.3.1 Transposition table

Transposition table is a large hash table that stores results of previously performed searches. During a search, the program may encounter the same board configuration in different subtrees (this is known as

transposition). Inspired by dynamic programming paradigm, this approach greatly reduces search space of the program by removing redundant search.

In our hash table, the score, alpha value, beta value and depth of tree is stored. *Zobrist hashing* is used to generate the hash of the table. In a parallel environment, usage of lock is inevitable for synchronization of data. However, usage of lock is detrimental increases the *synchronization overhead* when the number of processor increases.

To solve this problem, this paper follow the approach of [3] of storing the XOR of hash key with the board state. If a corruption is detected, the value is recalculated. Another approach is to use a coarse-grain lock on the transposition table, where multiple entries of the table share a same lock to reduce the overhead of using a locking mechanism.

There are 2 advantages of using a transposition table: 1) reduce the search space of game tree and 2) move ordering based on previous searches which is essential for alpha-beta pruning.

2.3.2 Killer heuristic

Killer heuristic works by storing N moves that generated a cutoff for a particular depth. This works under the assumption that the *killer moves* may generate a cutoff in the current position.

2.4 Evaluation function

I propose a linear combination of piece differential, mobility potential and stability potential as the evaluation function for the otheelo program. Generating a sophisticated can be time consuming and requires a lot of machine learning which is out of the scope of the project. Below is brief description of the implemented evaluation function:

1. **Piece differential**
   score = number of discs - number of opponent's discs

2. **Current mobility**
   score = number of legal moves - number of opponent's legal moves

3. **Frontier mobility**
   score = number of discs with empty squares in all direction - number of opponent's discs with empty squares in all direction

4. **Stability**
   score = number of discs that cannot be flipped - number of opponent's discs that cannot be flipped

3 Board representation

3.0.1 Bitboard

Each board state is stored as bit array with one or more *uint64* because the board dimension is not fixed. Bitboard approach is both compact and allow for faster evaluation and move generation. These steps that are performed by each node could benefit massively from usage of bitwise operations that consequently reduces the search time of the program.

Using various filters or masks, generation of legal moves can be achieved easily as shown in Figure 4. Legal moves in Othello consists of moves that can capture a line of opponent's discs.

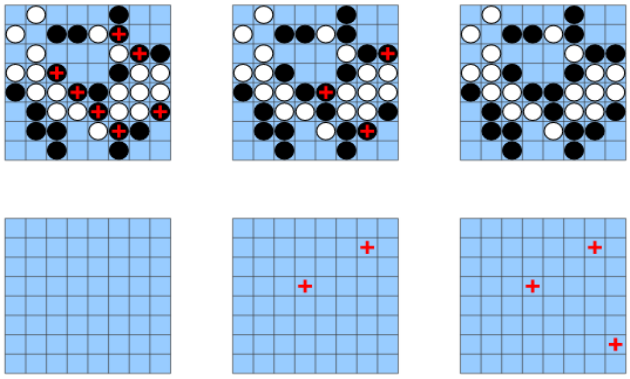

Figure 4: Top: line propagation, Bot: legal moves

3.0.2 1D array

Though bitboard is perfect for move generation and evaluation, it can be slow for query such as what type of disc (black or white) is present at a particular position. Therefore, a 1D array is used to store positions of disc for both players.

4 Parallel Game Tree Search

4.1 Naive Parallel Alpha-Beta

The first approach to parallelize alpha beta pruning involve splitting legal moves at depth = 1 to each available processors. Consequently, we try splitting the subtrees at deeper depth which causes higher workload imbalance as the lack of information from other trees disallows for pruning to be done. Tree-decomposition at width means each processor can be assigned different possible legal moves.

4.1.1 Tree-decomposition models

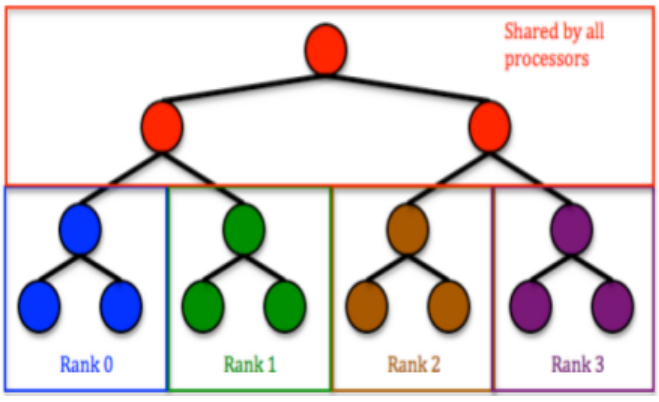

Figure 5: Even-split model for tree-decomposition

Even split model splits n children among p processors for processing independently. *MPI_GATHER* is used to aggregate best result from each processor to process 0. The process will then broadcasts the best

results to the other processes. An example of how the decomposition is performed can be seen in Figure 5

The **master-slave** model on the hand explores "manager-worker" and "asynchronous iteration" paradigms. The **manager** process is responsible for:

1. distibutes evaluation of a particular legal move to a worker process
2. gathers the best cost functions and values returned by the worker processes using *MPI_GATHER*
3. determines best evaluation function value using *MPI_REDUCE*
4. broadcasts the best move so far to worker processes using *MPI_Bcast*

The **worker** process on the hand has to:

1. Evaluates a board state assigned by manager process
2. Sends the score and best move to manager process
3. Receives designated move from manager process

With this approach we are still missing the benefit of pruning as the best alpha beta bounds are only found at deeper depths.

4.1.2 Evaluation

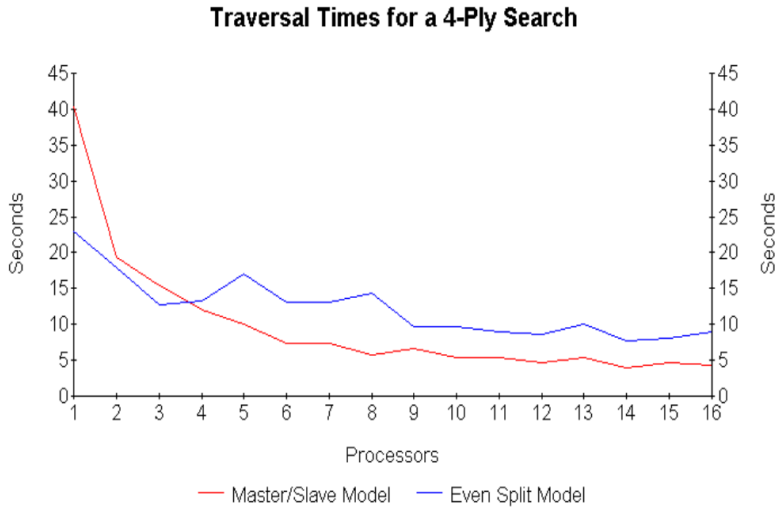

Figure 6: Comparing traversal time for master-slave model and even-split model

As we can see in Figure 6, master-slave model has a shorter traversal time compared to the even-split model for lower number of processors.

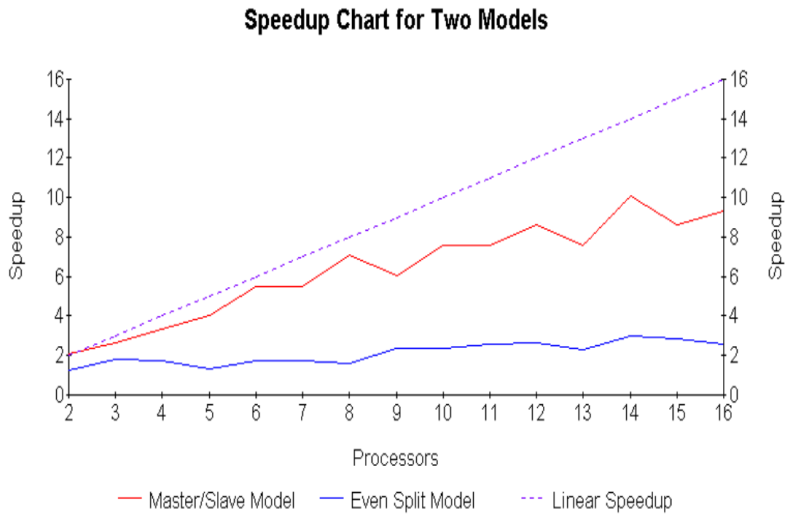

Figure 7: Comparing speedup for master-slave model and even-split model

From Figure 7 that compares the speedup of the even-split model and master-slave model, the master-slave model outperforms the even-split model in terms speedup and efficiency. It showed an almost linear speedup which can be attributed to pre-assigning processors to subtrees prevent the program from making full use of all the processors. Some processors that have completed their work have to wait for completion of other processors.

4.2 Combining OpenMP & MPI

In addition, this paper also explores a hybrid approach of OpenMP for thread-management and MPI that see a lot of application in recent years. In this section, we will look at brief overview of this approach and its benefits:

4.2.1 Overview

The hybrid approach basically launches threads which shares memory within a MPI task that are independent of each other. Performs shared memory programming inside a node and message passing between nodes. How many MPI task per node is determined through trial and error.

4.2.2 Hybrid programming styles

Fine-grained use omp parallel on the most intensive loops and easier to integrate with MPI. **Coarse-grained** use OpenMP thread to replace MPI task which enables all cores to communicate with each other. We can refer to Figure 8 for an overview of the available styles. For the hybrid approach, the *MPI_Init_thread* is place of *MPI_Init*. In addition, a thread tag is needed when using the MPI *send* and *recv*.

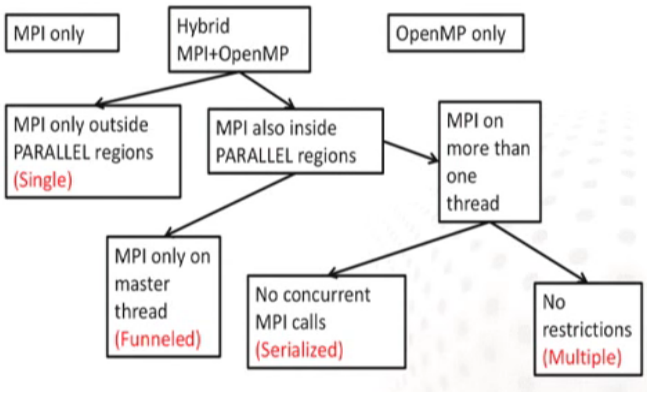

Figure 8: Hybrid programming styles

4.2.3 Benefits

Load balancing is hard to achieve as the number of processes increases. With the hybrid approach, we have less processes and can dynamically changes the number of threads per process which can improve the load balance. OpenMP also provides the capability to perform dynamic load balancing.

Reduced messages is observed as direct memory read and write are performed inside a node which reduces overhead caused by exchange of messages using the MPI API. In addition, the aggregated messages become larger which increase the throughput on inter-node communication.

Overlapping communication and computation can be achieved as master thread handles inter-communication between nodes while other threads are performing computation. With enough threads, the master thread can be used purely for communication.

Improve in memory usage is seen in domain decomposition algorithm as there are fewer data boundary points and less replicated data.In addition, cache usage is also improved as data reside in shared cache.

4.2.4 Evaluation

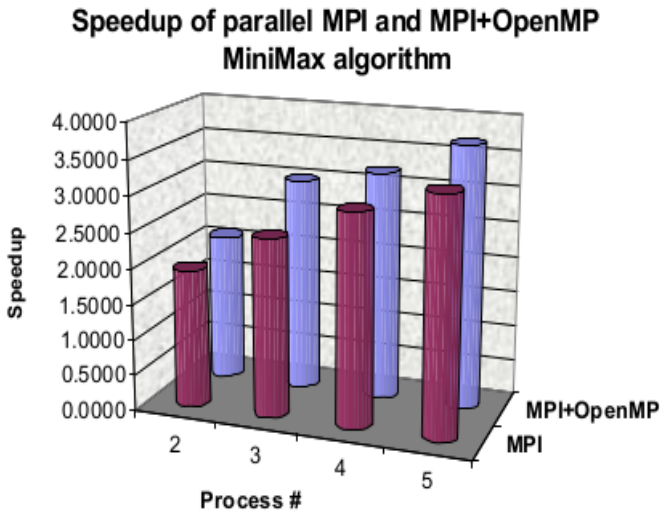

Figure 9: Comparing speedup using hybrid approach and pure MPI approach

From Figure 9, the hybrid has shown to outperform the pure approach and the increase in speedup is more significant with more processes.

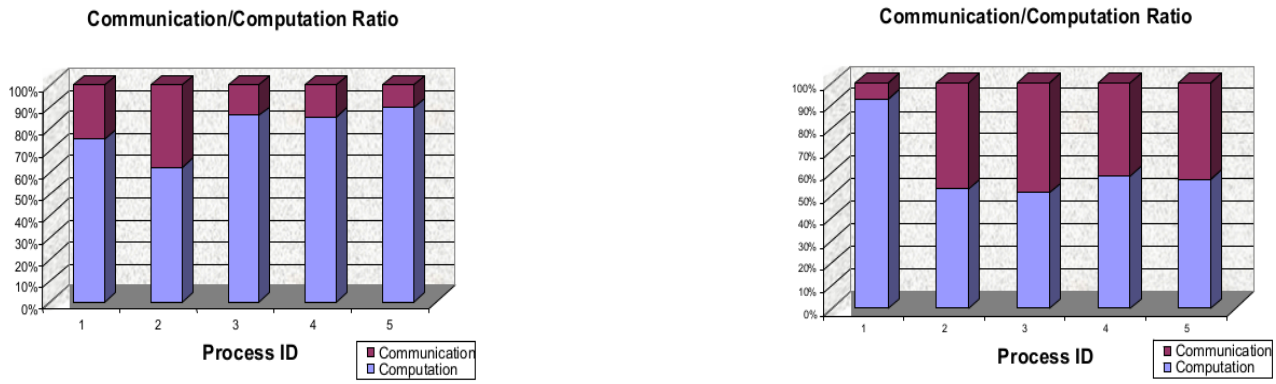

Figure 10: Communication/computation ratio: hybrid, pure MPI

In addition, the communciation overhead with reference to Figure 10 for hybrid approach is also significantly lower than the pure MPI implementation.

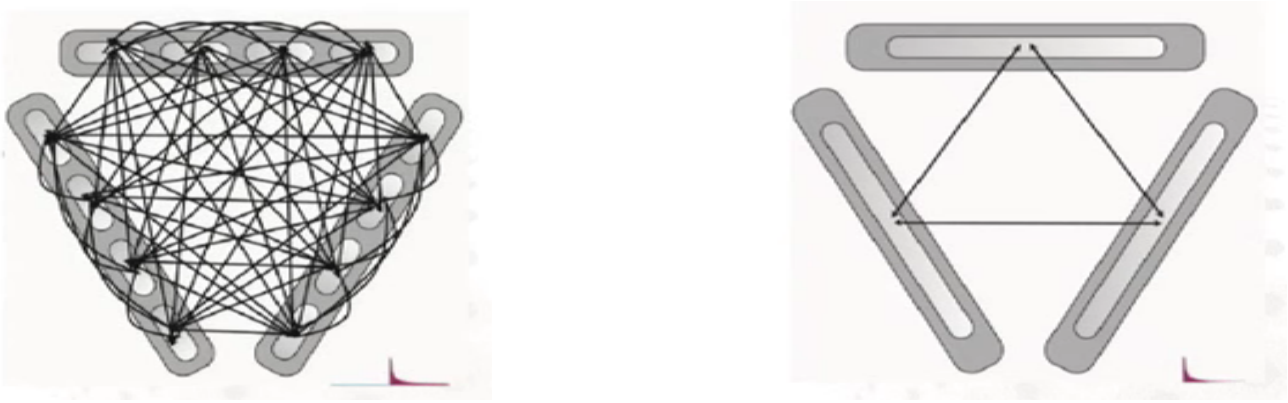

Figure 11: Benefit of hybrid programming approach

All-to-all communication is often the bottleneck for parallel algorithms as the communication overhead increases with number of processors.If we look at Figure 11, the hybrid approach reduces the communication events drastically.

4.2.5 Disadvantages

Compilicated programming is one of the downside of hybrid programmign styles. In addition, creation and destruction of threads can increase overhead.

4.3 PVS (Principle Variation Split)

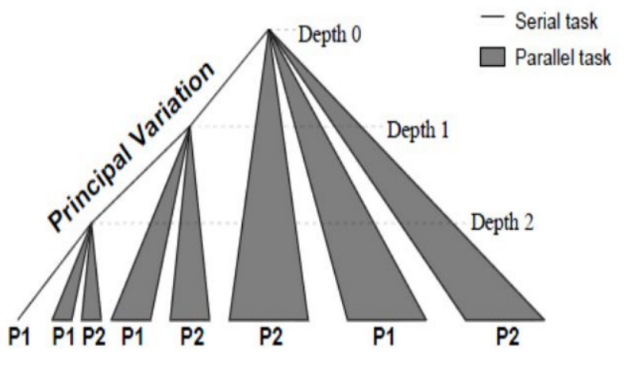

Figure 12: PV split using two processors

This approach benefits most from a strongly ordered game tree where the leftmost child of a node is the best move. Therefore, the introduction of *transposition table* and *killer heuristic* is essential. Firstly, the PV split function is called recursively on the leftmost branch of our tree. With the updated alpha beta bounds, regular alpha pruning is then ran in parallel on the other children of the current node. The tree-decomposition using PV split is shown in Figure 12

Processors	Time (ms)	Speedup
1	27	-
2	33	1.50
4	18	2.70
8	10	3.31
8	10	4.73
16	8.14	6.43
32	5.7	5.63
64	4.2	4.38
128	4.79	4.13
256	6.162	3.90

Table 1: Speedup at depth 5

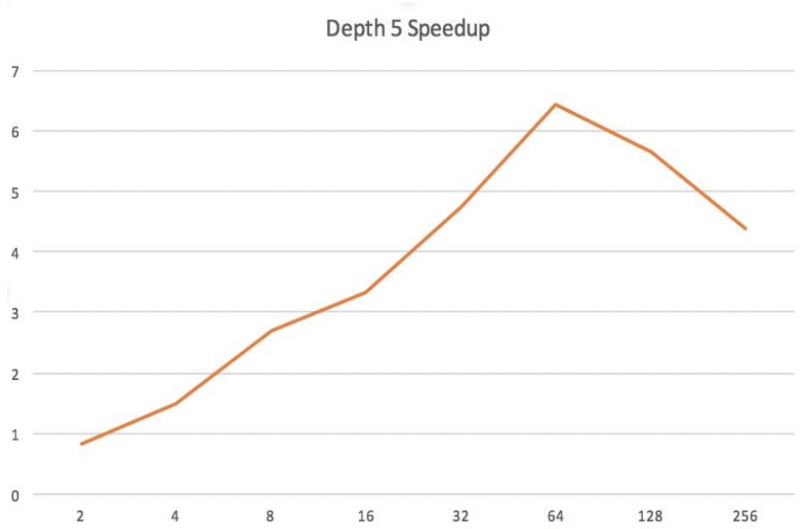

Figure 13: Speedup vs processors at depth 5

Looking at Figure 13 of speedup against number of processors, we can observe a steady increase in speedup with increasing processors as the number of nodes grow exponentially with increase in depth which benefit from more parallelism. For lower depth, the speedup decreases after certain number of processors because the cost of communication and synchronization overshadows benefit of parallelism.

4.4 YBWC (Young Brother Wait Concept)

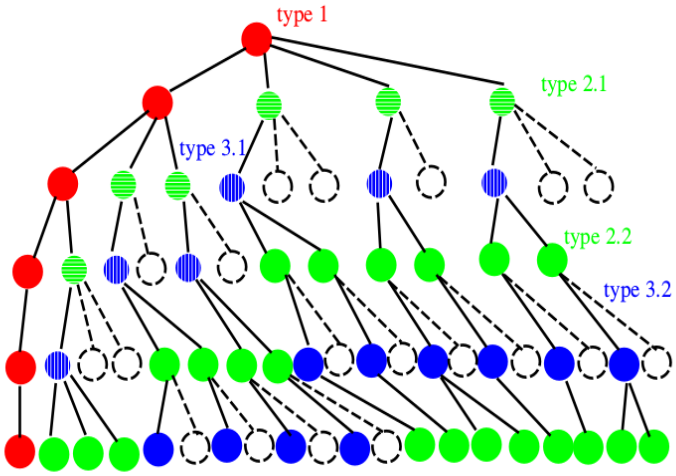

Figure 14: Classification of nodes according to Knuth and Moore

To achieve better parallelization for larger number of processors, we look at Figure 14 classification of nodes into 3 different types by [2]. The type of node is defined according to the rules below:

1. The root of the game tree has **type-1**
2. The first successor of a **type-1** node has **type-1**, the other chilrdren have **type-2**
3. The first successor of a **type-2** node has **type-3**, the other chilrdren have **type-2**
4. All successors of a **type-3** node have **type-2**

To achieve better load balancing, this paper uses the variation of YBWC [1] that follows:

1. Parallel evaluation of the successors of **type-1** node v is allowed only if the first son of v has been completely evaluated
2. Parallel evaluation of the successors of **type-2** node v is allowed only if all the promising successors of v have been completely evaluated
3. Parallel evaluation of the successors of a **type-3** node v is allowed whenever v is generated

A promosing successor is one that is suggested by the transposition table (best move evaluated previously) or killer heuristic.

4.5 Better load distribution strategy

Inspired by the work of [1], this paper tested the same strategy for this project. In general, 3 stages of load distribution are performed to achieve optimal load balancing.

Local load distribution is achieved through request for work from a neighbour of the current idle processor.

Medium range load distribution is triggered when the requested processor does not need any help. In this case, the request is passed to its nearest neighbour until the request has been passed to p processors. In this case, the requesting processor is informed of the cancellation.

Global load distribution leverages on locality of shared transposition table and killer heuristic, master is programmed such that it prefers processors that have previously worked with it.

5 Tabulation of data

5.1 Evaluation criteria for parallel algorithms

$t_n(p)$ is the total time spent to evaluate position p at depth d with processor i.

1. SPE (Speedup)

$$SPE(n) = \frac{\sum_{p \in P} t_1(p)}{\sum_{p \in P} t_n(p)}$$

2. SO (Search Overhead), average number of nodes the parallel version visits more than the sequential version for a set of positions P

$$SO(n) = 100 \cdot \frac{\sum_{p \in P} k_n(p)}{\sum_{p \in P} k_1(p)}$$

3. LD (Processor workload), average percentage of time the processor is busy. $w_n(p)$ is the average amount of time a processor was busy

$$LD(n) = 100 \cdot \frac{\sum_{p \in P} w_n(p)}{\sum_{p \in P} t_n(p)}$$

5.2 Speedup vs depth of tree

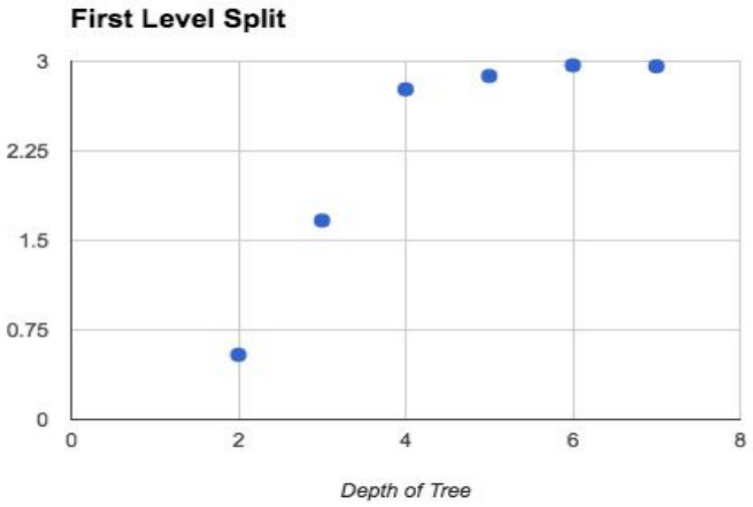

Figure 15: Speedup vs depth of tree for level 1 split

5.3 Per-processor elapsed time

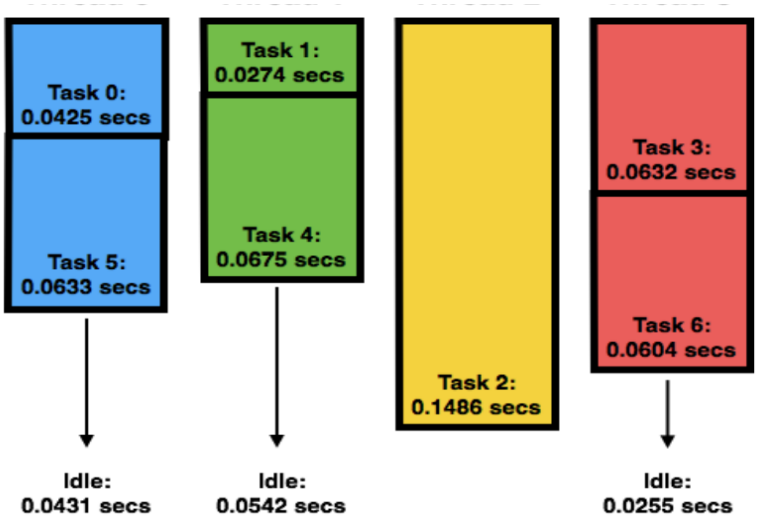

Figure 16: Time spent by each processor

5.4 Efficiency

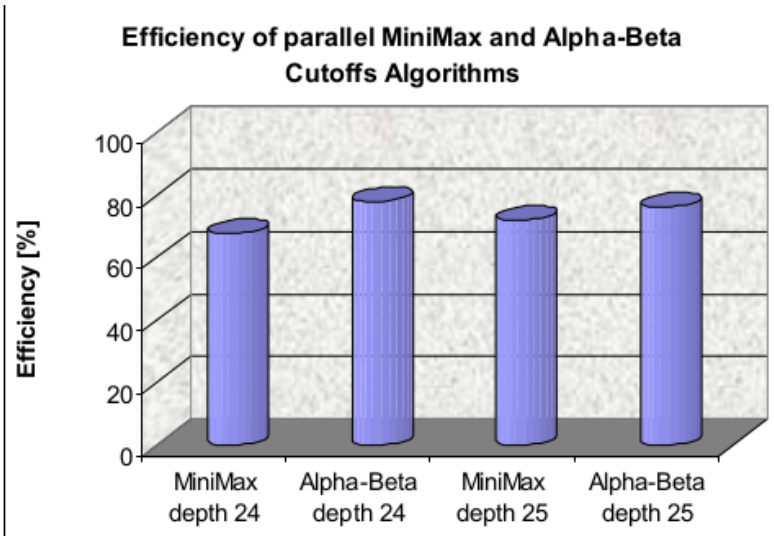

Figure 17: Efficiency of different algorithms

5.5 Performance of parallel algorithm

	SPE	LD	SO	PER	t_Φ
8-ply search					
GC:G(32,16)	176.80	60.99 %	53.52 %	87.12 %	130
GC:G(32,32)	237.15	46.12 %	68.40 %	84.57 %	97
9*-ply search					
GC:G(32,16)	237.03	74.56 %	39.82 %	86.96 %	384
GC:G(32,32)	344.49	62.67 %	55.55 %	83.61 %	264

Figure 18: SPE(speedup), SO(search overhead), Processor work load(LD), Performance per processor(PER), average run time per problem

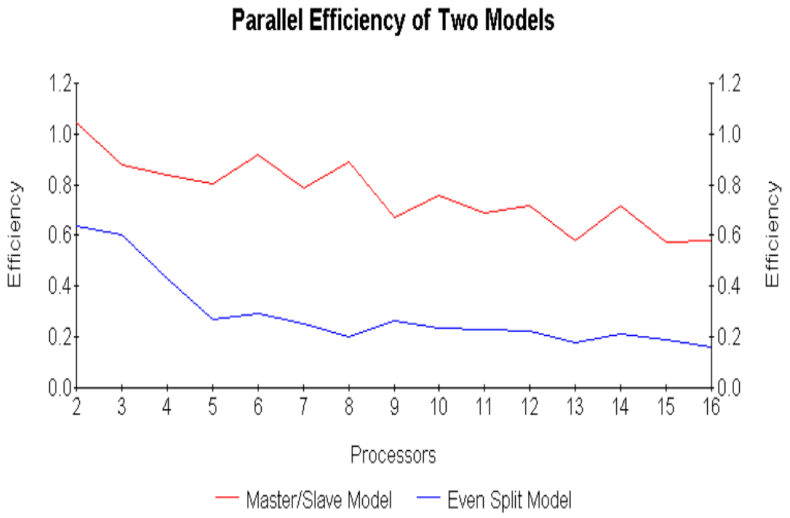

Figure 19: Comparing efficiency for master-slave model and even-split model

5.6 Nodes per second

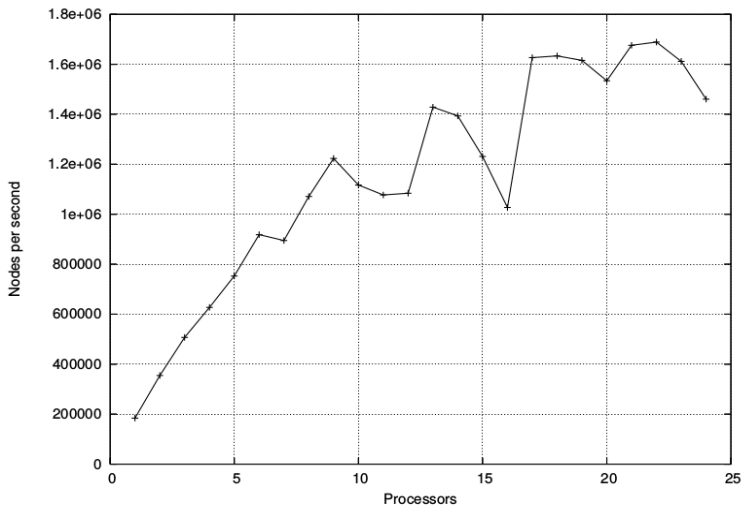

Figure 20: Nodes per second (speed) for different processors

5.7 Board size vs depth

Board dimension	Depth							
	1	2	3	4	5	6	7	8
6 x 6	0.596	0.718	0.429	0.583	1.278	0.891	1.588	2.122
8 x8	1.451	1.612	0.904	2.331	2.469	2.872	3.121	3.501
10 x 10	3.211	3.871	3.532	3.911	-	-	-	-
12 x 12	4.125	4.483	-	-	-	-	-	-

Table 2: Speedup for different board size and depth

5.8 Parallel performance of PV split

Processors	Speedup		Search overhead %		Time idle %	
	Average	Standard deviation	Average	Standard deviation	Average	Standard deviation
2	1.48	0.325	8.73	17.4	12.8	4.90
4	2.03	0.546	24.3	26.3	30.8	8.17
8	2.71	0.803	46.4	46.3	48.8	7.74
16	2.91	1.07	83.5	63.5	67.1	7.00
32	3.23	1.21	94.1	72.5	74.2	6.81
64	4.21	1.45	115.4	81.2	80.1	8.21

Table 3: Parallel performance measure of PV split

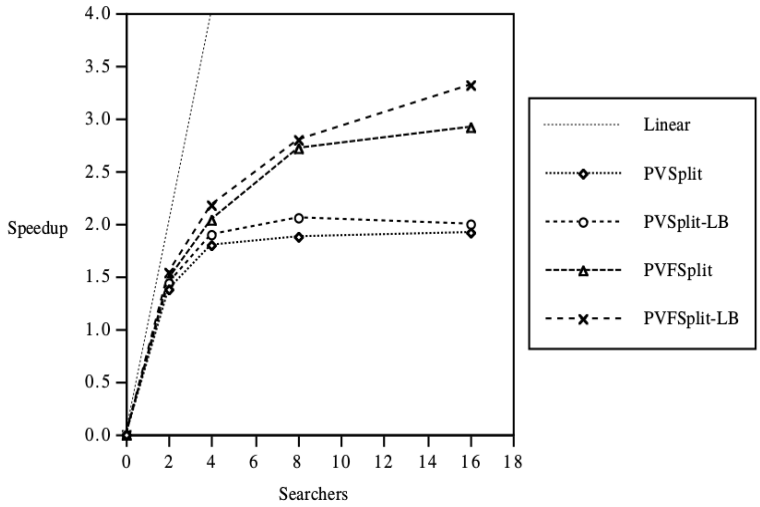

Figure 21: Summary of different parallel algorithms

6 Discussion

6.1 Split model

The result has shown that a master-slave model provides better speedup than the even-split model solving an othello game. The rationale is that master-slave relationship can full use of every processors as processor that has completed can request for more work from the master process.

6.2 Hybrid vs Pure MPI

The impact of hybrid programming is very relevant for our game because the number of nodes grow exponentially with increasing depth which will increase the communciation overhead incurred in message passing. Taking advantage of shared memory's data locality, launching threads in MPI task can help reduce the number of processes.

6.3 Comparison of parallel search

Judging from empirical results, **PV split** achieves considerable speedup for a deep game tree and best-move ordering. However, **YBWC** manages to achieve even better speedup when coupled with the Helpful Master concepty where a master that has nothing to do can help one of the slave node in evaluating a subproblem. In addition, a lockless distributed transposition table also to reduce the synchronization overhead compared to when using a fine-grained locking mechanism.

References

[1] FELDMANN, R. *Game tree search on massively parallel systems*. PhD thesis, Citeseer, 1993.

[2] KNUTH, D. E., AND MOORE, R. W. An analysis of alpha-beta pruning. *Artificial intelligence 6*, 4 (1975), 293–326.

[3] RASMUSSEN, D. Parallel chess searching and bitboards. Master's thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2004.