COMP 534: Project 1: Parallel Exploratory Search using Cilk Plus

Dejan Grubisic

February 16, 2020

Abstract

This project implements an parallel algorithm for game Othello (also known as Reversi) based on Cilk Plus model. The algorithm for creating a move is based on exploratory search of possible moves and evaluating the board position after defined search depth. The rules about this game could be found here [1]. The experiments are done for 1 to 32 cores for the lookahead depth from 1 to 7 moves.

1. **The problem statement and a short motivation for why solving this problem is useful.**

   The game Othello consists of alternating moves selection for two opponent players. In order to find the best move, the algorithm explores all possible moves up to some level and uses Mini-Max algorithm 1 to calculate the best move for a player on each level in the search tree.

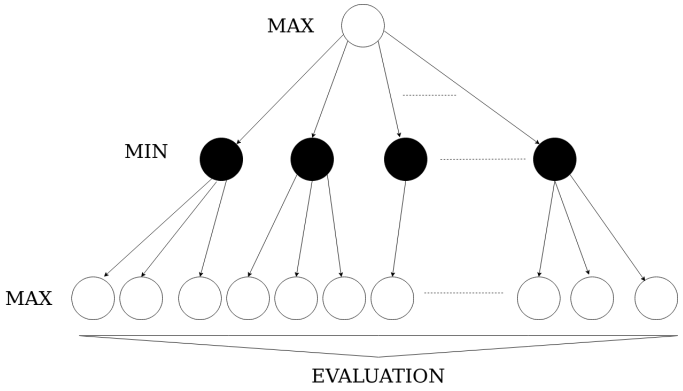

Figure 1: Exploratory search tree

Based on [2] average branching factor for the game Othello is about 10. This means that with each new level in the search tree there are 10 times more boards to explore. Consequentially, the execution time of the sequential implementation grows exponentially and it becomes impossible to deepen the search tree, without some additional technique like alpha-beta pruning [3]. Even with using the pruning, there is still the opportunity to exploit available parallelism and the program even faster. Due to the nature of exploration tree, this problem represents a good example of problem suitable for parallel implementation and understanding the compromises in parallel algorithm design.

2. **The details of your approach and an explanation of how/why this approach solves your problem.**

   In order to maximally utilize available parallelism, it is necessary to spawn tasks as soon as possible at the granularity at the level that will keep all available processors busy. Possible bottleneck could be near the root of the search tree 1, as there are no enough work for all threads. Because of that, the parallel algorithm spawns all tasks (board positions) first N levels. On the other hand, every task introduces some overhead and it is not beneficial to spawn the tasks any further, once we achieve

enough parallelism to employ all threads. Based on empirical evidence, spawning depth = 4 gave the best results.

The idea for the algorithm is to recursively spawn negamax algorithm [4], immediately after the algorithm finds possible move. In this way, the algorithm makes available a new branch in the tree to be explored and shorten the critical path compared to situation when it first finds all possible moves and then spawn computation.

For calculating the best move from a node, the negamax algorithm finds maximum value from all of it's children. To be able to collect the results from the children nodes in parallel and keep deterministic result, the algorithm uses reducers. Every time a node depth is less than specified it creates reducer that collects the results from children nodes. In this implementation the reducer is implemented as cilk::reducer_max_index<Board, int>, so that it collects not only the sco

Pros re of the best move, but also it's board-position. This way it is possible to return to root of the tree the next move position together with it's evaluated value.

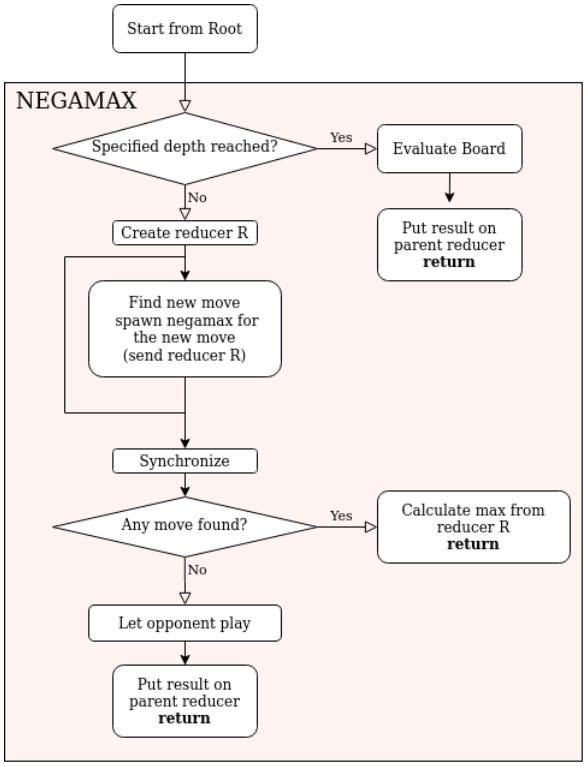

Figure 2: Negamax algorithm

If there is no legal moves, the algorithm needs to check few more conditions. If node is root node and there is no legal moves there is no need to explore the tree at all and the algorithm returns false, which means that opponent plays. If there is no legal moves and there wasn't legal moves for opponent in the move before the algorithm should evaluate the position as that is end of the game. Otherwise, the algorithm searches for opponent move in the tree on the same depth. The sketch of the algorithm is shown on the figure 2.

In this implementation could also parallelize functions like EnumerateLegalMoves that finds neighboring squares to opponent player and FlipDisks that tries to connect that squares with pieces for

given color. In the both cases, parallelization would make even more fine grained task which would increase overhead. Due to this reason, the algorithm implements creates granularity on board level.

The algorithm is additionally improved by implementing the evaluation function such that gives more points to pieces on the edges of the board. This lead to choosing moves that puts pieces on the edge, which make smaller number of possible moves and decreases amount of the work.

3. **A description of the experiments you conducted. Give the quantitative and qualitative analysis of your results.**

   Time Results

   Implemented parallel algorithm is tested on Intel-Xeon-Gold-6126@2.60GHz with 48 CPUs. The first experiment measures the influence of the spawning level [1] on the parallel efficiency. The arallel program has been run on 1 to 16 cores with lookahead 7 and obtained results are shown in figure 3.

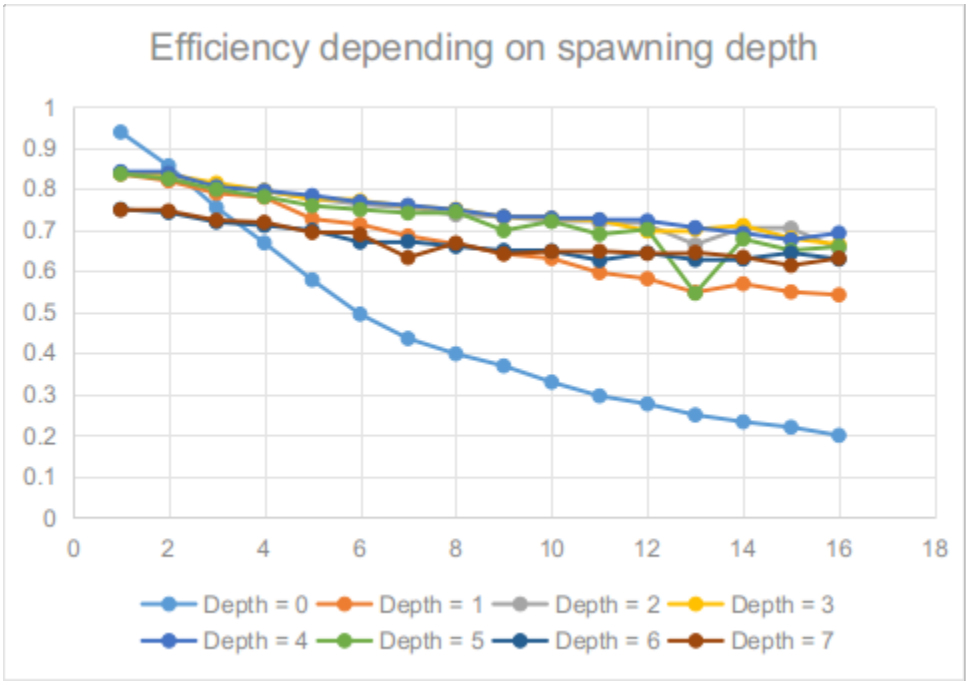

Figure 3: Parallel Efficiency

From the figure 3 we can conclude that the best efficiency is for spawn level 4, and the worst is level 0, which means that only root spawns parallel computation while all other nodes are done sequentially. On the other hand, if the algorithm spawns every computation in parallel, it will again perform worse, due to task overhead (10 % worse than spawn level 4).

In the figure 4 it is represented time for lookahead 7 and spawn level 4. From this figure we can see the benefit of parallelization. By looking at user time, we can conclude that there is no much more additional work as a consequence of parallelization. More precisely it differences only by a constant factor from sequential implementation, which mean that the parallel algorithm is work optimal.

[1] All levels between the root and spawning level are spawned in parallel, while all levels from spawning level to leaves are executed sequentially

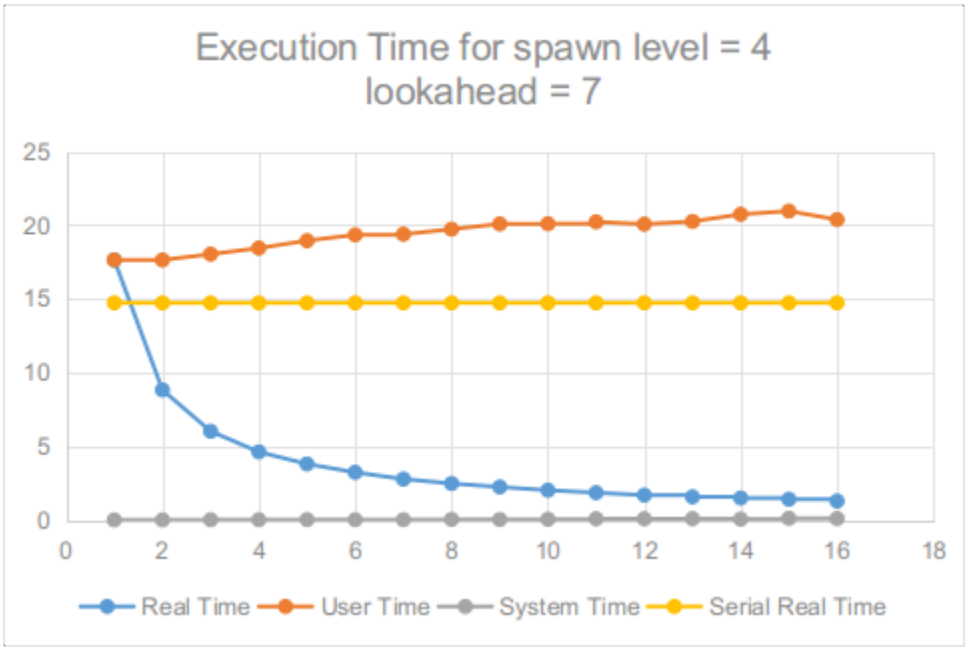

Figure 4: Parallel Efficiency

Cilkviewer Results

The second experiment was based on Cilkview analysis of the program. For this experiment there are measured parallel characteristics of the implemented program with the lookahead depth from 1 to 7 and spawning depth 4 on 8 cores. Table 1 describes whole program statistics. From these results we can see increase of both performed work and parallelism with increasing the lookahead depth. The span is slightly increasing because of the need to collect data deeper in the tree, while burden span has higher increase rate, which is the consequence of task creation overhead. Number of atomic instruction grows the same as the number of spawns / sync because every spawn creates a atomic counter that finds maximum score of children tasks. Average number of instructions per strand has minimum for lookahead 4, because the program spawns every node in parallel for the first 4 levels.

Whole Program Statistics	Lookahead depth						
	1	2	3	4	5	6	7
Work(M):	6.8	15.3	36.4	491	2,165	18,890	150,593
Span(M):	6.6	7.2	7.6	7.9	8.7	8.8	9.4
Burdened span(M):	13.6	26.6	26.3	39.6	45.9	50.6	59.2
Parallelism:	1.02	2.11	4.78	61.51	246	2136	15881
Burdened parallelism:	0.5	0.58	1.39	12.40	47.12	372	2540
Number of spawns/syncs:	464	7,688	19,291	402,215	1,460,755	15,245,951	110,512,623
Avg instr/strand:	4,890	665	630	407	494	413	454
Strands along span:	929	2,459	2,173	3,725	4,153	4,445	4,479
Avg instr/strand(span):	7,179	2,955	3,515	2,145	2,112	1,989	2,117
Tot num atomic instr:	764	7,989	19,592	402,516	1,461,056	15,246,252	110,512,924
Frame count:	928	15,380	38,592	804,444	2,922,018	30,496,812	221,029,264

Table 1: Whole Program Statistics

Cilkview provides also the scalability information which is presented in Table 2. Based on this result we can see that with increase of both lookahead depth and number of processors, rises the estimated speedup. If lookahead depth is small there will be no enough work for all threads. On the other hand, when the number of workers are small it is impossible to get speedup higher than their number, by the Cilk model.

Proc	Lookahead depth						
	1	2	3	4	5	6	7
2	0.45 - 1.02	0.45 - 1.02	0.90 - 2	1.76 - 2	1.90 - 2	1.90 - 2	1.90 - 2
4	0.36 - 1.02	0.41 - 2.11	0.85 - 4	2.83 - 4	3.61 - 4	3.80 - 4	3.80 - 4
8	0.32 - 1.02	0.37 - 2.11	0.83 - 4.78	4.08 - 8	6.39 - 8	7.60 - 8	7.60 - 8
16	0.31 - 1.02	0.35 - 2.11	0.82 - 4.78	5.23 - 16	10.38 - 16	14.98 - 16	15.20 - 16
32	0.30 - 1.02	0.35 - 2.11	0.82 - 4.78	6.09 - 32	15.11 - 32	28.04 - 32	30.40 - 32
64	0.30 - 1.02	0.34 - 2.11	0.82 - 4.78	6.64 - 61.51	19.55 - 64	49.72 - 64	60.80 - 64
128	0.30 - 1.02	0.34 - 2.11	0.82 - 4.78	6.95 - 61	22.93 - 128	81 - 128	117 - 128
256	0.29 - 1.02	0.34 - 2.11	0.82 - 4.78	7.12 - 61.51	25.10 - 246	118 - 256	218 - 256

Table 2: Whole Program Statistics: Speedup

Table 3 gives the result from the analysis of the parallel regions. Work, number of atomic operation and number of spawns remain the same, while parallelism and burden parallelism increases more. Spawn and burden spawn are lower than whole program statistics, because they don't include sequential regions between searching for the moves.

Parallel Region(s) Statistics	Lookahead depth						
	1	2	3	4	5	6	7
Work(M):	0.64	9.2	30.3	485	2.159	18.884	150.587
Span(M):	0.5	1.1	1.4	1.8	2.5	2.6	3.3
Burdened span(M):	7.4	20.4	20.1	33.5	39.7	44.5	53.1
Parallelism:	1.28	8.23	20.75	263.32	839	7034	45304
Burdened parallelism:	0.09	0.45	1.50	14.49	54.31	424.16	2834.59
Number of spawns/syncs:	464	7,688	19,291	402,215	1,460,755	15,245,951	110,512,623
Avg instr/strand:	464	399	523	402	492	412	454
Strands along span:	464	1,229	1,086	1,862	2,076	2,222	2,239
Avg instr/strand(span):	1,086	910	1,344	989	1,239	1,208	1,484
Tot num atomic instr:	764	7,989	19,592	402,516	1,461,056	15,246,252	110,512,924
Frame count:	928	15,380	38,592	804,444	2,922,018	30,496,812	221,029,264
Entries to par reg:	60	60	60	60	60	60	60

Table 3: Parallel Region(s) Statistics

Speedup estimate for parallel Regions statistics (Table 4) mainly differences with whole program statistics by higher increasing of values, especially in situations with higher number of processors and deeper searching tree. This advocates the good utilization of parallelism, when searching depth is higher than 3.

Proc	Lookahead depth						
	1	2	3	4	5	6	7
2	0.10 - 1.28	0.42 - 2	0.94 - 2	1.79 - 2	1.90 - 2	1.90 - 2	1.90 - 2
4	0.07 - 1.28	0.32 - 4	0.91 - 4	2.96 - 4	3.66 - 4	3.80 - 4	3.80 - 4
8	0.06 - 1.28	0.29 - 8	0.90 - 8	4.39 - 8	6.56 - 8	7.60 - 8	7.60 - 8
16	0.05 - 1.28	0.28 - 8.23	0.89 - 16	5.80 - 16	10.89 - 16	15.09 - 16	15.20 - 16
32	0.05 - 1.28	0.27 - 8.23	0.89 - 20.75	6.90 - 32	16.24 - 32	28.46 - 32	30.40 - 32
64	0.05 - 1.28	0.27 - 8.23	0.89 - 20.75	7.63 - 64	21.53 - 64	51.10 - 64	60.80 - 64
128	0.05 - 1.28	0.27 - 8.23	0.89 - 20.75	8.05 - 128	25.73 - 128	84 - 128	118 - 128
256	0.05 - 1.28	0.27 - 8.23	0.88 - 20.75	8.28 - 256	28.50 - 256	126 - 256	222 - 256

Table 4: Parallel Region(s) Statistics: Speedup

Results from HPCToolkit

The third experiment was based on HPCToolkit tool for profiling and collecting the traces from execution with lookahead 7 with spawning depth 4 on 32 cores. The obtained results are presented by using hpcviewer(Figure 5) and hpctraceviewer(Figure 6) visualization tools. The upper part of the figure 5 shows top-down view in hpcviewer that shows the critical part of the code for the performance. In this case that is function TryFlips that is used in finding possible moves. From bottom-up view it can be seen the callpath in which the program spends the majority of its time. In this case it could be seen a chained recursive call to FlipDisk function, that calls TryFlip function. This behavior is expected because the larger amount of the time is spent on provided functions that are used for finding positions in the deeper level in the search tree.

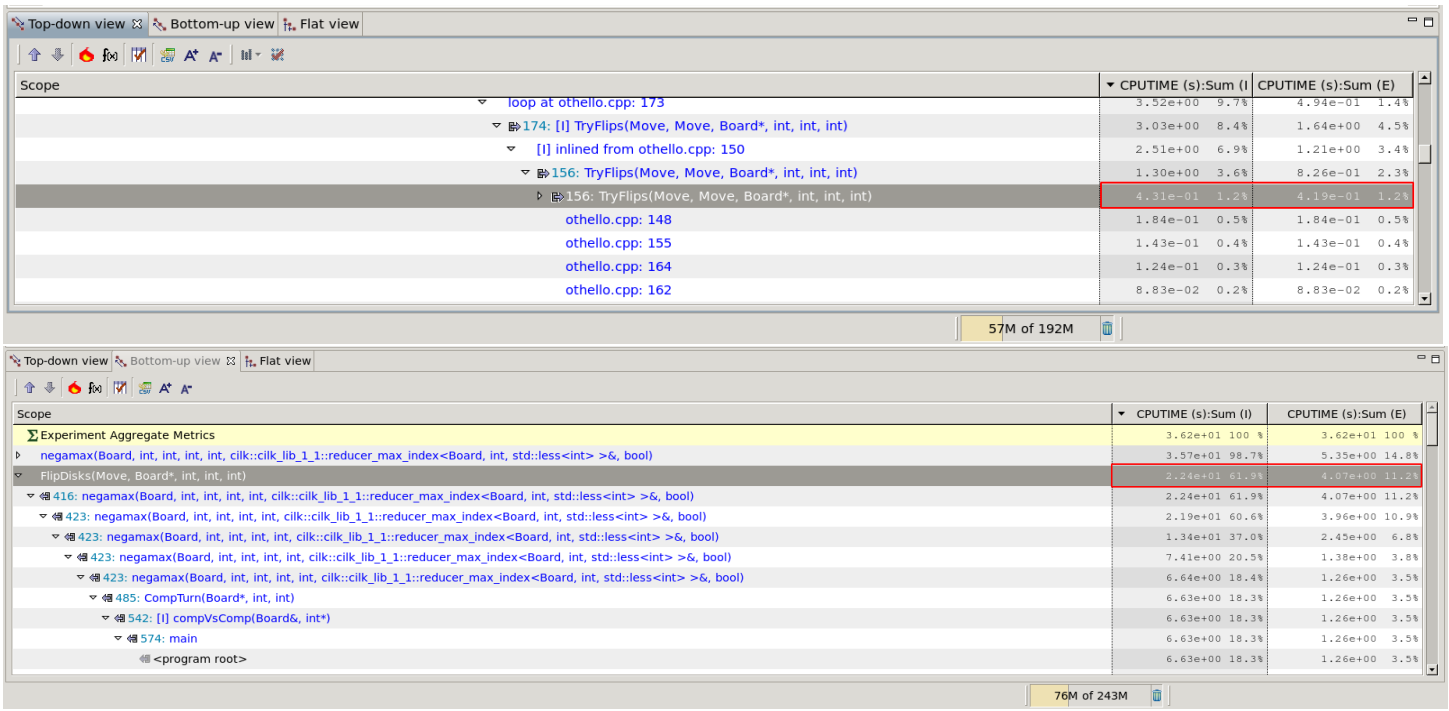

Figure 5: Hpcviewer data

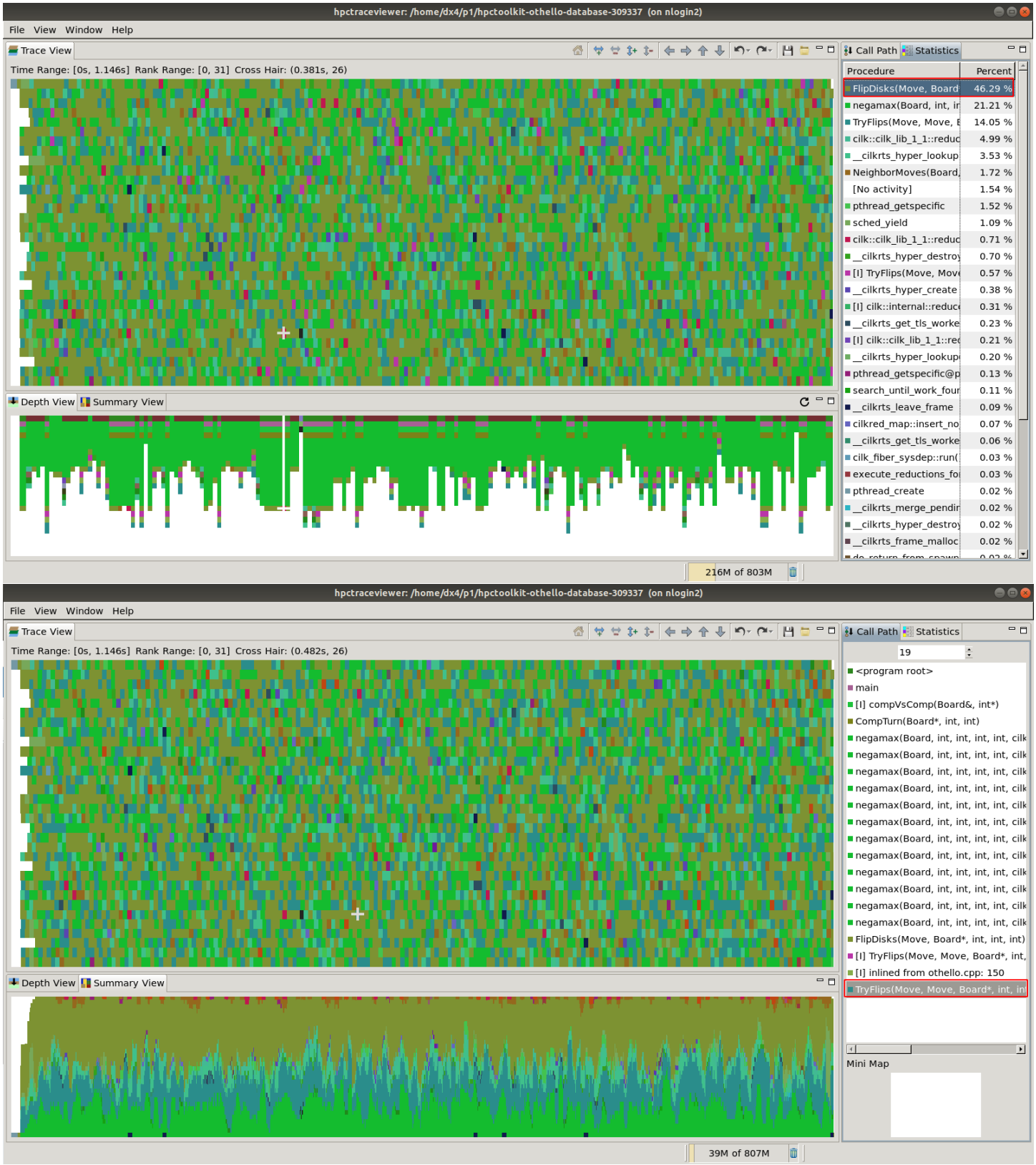

Figure 6: Hpctraceviewer data

Hpctraceviewer provides us with the collected traces and their callstack (Figure 6). From the upper part it could be seen that FlipDisk function takes the most of the execution time. From the Depth View we can see deep recursive call(colored light green) that is consistent with what could be seen from cilkview.From the Summary View on bottom part we can seen the percentage of time spend in functions and once again FlipDisks together with TryFlips dominate execution time.

4. **Conclusion** This project implements and analyses the parallel version of othello board game. In this report there are shown the idea behind parallel algorithm, time results, cilkview analysis and hpctoolkit results. For time analysis, it is performed the experiment that analyse the spawning depth in searching tree with lookahead 7. The experiment is performed for each 1 to 16 cores. The best execution time is achieved with spawning depth 4. The second experiment shows the results from cilkviewer program. These results includes static data about program parallelism. Based on this results the implemented program could be considered as scalable with high expected speedup for the lookahead deeper than 3.

   From the results obtained from hpctoolkit, it could be seen that the main bottleneck is in FlipDisk and TryFlips functions that are calculating possible moves for some position. To make the code even faster it is necessary to optimize the sequential version of these two function. Nevertheless, their parallelisation would not gain any more benefit as they are executed at fine grained level in the every node in the search tree.

References

[1] https://en.wikipedia.org/wiki/Reversi

[2] https://www.sciencedirect.com/topics/computer-science/branching-factor

[3] https://en.wikipedia.org/wiki/Alpha

[4] https://en.wikipedia.org/wiki/Negamax