COMP 534 – Parallel Computing Assignment 1
Swanand Sanjay Khonde – sk218

Othello (Reverse)

The game starts by initializing the board with the starting position of the disks, and we take the input from the user as follows:

1) Who will be the first player (Human or Computer).
2) If the player is computer, then what will be the depth.
3) Who will be the second player (Human or Computer).
4) If the player is computer, then what will be the depth.

The depth search is for the negamax algorithm. In the game loop the computer chooses the move with the help of negamax algorithm. For each move the program checks if the move is legal move or not, updates the board and evaluate the game state to determine if the game has ended or if moves are possible.

The game ends when the board is full or when both the players cannot make any legal move. The winner is determined based on the final score.

Parallelization Strategy

The key elements of the parallelization approach include the use of "cilk::reducer_max" for maintaining the maximum score across the parallel branches and "cilk_for" for parallel loops in the negamax algorithm.

The use of "cilk_for" allows the program to simultaneously explore different legal moves going into different branches of the game tree of given depth in parallel. This helps us reducing the implementation time of the negamax algorithm, where evaluating the outcomes of moves to different depths is computationally expensive and time taking.

The "cilk::reducer_max" helps us calculating the max score efficiently. "cilk::reducer_max" correctly calculates the maximum score in the for loops among all the explored moves in the game tree without running into race conditions.

The use of this parallelization strategy improves the performance of the code and allows the algorithm to search deeper in the tree within small amount if time. The more the depth the better the move.

Pseudo Code for Negamax Algorithm

Function Negamax:
If depth is 0 or the game is over:

- Return the evaluation of the evaluation of the board for the current color.

Initialize an empty list of legal moves for the current color.
Populate the list with all legal moves for current color.
If no legal moves are available:

- Return the negated value of Negamax called with the opponent’s color and decrease the depth by 1.

Define cilk reducer (cilk::reducer_max<int>) for maximum score.
Use a parallel for loop (cilk_for) to iterate over all legal moves:
For each legal move:

- Copy the current board state.
- Apply the move to the copied board state.
- Recursively call Negamax with the new board state, opponent's color, and decreased depth.
- Negate the returned score.
- Use the reducer to update the maximum score if the current score is higher.

Return the value of the maximum score from the reducer.

Exploiting Parallelism using Cilk Plus

The negamax algorithm involves tree exploration with the given depth. I chose to apply parallel constructs to the negamax algorithm loop because different branches of this tree can be evaluated independently, and I think it fits well with the parallel loop and reducer concept. Evaluating all the moves parallelly and giving the maximum score result quickly and efficiently is the requirement for this game. In this way the use of parallelism in the negamax algorithm using cilk_for for the outer loop and using cilk_reducer_max for calculating the maximum score was the best solution according to me. I chose to use cilk_for only in the outer for loop of the negamax as the granularity of the parallel tasks plays a crucial factor in the efficiency of the algorithm. As explained by the professor in class, if the amount of work in each parallel task is too small, the overhead of managing these tasks will surpass the benefit of the parallel execution.

In the Negamax algorithm, changing the outer loop to cilk_for, creates a parallel task for each row of the board, which already is significant amount of work. That too if the depth is greater. That means for 8*8 board, using cilk_for in the outer loop will create 8 tasks that will run in parallel, each will evaluate a row of potential legal moves. Each task has a good amount of work to do. Here the overhead involves creating, scheduling and finally synchronizing 8 tasks. But if I would have chosen to use cilk_for for the inner loop, then there would be a task to evaluate each cell in the game. This would be 64 tasks in total would be created to run in parallel to evaluate the board. The overhead will involve of creating, scheduling and synchronizing 64 tasks for each board evaluation. Also, in the negamax algorithm, the negamax function is called recursively, where each move leads to evaluation of another board. Here the number of tasks can grow exponentially for given depth.

Also considering the load balancing aspect of cilk plus, cilk Plus uses a work-stealing scheduler to dynamically balance the workload across available processor cores. When the outer loop is parallelized, it provides a good balance between creating enough tasks for efficient parallel execution and not overloading the system with too many tasks. The scheduler can effectively distribute these tasks across cores, ensuring good utilization of computational resources.

Opportunities of Parallelism I chose not to Exploit.

I think, enumerate legal moves function could have been the function which could have made use of cilk reducers and cilk_for to calculate the number of moves. I did not get enough time to explore parallelizing that part of the code. It took a lot of time for me to just use “cilk” for the negamax algorithm and then perform all the experiments given in the assignment. I would like to apply parallelism in the enumerate legal moves function even after submitting the assignment to check if the performance and the speed efficiency increases. But I think the complexity of the code will increase if we apply parallelism in enumerate legal moves function. Especially regarding data synchronization and keeping the game state consistent.

Explaining Parallel Work Synchronization.

Parallel work is synchronized primarily using cilk_for and cilk::reducer_max. In the Negamax algorithm function synchronization of the parallel work is very important to calculate the maximum score before returning it.

Cilk_for loop: the cilk plus runtime ensures that all parallel tasks are spawned within the loop and completed before proceeding to the next statement after the loop ends. This means all the recursive negamax calls within the cilk_for loop will execute parallely and finish its execution and return the score before the loop completes. Cilk plus runtime waits for all the parallel tasks to complete before moving on, ensuring that we get all the evaluated scores for the next step in the algorithm.

Cilk::reducer_max: This object is used to safely compute the max score across all the parallel tasks. Each thread will update its score to the reducer thread-local views. Upon completion of the cilk_for loop the Cilk Plus runtime system combines these thread-local views to compute the final maximum score. Cilk plus runtime ensures that the maximum score computed reflects the outcomes of all parallel tasks, synchronized correctly across threads.

Output of Cilkview for Lookahead Depths 1-7

Whole Program Statistics

Lookahead Depth	Work	Parallelism	Number of spawns/syncs	Speedup Estimate (16 processors)	Elapsed time
1	-	-	-	-	-
2	19,009,298 instructions	1.15	3,224	0.22 - 1.15	0.032 seconds
3	83,138,348 instructions	2.38	26,136	0.29 - 2.38	0.032 seconds
4	307,097,029 instructions	6.08	118,240	0.60 - 6.08	0.031 seconds
5	4,766,302,492 instructions	20.11	1,635,656	1.68 - 16.00	0.034 seconds
6	36,656,567,435 instructions	65.90	12,954,344	4.38 - 16.00	0.030 seconds
7	1,042,778,024,109 instructions	426.17	324,374,096	11.65 - 16.00	0.030 seconds

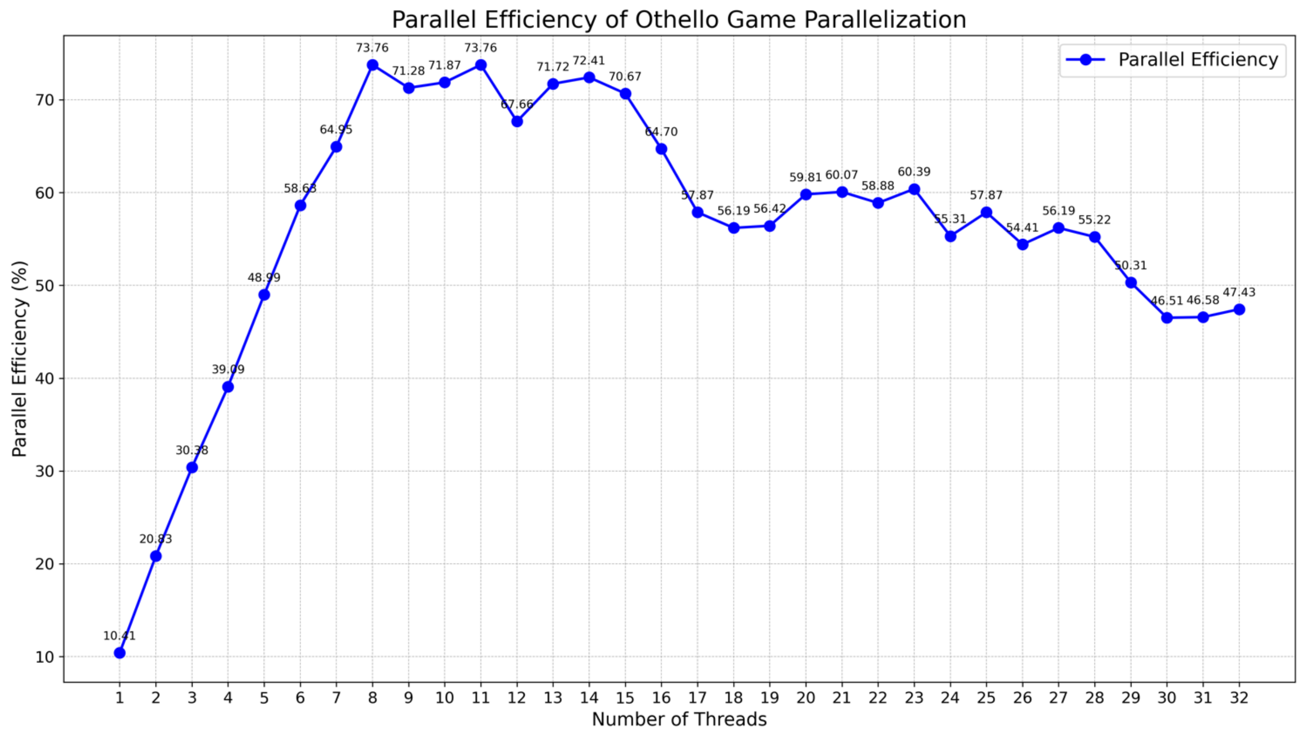
Cilk Parallel Region(s) Statistics

Lookahead Depth	Work	Entries to parallel region	Average instructions / strand on span	Frame Count	Elapsed time
1	-	-	-	-	-
2	3,816,137 instructions	403	863	6,448	0.032 seconds
3	67,827,476 instructions	390	3,266	55,149	0.032 seconds
4	293,251,770 instructions	319	3,396	250,941	0.031 seconds
5	4,751,119,737 instructions	392	3,938	3,475,377	0.034 seconds
6	36,642,494,794 instructions	349	4,001	27,527,632	0.030 seconds
7	1,042,761,765,578 instructions	441	4,605	689,294,513	0.030 seconds

As the lookahead depth increases, the amount of work increases exponentially. This is expected because each additional level of depth in the game tree adds multiple new game states to evaluate. An increase in lookahead depth leads to more opportunities for parallel execution, as seen by the increase in this metric. However, the parallelism doesn't scale linearly due to the nature of the game tree where not all nodes have the same number of children. While the upper bound of the speedup increases with the lookahead depth, it doesn't increase as fast as the work does. This is because as program gets the benefit of more processors but has diminishing results due to overheads like synchronization and non-uniformity of the game.

We can conclude that parallelization scales with the lookahead depth in the beginning, the program becomes inefficient at higher depths.

Parallel Efficiency Graph



The formula to calculate parallel Efficiency is given by S/(p * T(p)) *100.

S = 110 seconds
P = 8 processors
For T(p) for 32 threads, the table is as follows:

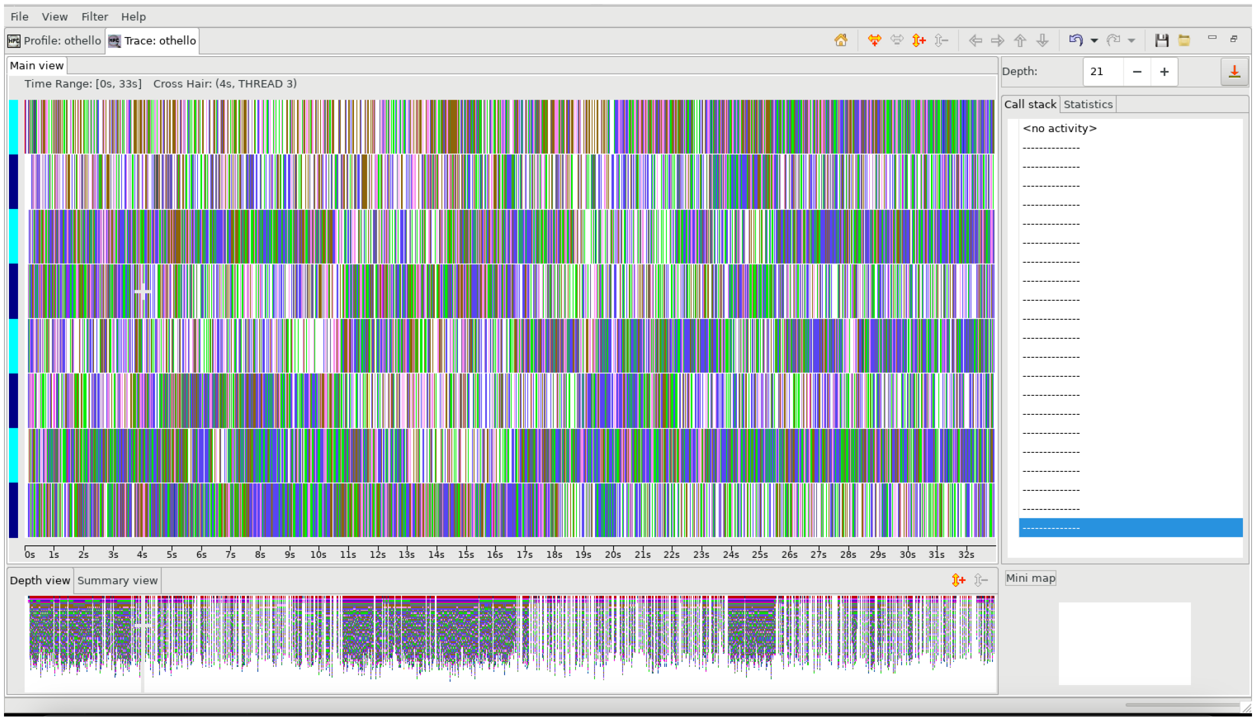
N	1	2	3	4	5	6	7	8
S	110	110	110	110	110	110	110	110
P	8	8	8	8	8	8	8	8
T(P)	10.41	20.83	30.38	39.09	48.99	58.63	64.95	73.76

N	9	10	11	12	13	14	15	16
S	110	110	110	110	110	110	110	110
P	8	8	8	8	8	8	8	8
T(P)	71.28	71.87	73.76	67.66	71.72	72.41	70.67	64.7

N	17	18	19	20	21	22	23	24
S	110	110	110	110	110	110	110	110
P	8	8	8	8	8	8	8	8
T(P)	57.87	56.19	56.42	59.81	60.07	58.88	60.39	55.31

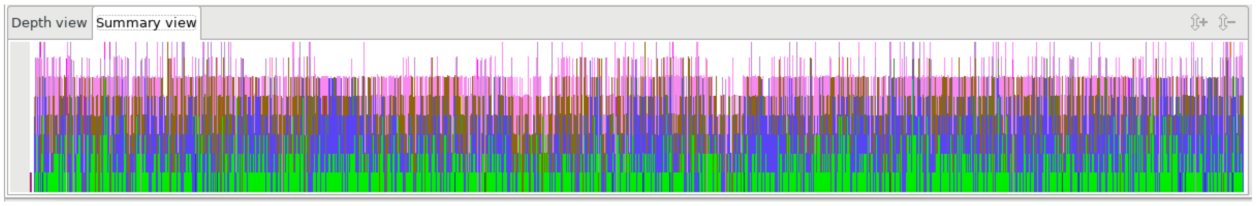
N	25	26	27	28	29	30	31	32
S	110	110	110	110	110	110	110	110
P	8	8	8	8	8	8	8	8
T(P)	57.87	54.41	56.19	55.22	50.31	46.51	46.58	47.43

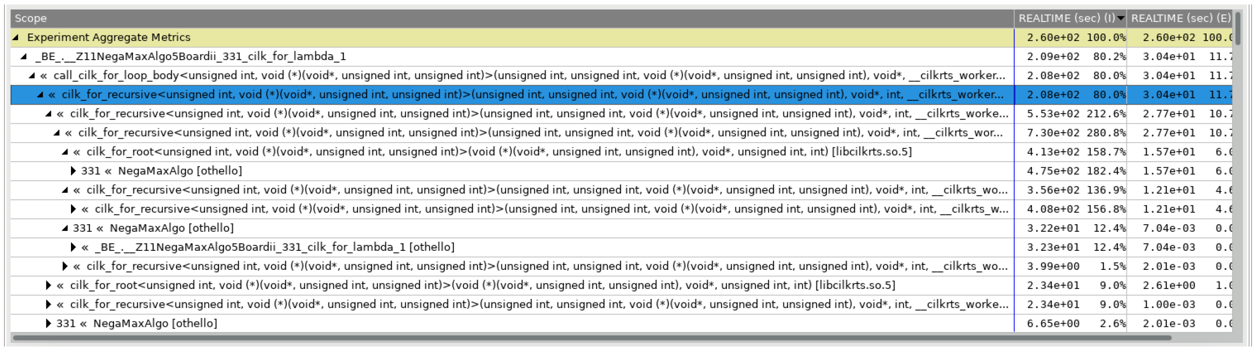
HPCTOOLKIT RESULTS



Time Range: [0s, 33s]   Cross Hair: (4s, THREAD 3)

SUMMARY VIEW





Scope	REALTIME (sec) (I)		REALTIME (sec) (E)	
◢ Experiment Aggregate Metrics	2.60e+02	100.0%	2.60e+02	100.0
◢ _BE_.__Z11NegaMaxAlgo5Boardii_331_cilk_for_lambda_1	2.09e+02	80.2%	3.04e+01	11.7
◢ « call_cilk_for_loop_body<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, __cilkrts_worker...	2.08e+02	80.0%	3.04e+01	11.7
◢ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_worker...	2.08e+02	80.0%	3.04e+01	11.7
◢ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_worke...	5.53e+02	212.6%	2.77e+01	10.7
◢ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_wor...	7.30e+02	280.8%	2.77e+01	10.7
◢ « cilk_for_root<unsigned int, void (*)(void*, unsigned int, unsigned int)>(void (*)(void*, unsigned int, unsigned int), void*, unsigned int, int) [libcilkrts.so.5]	4.13e+02	158.7%	1.57e+01	6.0
▶ 331 « NegaMaxAlgo [othello]	4.75e+02	182.4%	1.57e+01	6.0
◢ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_wo...	3.56e+02	136.9%	1.21e+01	4.6
▶ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_w...	4.08e+02	156.8%	1.21e+01	4.6
◢ 331 « NegaMaxAlgo [othello]	3.22e+01	12.4%	7.04e-03	0.0
▶ « _BE_.__Z11NegaMaxAlgo5Boardii_331_cilk_for_lambda_1 [othello]	3.23e+01	12.4%	7.04e-03	0.0
▶ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_wo...	3.99e+00	1.5%	2.01e-03	0.0
▶ « cilk_for_root<unsigned int, void (*)(void*, unsigned int, unsigned int)>(void (*)(void*, unsigned int, unsigned int), void*, unsigned int, int) [libcilkrts.so.5]	2.34e+01	9.0%	2.61e+00	1.0
▶ « cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>(unsigned int, unsigned int, void (*)(void*, unsigned int, unsigned int), void*, int, __cilkrts_worke...	2.34e+01	9.0%	1.00e-03	0.0
▶ 331 « NegaMaxAlgo [othello]	6.65e+00	2.6%	2.01e-03	0.0