Assignment 1

1. Simple Recursive Solution

Brief introduction of the solution

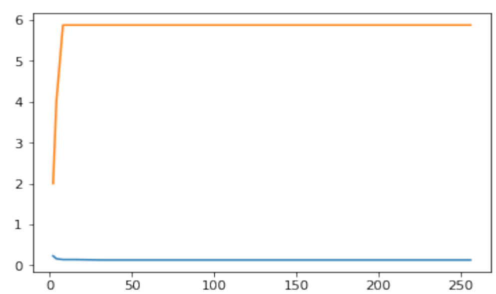
I used the maxmin algorithm to decide which move is the best. A tree can be used to represent all the possible boards after some legal moves. The height of the tree is decided by the search level. Every node has a score which is decided by its children, except for the leaf nodes whose score is decided by the count of bits of the according color. So this can be easily solved by a recursive algorithm as following:

FindBestMove(node) {
    score = Nan
    IF node is a leaf node
        Return count of the right color
    ELSE
        LOOP for every children node:
            update score according to FindBestMove(children node)
        END LOOP
    END
    RETURN score
}

To make this algorithm parallel, we can simply use cilk_for to let the program search different branches of the tree at the same time. We also need to use a reducer to remember both the best score during the searching. Actually what we need is the best move, so the reducer I use is op_max_index and op_min_index. Because the board is 8*8, every move can be represent by a 64 bit integer. And I consider this integer to be the index of the max/min score.

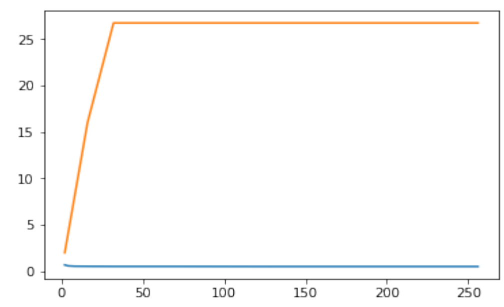
Cilkview profiles analysis

Since the table is too long, I only show the Cilk Parallel Region part:

Level	Parallelism Profile	Speed Estimate
1	Work 2,589,132 Span 440,008 Burdened span 11,925,089 Parallelism 5.88 Burdened parallelism 0.22 Number of spawns/syncs 2896 Average / strand 297 Strands along span 480 Average / strand on span 916 Total number of atomic 3197 Frame count 6094 Entries to parallel region 60	

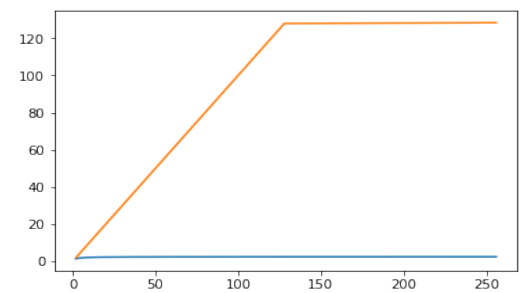
2

Work	21,386,301
Span	800,100
Burdened span	24,114,852
Parallelism	26.73
Burdened parallelism	0.89
Number of spawns/syncs	26216
Average / strand	271
Strands along span	956
Average / strand on span	836
Total number of atomic	26517
Frame count	55649
Entries to parallel region	60



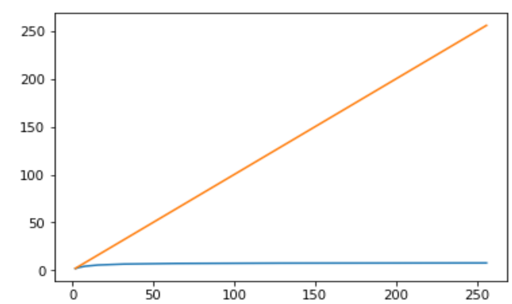
3

Work	154,739,121
Span	1,204,837
Burdened span	36,025,094
Parallelism	128.43
Burdened parallelism	4.3
Number of spawns/syncs	194632
Average / strand	265
Strands along span	1428
Average / strand on span	843
Total number of atomic	194933
Frame count	413533
Entries to parallel region	60



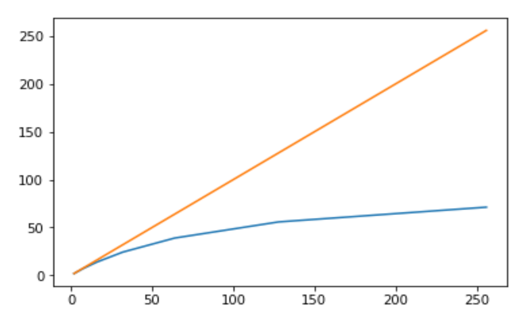
4

Work	645,375,394
Span	1,522,881
Burdened span	48,119,202
Parallelism	423.79
Burdened parallelism	13.41
Number of spawns/syncs	887280
Average / strand	242
Strands along span	1896
Average / strand on span	803
Total number of atomic	887581
Frame count	1885410
Entries to parallel region	60



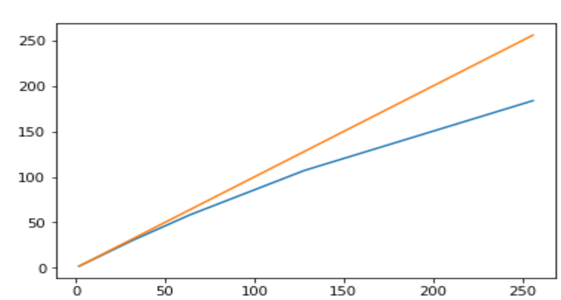
5

Work	9,992,509,676
Span	2,016,643
Burdened span	59,745,042
Parallelism	4955.02
Burdened parallelism	167.25
Number of spawns/syncs	11955960
Average / strand	278
Strands along span	2364
Average / strand on span	853
Total number of atomic	11956261
Frame count	25406355
Entries to parallel region	60



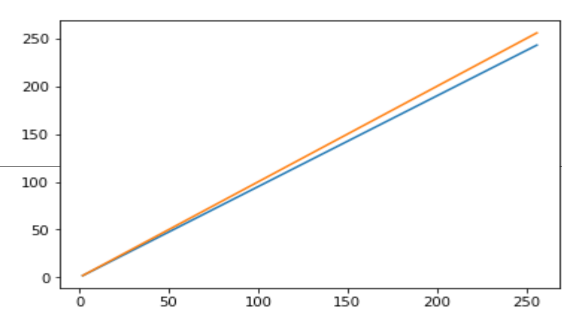
6

Work	79,496,532,634
Span	2,310,359
Burdened span	71,855,804
Parallelism	34408.74
Burdened parallelism	1106.33
Number of spawns/syncs	99422200
Average / strand	266
Strands along span	2820
Average / strand on span	819
Total number of atomic	99422501
Frame count	211272115
Entries to parallel region	60



7

Work	2,234,513,357,696
Span	2,611,369
Burdened span	83,662,516
Parallelism	855686.56
Burdened parallelism	26708.66
Number of spawns/syncs	2579520768
Average / strand	288
Strands along span	3276
Average / strand on span	797
Total number of atomic	2579521069
Frame count	5481481572
Entries to parallel region	60



We can see that when the search level is low, there are less parallelism in our program (the burden parallelism is less than 8 which means at most we only need 8 cores), so the heavy overheads makes the parallel program slower than serial version (the speed up < 1). When the search level is high, we have enough parallelism and get a linear speed up. The results above suggest me that it is better to do serial calculation when the search level is less than 4.

2. Upgrade the Solution with Truncating

The brief introduction of truncating

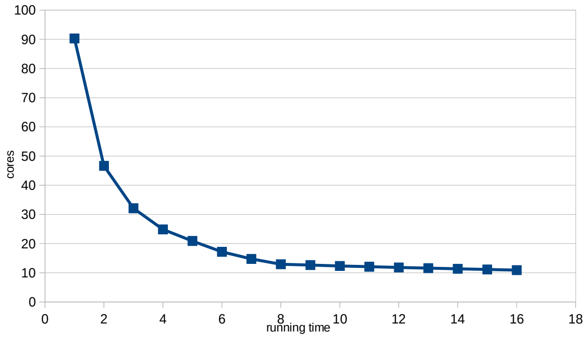
I defined a serial version of s_FindBestMove and a parallel version of p_FindBestMove. The pseudo-code is:

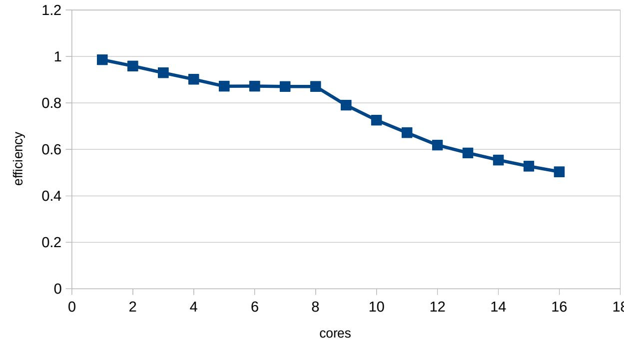
```
p_FindBestMove(node) {
    score = Nan
    IF level <= 3
        update score according to s_FindBestMove(node)
    ELSE
        CILK LOOP for every children node:
            IF level <= 4
                update score according to s_FindBestMove(children node)
            ELSE
                update score according to p_FindBestMove(children node)
            END
        END LOOP
    RETURN score
}
```

s_FindBestMove(node) is as same as the FindBestMove function is section 1.

The parallel efficiency of solution with truncating

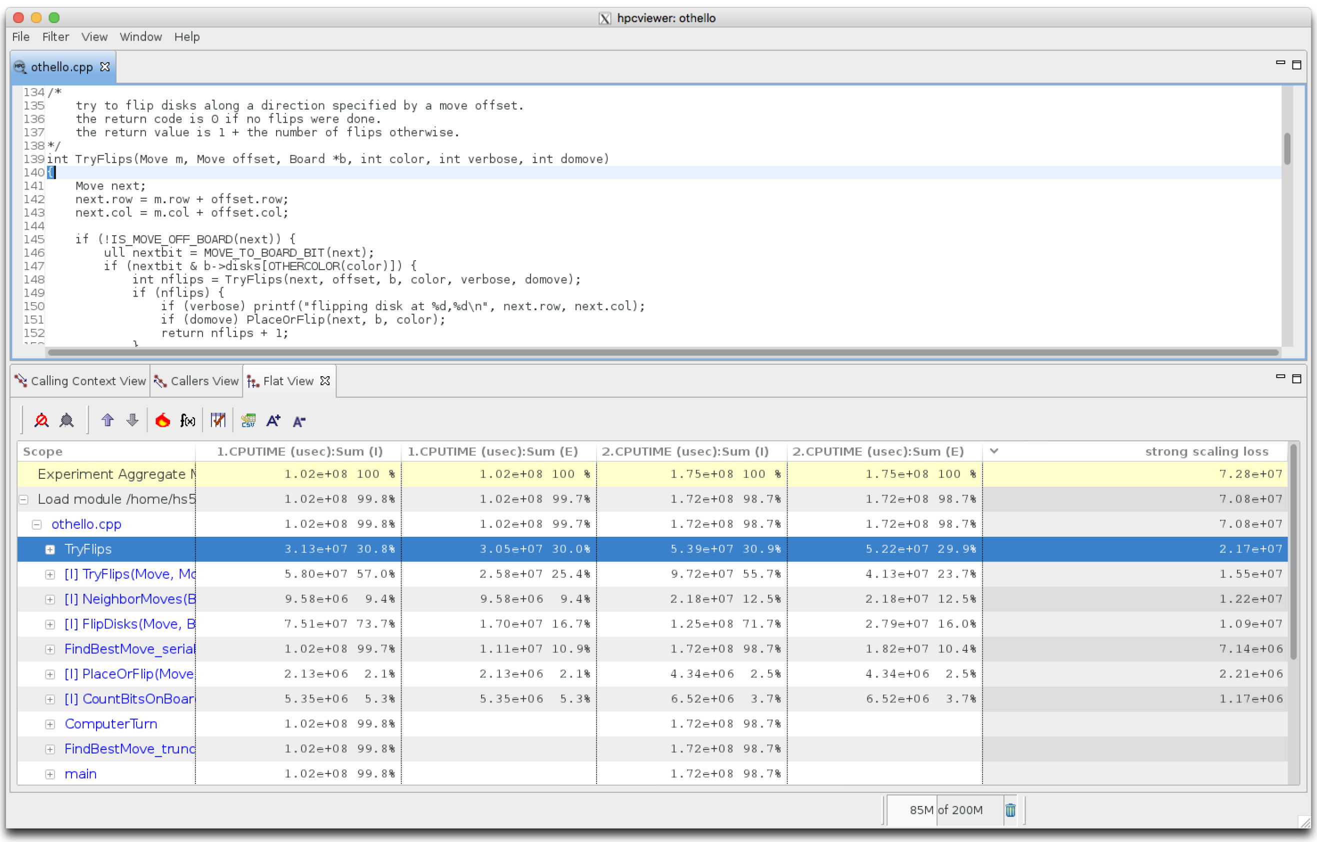
The real running time of serial version program is 1m40.419s. So the parallel efficiency figure is as following:





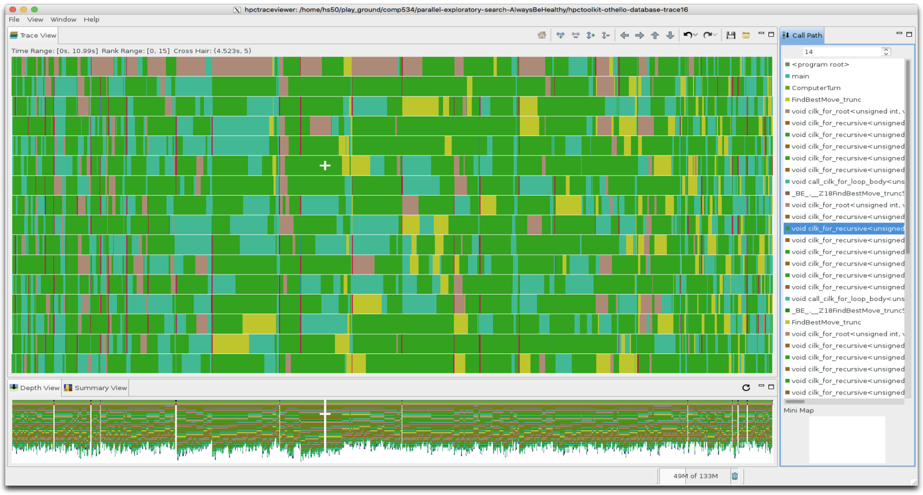
Since the efficiency begin to drop down quickly when the number of cores is larger than 8, I use hpcview and hpctraceview to see what happens there.

3. HpcToolkits Usage



I compared the performance under 4 cores and 16 cores. As we can see, the FindBestMove_serial and TryFlips are two main scaling loss. I do not make any change of TryFlips because the granularity will be too fine when this function is paralleled. Actually, I have tried to do so, but the heavy overhead slowed down my program. So I recovered the change.

As for FindBestMove_serial, it is related to the truncate level, so I tuned the parameter of truncate level to the best, which is 3. And I move to use Hpctraceviewer to see what else to do.

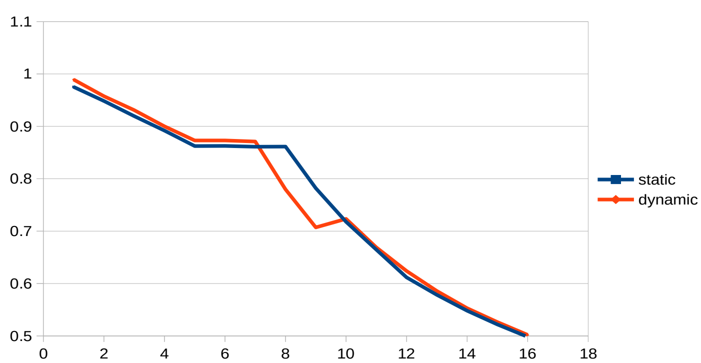


The figure above is the trace figure of program with 16 cores. We can see multiple vertical lines in the graph, which means some processors are waiting results from FindBestMove_trunc(). Besides waiting time is longer in middle than the beginning and the end of the program. By look into the call stack, I think one reason must be that at the middle possible legal moves are more than legal moves at the beginning and the end. As a result, with the same truncate parameter, the job of FindBestMove_serial is also increasing in the middle. So the waiting time is longer.

Improvement by dynamic truncating

Because the jobs of FindBestMove_serial become more when it is in the middle of the program, it maybe become worthy to make FindBestMove_serial parallel in the middle of the program. One easiest way to do this is to low down the truncating parallel to decrease FindBestMove_serial's jobs. I used a global variable total_bit_count to record the total bits on the board, the truncate parameter will be decreased to 2 when 33 < total_bit_count < 42.

The comparison of static truncating and dynamic truncating is as below:



The dynamic truncating worked, but only little progress is made. I think I can get a better performance after tuning the parameters. (Sorry for time is up).