

Othello

Parallel Programming - Assignment 1

Program Overview:

The main function takes in the required parameters to play the game i.e. the type of player and the lookahead depth if the player is a computer. It evaluates it and then performs sequential turns. The game stops when there are no possible moves for either player. If the player is a human, then we call the **HumanTurn** function, else we call **ComputerTurn**. Before getting into the details let us define a few terms:

Legal Move: A legal move is the one which has a disk of the same color in a given direction and all positions in between occupied by the opponent.

Player: A player can be a computer or a human. If it is a computer it has to have a lookahead depth to compute the best move possible during its turn. Each player has its color, either O or X and both players play consecutively unless there isn't a legal move available for the player.

The HumanTurn function:

It takes inputs from the user for a Move, evaluates if it is legal and flips the game disks accordingly. If the move is not possible, it will print a message and ask the user to enter the correct value again.

The ComputerTurn function

This function instead calls the **findBestMove** function to look up the best move possible with a given "depth" for the computer player. This is where the major crux of the code lies. If there was no move obtained from the findBestMove function, then we just skip the turn and print the statement with the appropriate result.

Below is a pseudo code of both the functions:

It takes the parameters:

1. gameboard - current Game Board configuration
2. color - the color of the current player (0 for O, 1 for X)
3. depth - the depth of the current search iteration
4. search_depth - the lookup depth of the player as given by the user
5. mul - the factor by which the result has to be multiplied with. Is -1 or 1 for negamax algorithm
6. is_parent_skipped - check if the previous iteration was skipped because there was no legal move left. If it is true and we currently have no legal move left then we should stop searching
7. best_move - a pointer to store the best move for the given turn. This is substituted for the best move after the search is completed.

```

void ComputerTurn(Board gameboard, Player player):
    /* just initialize best_move with a move that is not possible */
    Move no_move = {-1,-1};
    Move best_move = no_move;
    findBestMove(gameboard, player.color, 1, player.depth, 1, true, best_move);

    if(best_move == no_move):
        SkipTurn();
    else:
        /* Flip Disks with the best_move obtained and print flipped */
        FlipDisks(best_move, gameboard, color, depth);
        PlaceOrFlip(legal_move, gameboard, color);
        PrintBoard(gameboard);

int findBestMove(Board gameBoard, int color, int depth, int search_depth, int mul, bool
is_parent_skipped, Move &best_move):

    /* find all legal_moves */
    Board legal_moves = EnumerateLegalMoves(gameBoard, color, &legal_moves);

    int max_diff;

    /* for every move find the difference and select the best diff */
    for(legal_move : legal_moves):
        Board boardAfterMove = gameboard;
        PlaceOrFlip(legal_move, boardAfterMove, color);
        if(search_depth == depth) max_diff = max(max_diff, findDifference(boardAfterMove,
color));
        else max_diff = max(max_diff, findBestMove(boardAfterMove, OTHERCOLOR(color),
depth+1, search_depth, -1, false, best_move));

    /* skip turn for this player else return if there are no moves left for both players */
    if(num(legal_moves) == 0):
        if(is_parent_skipped): max_diff = findDifference(b,color);
        else max_diff = findBestMove(b, OTHERCOLOR(color), depth, search_depth, -1, true,
best_move));

    /* store the corresponding best move if depth==1 */
    if(depth==1) best_move = MoveWithBestDifference(max_diff);
    return best_diff * mul;

```

Pseudo code for the ComputerTurn and findBestMove functions

- It first checks the possible legal moves in the gameboard. If there are no searches possible at the start of the search tree then the computer player should skip a turn.
 - It then searches for the best move by playing the move and searching for **depth+1** with the opponent's color.
 - If depth == given search depth , then we just find the difference of the disks between the gameboard.
 - If a legal move was not possible then we just recursively call the findBestMove with the opponent's color and same search depth.
 - The algorithm used is negamax. So the best move is the one which maximizes the product (-1* findBestMove(opponent's color, depth-1, boardAfterMove))
 - For depth =1, we have to also store the best move, so that we can print the move and flip the disks of the original game board.
-

Serial Vs Parallel Code

The main difference between the serial and parallel code is the implementation of the findBestMove function where a **cilk_for** is used to find which is the best move from a set of legal moves for the given board configuration, color and depth.

```
/* for every move find the difference and select the best diff */
for(legal_move : legal_moves):
    .....
    .....
```

```
/* for every move find the difference and select the best diff */
cilk_for(legal_move : legal_moves):
    .....
    .....
```

The parallel version of the code spawns a thread for every legal move, searches it recursively till it reaches the end of the search tree. Returns the result back to the parent. An implicit **cilk_sync** is present at the end of the for loop implying every spawned thread waits for all legal moves to get computed, combines the result and returns the best move to its parent procedure.

Data Races and Reducers:

The cilk code was checked for data races using the cilkscreen command provided by the cilk runtime and the output of which is present at the bottom of '**cilkscreen.out**' file. \

To prevent the data races in the parallel section, two reducers were used:

- `cilk::reducer_max<int> best_diff`
 - This reducer was used to store the best difference possible from choosing a legal move in a given iteration. Every legal move is searched and the difference between the disks of the player and opponent is stored inside the reducer to prevent data races on the object.
 - The value of the reducer is then passed to the parent after the `cilk_for` ends and an implicit `cilk_sync` finishes so that all parallel executions have finished pushing value to the reducer.
- `cilk::reducer_max<pair<Move, int>, MoveComparison> best_move_reducer`
 - This reducer has a special use only for the case `depth == 1`. This is because we also need to store the best move so that we can print it and also apply this move to the original game board.
 - This reducer takes the pair of a move which in itself is just a pair of two ints: **row** and **col** and also the difference produced by this move.
 - The reducer has a custom comparator because we are not dealing with native C++ data-types.
 - The custom comparator finds the move with the best difference. In case, the difference is the same, we consider the move with lesser row and column configuration.

Why do we need the custom comparator to sort moves with the same bit difference based on row and column?

Answer: To enforce determinism

Each of the cilk spawned threads computes independent of other threads and we have no control of their execution. This means that any thread can push its value to the reducer at any time. If we do not sort the moves based on their row and columns, we might get different moves with the same difference in every execution. This can lead to an entirely different execution path and the parallelism can vary drastically.

Observations:

The entire board is searched for every legal move possible.

At max the possible legal moves can be upto 32. This is because every legal move has to be a neighbour of an already occupied disk. So possible legal moves \leq occupied disks in the gameboard. Possible legal moves + occupied disks \leq 64. In practice however the number of possible moves are often less because of the game restrictions and definition of a legal move.

I found that on an average, there are only 10.5 possible legal moves for a given board configuration and current turn. This means that we can at max only achieve a parallelism of about 10 using our implementation. However, there are other factors to consider like not every legal move can lead to a similar search path. For cases, when the number of legal moves possible are less than the threads available, we will have a greater overhead because of lesser work available and threads sitting idle waiting for the next steal or continuously stealing small chunks.

Observations from Cilkview:

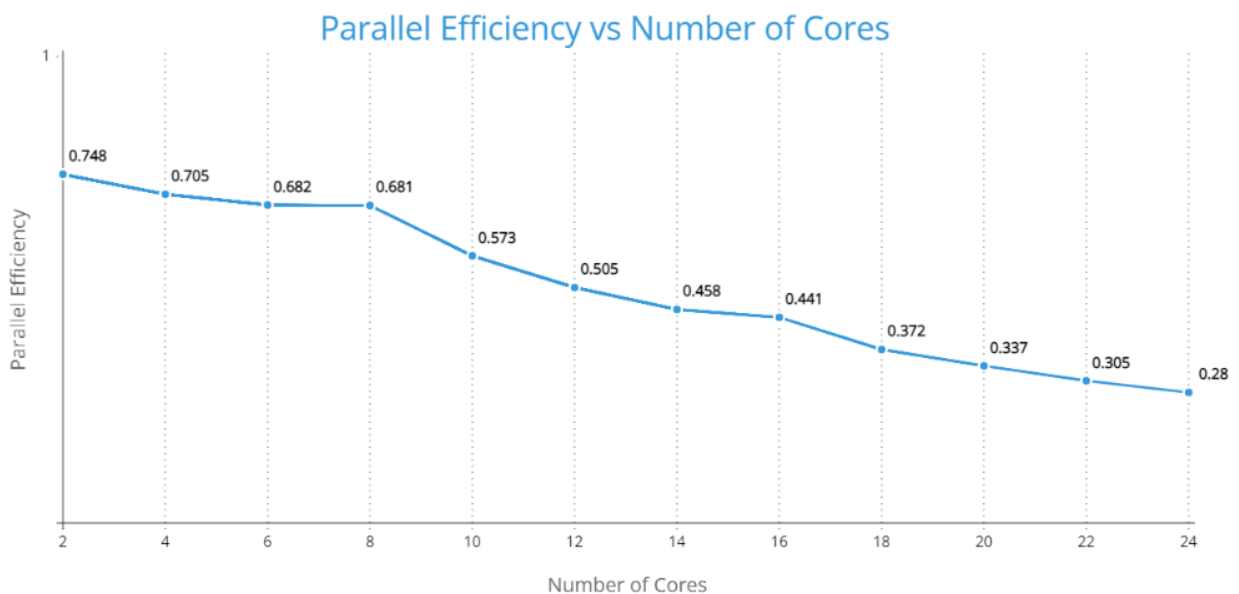
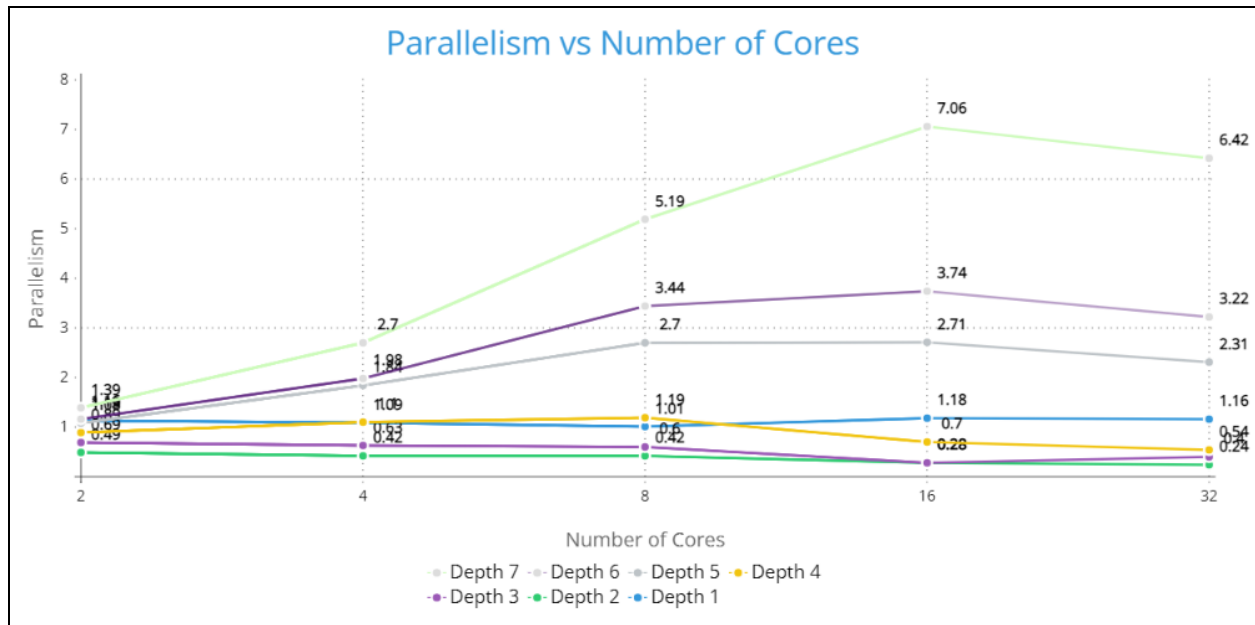
The table below shows the different metrics by cilkview for different search depths.

| Depth | Work | Span | Burdened Span | Parallelism | Burdened parallelism | Spawns/Syncs |
|-------|-----------------|------------|---------------|-------------|----------------------|--------------|
| 1 | 10,204,801 | 9,823,895 | 14,566,479 | 1.04 | 0.7 | 425 |
| 2 | 17,197,624 | 10,394,434 | 20,848,626 | 1.65 | 0.82 | 3917 |
| 3 | 70,060,867 | 11,272,397 | 27,816,824 | 6.22 | 2.52 | 26,083 |
| 4 | 285,954,780 | 11,798,924 | 33,311,797 | 24.24 | 8.58 | 121,793 |
| 5 | 3,971,696,090 | 12,603,582 | 41,021,255 | 315.12 | 96.82 | 1,803,351 |
| 6 | 11,237,995,841 | 13,434,714 | 45,125,968 | 836.49 | 249.04 | 4,942,786 |
| 7 | 872,558,761,606 | 14,220,965 | 56,084,696 | 61357.21 | 15557.88 | 433,749,541 |

- For search depth 1 & 2, we see that the span is almost equivalent to the work, this is because there aren't a lot of possible moves and hence not a lot of branching from the main thread. This also is evident from the amount of parallelism in the workflow which is just about 1
- For a search depth of 3 & 4, we see that burdened parallelism is pretty scanty as compared to parallelism. This implies that the parallel overhead from the cilk spawns and sync is outweighing the benefit obtained from parallelism. In fact, the performance is poor than the sequential code for depth = 3
- For a search depth of 5-7 we see some benefit obtained from the parallelism. Also observed that the work is pretty high as compared to the span which means that there are a lot of different cases and corresponding paths to consider. This is also evident from the spawns and syncs. In fact, if we consider that on an average there can be 10 legal moves available for any configuration, then every subsequent search multiplies the available paths by a factor of almost 10.
- The ratio between the parallelism and burdened parallelism is pretty interesting. It first is almost the same (depth == 1) when there is almost no parallelism available. Then it slightly increases upto a point where the parallel overheads almost negates the benefit obtained from it (depth == 2 to depth == 5). And it finally starts to decrease after the benefit starts increasing (depth == 6 to depth == 7).

Observations from Parallelism vs Number of Cores:

- For depth 1-5, the predicted burdened parallelism expected by the cilk system is the same as that by obtained by the code
- For depth 6-7, the predicted burdened parallelism is higher than the obtained. This is because the cilk runtime expects an infinite number of cores while computing the parallelism.
- For all depths, the general trend is that parallelism increases almost linearly with the number of cores upto a certain value. However, it stays the same later or varies slightly. This is because the highest parallelism by these search depths cannot go higher than a given value. As per Amdahl's law, the parallelism cannot be achieved over the sequential work done.
- Noticed that parallelism doesn't increase much after cores=16. This is because of the architecture of the hosts obtained from SLURM over which the code is run. At most we have only 16 cores available with 16 hyperthreads. Since Cilk runtime considers each thread as a core, it shows the number of cores as 32. Whereas in reality the number of cores available are only 16 and the hyperthreads interleave with other threads and hence we do not see much difference in parallelism.
- For depths 6-7, the overall speedup should have been higher than obtained. However, the overheads of spawns, syncs and stealing doesn't let the overall parallelism go beyond a certain threshold.



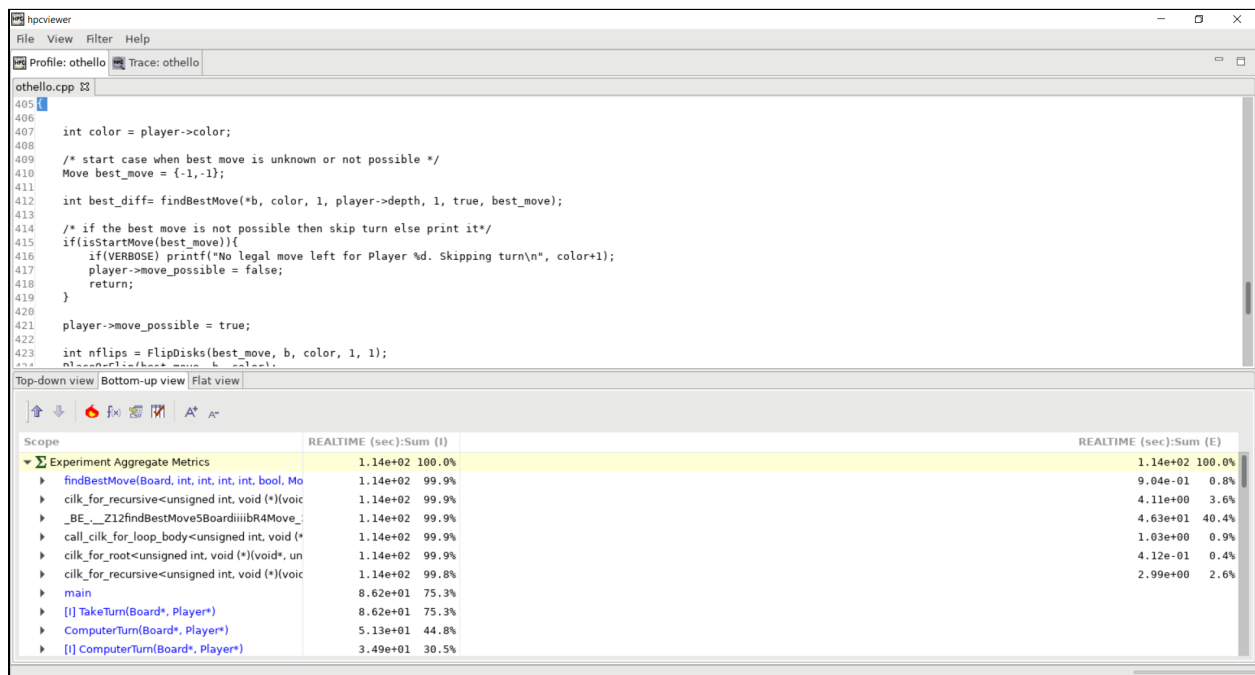
Parallel Efficiency vs Number of Cores plotted for search depth = 7

Observations from Parallel Efficiency vs Number of Cores:

- The parallel Efficiency in general decreases with the number of cores. This is because of the overheads obtained with adding more cores. The spawning and syncing along with the work stealing contributes some overhead with doesn't allow the process to scale exactly linearly with the cores

- The parallel Efficiency stays consistent upto 8 cores however, it takes a sharp dip afterwards indicating that the parallelism grows linearly with number of cores upto 8 and then does not change much.
- As previously stated the number of legal moves for each configuration on an average stays about 10. So the parallelism is expected to be less than 10 at max and that is what we see with this graph.

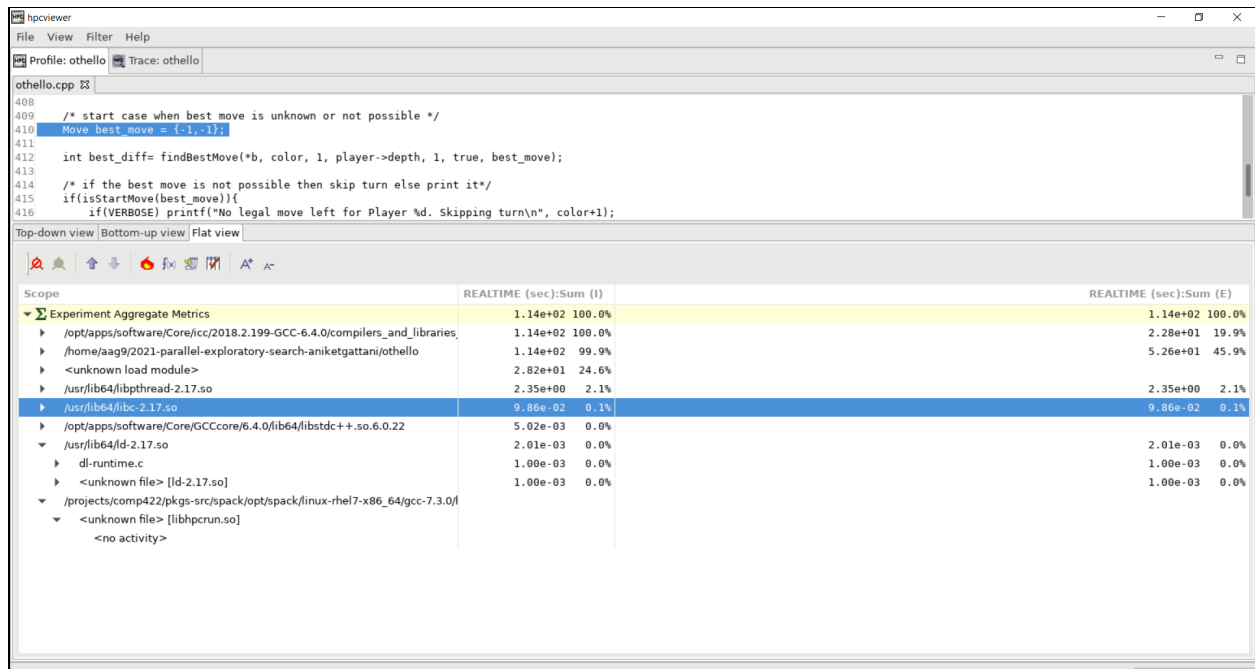
Observations from HPCToolkit:



The screenshot shows the hpcviewer application with the source code of othello.cpp open. Below the code, the 'Top-down view' tab is selected, displaying a performance metrics table. The table has two columns for 'REALTIME (sec):Sum (I)' and 'REALTIME (sec):Sum (E)'. The data is as follows:

| Scope | REALTIME (sec):Sum (I) | REALTIME (sec):Sum (E) |
|---|------------------------|------------------------|
| Experiment Aggregate Metrics | 1.14e+02 100.0% | 1.14e+02 100.0% |
| findBestMove(Board, int, int, int, bool, Mo | 1.14e+02 99.9% | 9.04e-01 0.8% |
| cilk_for_recursive<unsigned int, void (*)(&voic | 1.14e+02 99.9% | 4.11e+00 3.6% |
| _BE_...Z12findBestMove5BoardiiiiB4Move_ | 1.14e+02 99.9% | 4.63e+01 40.4% |
| call_cilk_for_loop_body<unsigned int, void (*) | 1.14e+02 99.9% | 1.03e+00 0.9% |
| cilk_for_root<unsigned int, void (*)(&voic*, un | 1.14e+02 99.9% | 4.12e-01 0.4% |
| cilk_for_recursive<unsigned int, void (*)(&voic | 1.14e+02 99.8% | 2.99e+00 2.6% |
| main | 8.62e+01 75.3% | |
| [I] TakeTurn(Board*, Player*) | 8.62e+01 75.3% | |
| ComputerTurn(Board*, Player*) | 5.13e+01 44.8% | |
| [I] ComputerTurn(Board*, Player*) | 3.49e+01 30.5% | |

HPCToolkit was run with cilk_workers=8 and depth = 7 (both computers)



Flat view of the HPCToolkit

- Bottom-up view:
 - Almost 99.9% time is spent in findBestMove which is of course expected because bulk of the work is done in this method.
 - Next, main, TakeTurn and ComputerTurn methods take the most time which is expected because they are the parents to the findBestMove method.
 - The next most time consuming method was EnumerateLegalMoves which tries to find legal moves for the given board configuration. This is expected because every findBestMove method calls this method once. Furthermore, every iteration of this method first finds all neighbors, then checks for the legal moves looping over all of the board. Finding neighbors takes constant time since we use the 8 offsets and perform bitwise operations on each.
 - Other procedures like FlipDisks and TryFlips are dependencies of EnumerateLegalMoves hence they are expected to take time.
 - Since we are using two reducers, the overall compute time in reducers is about 0.6%.
- Flat view:
 - The flat view shows that about 99.9% time is spent in the program and the remaining is spent in C binaries and cilk runtime system which is not harmful.
- Trace:
 - From the trace graph, 0.18% time is spent for no-activity which isn't harmful considering the execution time and number of instructions in the code
 - The call stack shows deeply nested calls for findBestMove and the cilk implementation of the call. This constitutes almost 99.8 % at call stack depth=4;

Other Approaches:

- Extract more parallelism:
 - The natural question is why not extract more parallelism. EnumerateLegalMoves can be used to extract parallelism since it consumes almost 27% runtime.
 - One possible solution is to spawn a new thread every row, check for a possible neighbor, check if it is a legal move and finally add it to the legal moves set.
 - However, EnumerateLegalMoves suffers from the issue that the neighbor moves are not localized and can be anywhere in the board configuration. Hence, one row might be completely empty with no neighbors possible any another might have all positions possible and hence we might need to check for all such positions.
 - Hence, some threads might perform fast while other threads may have large computations. This would not ideally upscale the parallelism and in fact can cost more overheads.
 - Another possible solution is to calculate neighbors and then try to find legal moves in the neighbors set. While this might work, the task associated with every spawn is so little that it costs more overheads for every sync.
 - Another possible
- Reduce overhead by truncation:
 - One possible solution is to just reduce the search depth tree's spawn by pruning it and evaluating the last few depths by sequential code.
 - While this seems to reduce work stealing a bit, it still does not solve the problem by much. This is because there is sufficient slackness available here (verified using cilkview profile where available parallelism is huge). So the problem cannot be solved using pruning.
- In short, the problem here is that there are a lot of atomic operations for which the code needs to wait for synchronization. Spawning threads do reduce the serial work but they have to be synchronized and the result has to be passed to the parent in order to evaluate the next result.