Assignment 1 Report

Yi Zhou, yz166

1. Program Description

The program runs the minimax function for a computer player. In the minimax function, it calls the recursive function until reaching the searching depth to get the disk placement position according to the maximum or minimum value calculated on the board. On each depth moves, the disk can further expand trees of large amount of moves until reaches the terminal state and this is where the parallelism applies to the program for improving its performance. The pseudo-code with key elements of the parallelization strategy is given below:

```
<score, Move> minimax_value():
        if is terminal state:
                score = evaluate_board();
                return <score, Move>
        if max player:
                cilk::reducer_max best
                cilk_for legal_moves:
                        place & flip
                        <score, Move> = minimax_value()
                return best.calc_max()
        else:
                cilk::reducer_min best
                cilk_for legal_moves:
                        place & flip
                        <score, Move> = minimax_value()
                return best.calc_min()
int minimax():
        if no moves:
                check the other player moves
        <score, Move> = minimax_value()
        Place & flip
```

Figure 1 pseudo-code

Since the recursion for each legal move at n depth can be treated as a parent node and child nodes, it is proper for the program to exploit the parallelism on it. In this way, tasks on each node will be given to available threads and fully utilize their capacity.

Based on my implemented minimax algorithm, there does not seem to have any better opportunities for other parallelism to be exploited. Because disk moves on each search depth depend on the ones on the previous depth, cilk_for is better to handle this

traversing situation than cilk_spawn. The parallel work is auto-synchronized by cilk_for after all iterations for legal moves are completed.

2. Cilkview Discussion

The table1 is a list of cilkview reports for lookhahead depths 1-7.

Table 1 cilkview for lookahead depths 1-7

depth	1 1	2 2	3 3	4 4	5 5	6 6	7 7
Whole Program Statistics							
1) Parallelism Profile							
Work	10,114,325 instructions	17,158,104 instructions	101,841,612 instructions	1,038,538,466 instructions	7,453,760,703 instructions	22,791,515,727 instructions	1,047,357,478,564 instructions
Span	9,813,381 instructions	10,697,459 instructions	11,662,786 instructions	12,422,430 instructions	12,566,664 instructions	13,210,733 instructions	13,623,504 instructions
Burdened span	14,432,361 instructions	21,523,472 instructions	28,888,455 instructions	36,183,723 instructions	41,182,941 instructions	47,172,598 instructions	55,814,671 instructions
Parallelism	1.03	1.60	8.73	83.60	593.14	1725.23	76878.71
Burdened parallelism	0.70	0.80	3.53	28.70	180.99	483.23	18764.91
Number of spawns/syncs	388	3,924	47,200	531,622	4,100,679	11,593,878	568,582,675
Average instructions / strand	8,681	1,457	719	651	605	655	614
Strands along span	407	917	1,417	1,941	2,357	2,695	3,365
Average instructions / strand on span	24,111	11,665	8,230	6,400	5,331	4,901	4,048
Total number of atomic instructions	698	4,875	64,581	854,203	6,784,670	11,594,059	945,130,556
Frame count	776	8,275	99,108	1,115,586	8,599,627	24,589,801	1,190,700,446

2) Speedup Estimate							
2 processors	0.58 - 1.03	0.64 - 1.60	1.35 - 2.00	1.89 - 2.00	1.90 - 2.00	1.90 - 2.00	1.90 - 2.00
4 processors	0.48 - 1.03	0.54 - 1.60	1.63 - 4.00	3.40 - 4.00	3.80 - 4.00	3.80 - 4.00	3.80 - 4.00
8 processors	0.44 - 1.03	0.50 - 1.60	1.83 - 8.00	5.66 - 8.00	7.51 - 8.00	7.60 - 8.00	7.60 - 8.00
16 processors	0.43 - 1.03	0.49 - 1.60	1.94 - 8.73	8.47 - 16.00	14.02 - 16.00	15.20 - 16.00	15.20 - 16.00
32 processors	0.42 - 1.03	0.48 - 1.60	2.01 - 8.73	11.28 - 32.00	24.78 - 32.00	28.85 - 32.00	30.40 - 32.00
64 processors	0.42 - 1.03	0.47 - 1.60	2.04 - 8.73	13.53 - 64.00	40.21 - 64.00	52.39 - 64.00	60.80 - 64.00
128 processors	0.41 - 1.03	0.47 - 1.60	2.06 - 8.73	15.02 - 83.60	58.37 - 128.00	88.47 - 128.00	121.60 - 128.00
256 processors	0.41 - 1.03	0.47 - 1.60	2.07 - 8.73	15.90 - 83.60	75.40 - 256.60	134.93 - 256.60	243.20 - 256.00
Cilk Parallel Region(s) Statistics							
Elapsed time	0.027 seconds	0.031 seconds	0.031 seconds	0.034 seconds	0.032 seconds	0.031 seconds	0.031 seconds
1) Parallelism Profile							
Work	451,694 instructions	7,170,540 instructions	91,590,280 instructions	1,028,135,960 instructions	7,443,638,015 instructions	22,781,201,310 instructions	1,047,347,627,170 instructions
Span	150,750 instructions	709,895 instructions	1,411,454 instructions	2,019,924 instructions	2,443,976 instructions	2,896,316 instructions	3,772,110 instructions
Burdened span	4,769,730 instructions	11,535,908 instructions	18,637,123 instructions	25,781,217 instructions	31,060,253 instructions	36,858,181 instructions	45,963,277 instructions
Parallelism	3.00	10.10	64.89	509.00	3045.71	7865.58	277655.64
Burdened parallelism	0.09	0.62	4.91	39.88	239.65	618.08	22786.62
Number of spawns/syncs	388	3,924	47,200	531,622	4,100,679	11,593,878	568,582,675

Average instructions / strand	387	609	646	644	605	654	614
Strands along span	203	458	708	970	1,178	1,347	1,682
Average instructions / strand on span	742	1,549	1,993	2,082	2,074	2,150	2,242
Total number of atomic instructions	698	4,875	64,581	854,203	6,784,670	11,594,059	945,130,556
Frame count	776	8,275	99,108	1,115,586	8,599,627	24,589,801	1,190,700,446
Entries to parallel region	60	60	60	60	60	60	60
2) Speedup Estimate							
2 processors	0.11 - 2.00	0.54 - 2.00	1.49 - 2.00	1.90 - 2.00	1.90 - 2.00	1.90 - 2.00	1.90 - 2.00
4 processors	0.07 - 3.00	0.43 - 4.00	1.96 - 4.00	3.55 - 4.00	3.80 - 4.00	3.80 - 4.00	3.80 - 4.00
8 processors	0.06 - 3.00	0.40 - 8.00	2.34 - 8.00	6.16 - 8.00	7.60 - 8.00	7.60 - 8.00	7.60 - 8.00
16 processors	0.06 - 3.00	0.38 - 10.10	2.59 - 16.00	9.76 - 16.00	14.46 - 16.00	15.20 - 16.00	15.20 - 16.00
32 processors	0.06 - 3.00	0.37 - 10.10	2.73 - 32.00	13.78 - 32.00	26.23 - 32.00	29.49 - 32.00	30.40 - 32.00
64 processors	0.06 - 3.00	0.37 - 10.10	2.81 - 64.00	17.36 - 64.00	44.23 - 64.00	54.55 - 64.00	60.80 - 64.00
128 processors	0.06 - 3.00	0.37 - 10.10	2.85 - 64.00	19.96 - 128.00	67.34 - 128.00	94.86 - 128.00	121.60 - 128.00
256 processors	0.06 - 3.00	0.37 - 10.10	2.87 - 64.00	21.57 - 256.00	91.14 - 256.00	150.47 - 256.00	243.20 - 256.00

From table 1, it can be observe that for Whole Program Statistics, everything is increasing, except average instructions and average instructions/strand on span. However, for the Cilk Program Region(s) Statistics, every analyzed data are increasing as the lookahead depth increases. This is because for the whole program, instead of running serialized in one chain, distributes tasks to threads (different chains). With the number of

lookahead depth increases, program uses more threads for handling each depth tasks, which makes the amount of average instructions in the cilk region becomes larger. Each thread execute the passed tasks and spawn other new threads for keeping parallelism, which cause the average instructions on the span decreases.

3. Parallel Performance

The table 2 records the real, user, and system time for running the app on 1 to 32 threads and figure2 plots the parallel efficiency when the program runs on 16 processors comparing and using different number of threads.

Table 2. The real, user, and system time for the run on 1-32 threads

# threads	real	user	system time
1	2m14.309s	2m13.966s	0m0.012s
2	1m22.579s	2m44.714s	0m0.015s
3	0m54.318s	2m42.475s	0m0.015s
4	0m42.841s	2m50.834s	0m0.027s
5	0m35.221s	2m55.552s	0m0.029s
6	0m30.125s	3m0.141s	0m0.032s
7	0m26.559s	3m5.272s	0m0.061s
8	0m22.870s	3m2.135s	0m0.070s
9	0m22.009s	3m17.313s	0m0.067s
10	0m20.170s	3m20.886s	0m0.098s
11	0m19.518s	3m33.823s	0m0.083s
12	0m17.920s	3m34.075s	0m0.126s
13	0m17.310s	3m44.009s	0m0.118s
14	0m16.432s	3m48.992s	0m0.124s
15	0m15.442s	3m50.543s	0m0.145s
16	0m14.401s	3m49.300s	0m0.148s
17	0m13.801s	3m53.424s	0m0.190s
18	0m12.511s	3m40.352s	0m0.174s
19	0m12.034s	3m46.094s	0m0.214s
20	0m11.926s	3m57.234s	0m0.162s
21	0m11.958s	4m9.379s	0m0.196s
22	0m11.535s	4m3.178s	0m0.256s
23	0m11.578s	4m20.565s	0m0.262s
24	0m11.364s	4m26.636s	0m0.361s
25	0m10.547s	4m3.553s	0m0.272s
26	0m10.719s	4m23.527s	0m0.295s
27	0m11.203s	4m29.312s	0m0.366s
28	0m10.505s	4m29.046s	0m0.312s
29	0m10.084s	4m49.433s	0m0.264s
30	0m10.269s	4m40.008s	0m0.357s

31	0m10.018s	4m44.534s	0m1.410s
32	0m9.968s	4m54.109s	0m0.552s

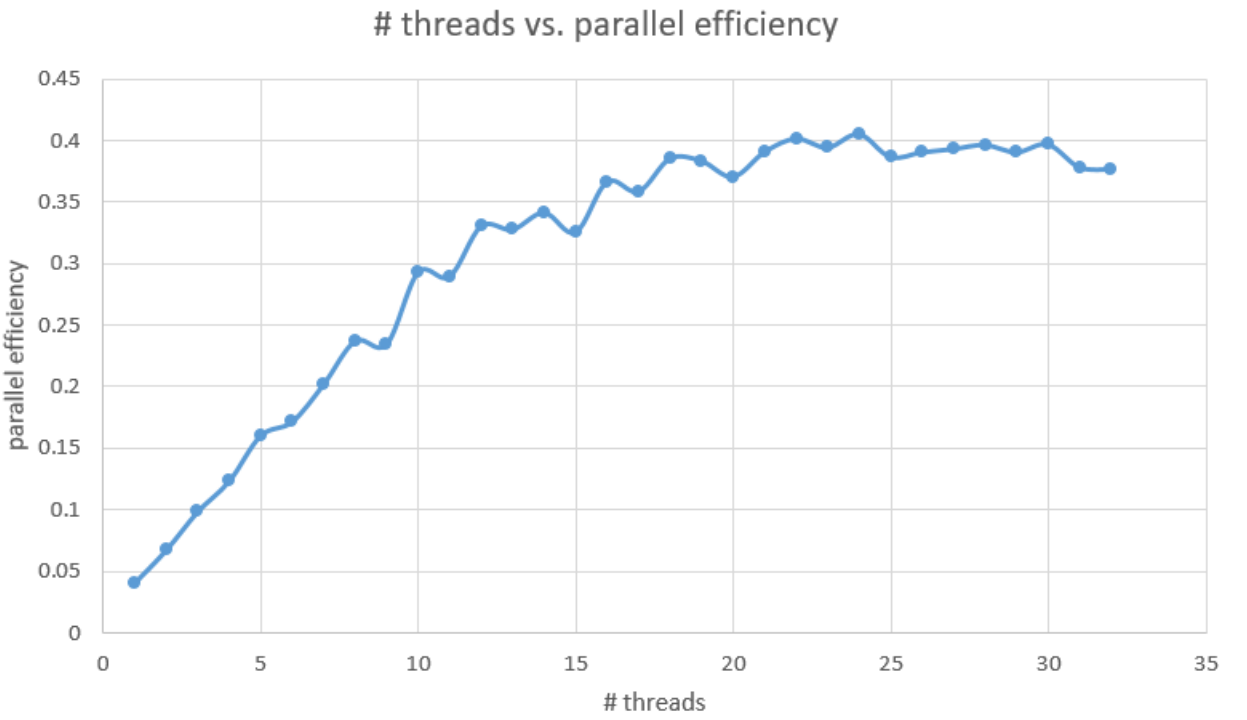

Figure 2 # threads vs parallel efficiency

From table 2, it can be observed that the real execution time for the program is decreased as the number of threads used increases. The execution time decrease dramatically until running on up to 22 threads, which implies a well parallelization. After that, the speedup performance gradually diminishes. Besides, the user and system time becomes longer, since more threads are required for the CPU to run the program. This finding can also be seen from the plot. As the number of threads in the x-axis becomes larger, the program's calculated parallel efficiency has a rapid upward trend, and gradually tends to be flat with small fluctuations when it reaches a certain peak value. The best value for parallel efficiency is around 0.40, so the program has an average quality of parallelization.