Parallel and Distributed Othello Engine

Willie Cohen
Johnny Jacobs

CSC 458 Spring 2017
Professor Michael Scott

Serial Implementation

We started with a serial implementation of an Othello engine that we created in CSC 242. The original code is C and assumes an 8x8 board. Also included is a Java jar that can run programs against each other in a gui or text environment. This was useful for testing the parallel versions, but we decided to test the mpi versions by hand instead of modifying the Java source for the tournament jar.

```
int AlphaBeta(Node cur_node, int alpha,
              int beta)
{
  if(cur_node.is_leaf)
    return Evaluate(cur_node);

  while(HasMoreSuccessors(cur_node)) {
    succ_node = GetNextSucc(cur_node);
    score = -AlphaBeta(succ_node, -beta,
                       -alpha);
    if(score >= beta)
      return beta;
    if(score > alpha)
      alpha = score;
  }
  return alpha;
}
```

Figure 1: Our serial version used the alpha-beta negamax algorithm. Taken from [2].

Bitboarding:

Our original implementation also included bitboarding. For bitboarding in Othello, 128 bits are used to represent a state, 64 for black positions, and 64 for white. As such, each color fits perfectly into a long long. Moves are pre-calculated for every unique pairing of rows based on color to move and stored in a table. For example `moveTable[whiteRow][blackRow][color]` gives the moves for a row that has white pieces as specified by `whiteRow`, black pieces as specified by `blackRow`, and it is `color`'s turn. This only gets row moves, to get column moves, the board is dynamically rotated 90 degrees. To get diagonals, the board is dynamically rotated left and right 45 degrees, respectively and a row-specific mask is applied. These masks are also pre-calculated. A large amount of time is spent rotating the board, so pre-calculating the rotation of a row could

provide a pretty big speedup to this code, however we chose to focus on the parallelization aspect as it was the main target of this endeavor.

Heuristic:

The original heuristic was based off three factors, material, mobility, and position. Material was simply which color had more pieces on the board. Mobility was the number of moves a position allowed, and position was positive for capturing corner tiles, and negative for placing a piece adjacent to a corner tile. Each of these four had a weight associated with it (two independent weights for position), and the heuristic was simply the weighted sum.

Bugs:

Our original alpha-beta algorithm was turned out to be deeply flawed. Because of a swapped alpha and beta, it ended up not pruning at all, and performing only a naive minimax. This was corrected in both the serial and parallel versions when we discovered it, and made the algorithm overall 2-4x faster.

The heuristic did not take into account whether a state was a victory or defeat state. Thus, a state where one was only behind slightly in the categories could be rated as 'ok', but in reality be a defeat state. This was fixed; victory states were given the highest value possible, and defeat states the lowest possible. Tie states were treated as defeat states for simplicity, though this could be improved upon.

Bitboarding was robust and needed no changes for the whole project. This was both a relief, as bitboarding was by far the most complex, convoluted, and unintuitive part of the code, as well as gratifying, as it showed a level of competency I thought beyond my sophomore self.

Initial Steps:

We started by profiling the serial version of our code using `gprof`, to inform us on what sorts of algorithms would be viable, and to make sure we directed our efforts the right locations. Most time was spent rotating the board (77%), and the rest generating children of the current state (20%).

Thus, as discussed above, a huge speedup would be achieved by implementing a rotate lookup-table. Rotation of the board couldn't be directly parallelized as each call took negligible time, but 50 million calls were made per move; parallelization would just add overhead to each of these calls. Parallelizing the whole alpha-beta algorithm looked viable as the rotation and generation happen inside a single recursive call stack frame, so those 50 million calls could be spread over threads.

Algorithms:

After this, we brainstormed possible splitting techniques. Our initial idea was to have each node send out its children as asynchronous processes, taking one child for itself. After some research, we decided to implement a parallel primary variation split (PVS) as well as a younger brothers wait concept (YBWC) in parallel. YBWC is essentially our initial idea, except the best child is executed sequentially first, and used as a base prune point for the other

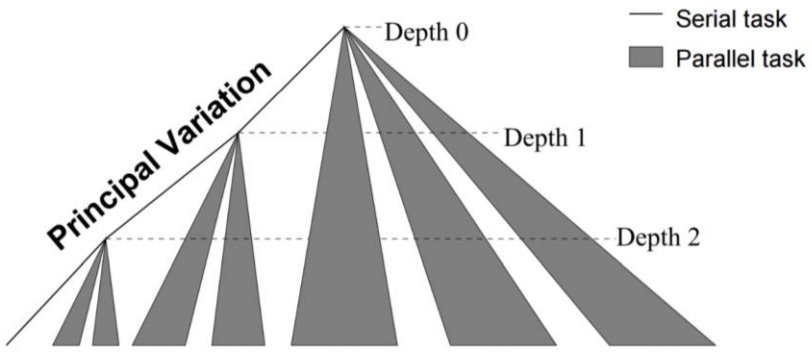


Figure 2: PVS execution path. Taken from [2].

children when they are called in parallel. PVS is similar, but only asynchronously splits off children for nodes in the leftmost branch of the tree (See Figure 2). Essentially YBWC is PVS where every node is treated as a new root for its subtree.

Implementation: Parallel

A threadpool seemed like great fit for these algorithms, as there were many tasks, all much shorter lived than the program so that the overhead of creating new threads for each new task would have a substantial effect on the parallel efficiency.

No good threadpool libraries were found in C, so the code was upgraded to C++. A standard implementation thread pool was nowhere to be found, so a third-party implementation (CTPL)[1] was found and used.

Primary Variation Split:

This implementation follows the PVS algorithm exactly, with a future and asynchronous call to serial versions of the alpha-beta pruning algorithm. If no idle threads are in the pool, the serial version is called directly. All children futures are waited upon before returning up to the next level. Once child-subtrees are less than a certain (adjustable) depth, they are executed serially. Futures were stored in a vector, with 10 reserved slots. This value was chosen as it is the average branching factor of Othello, and should therefore handle 50% of cases without a resize. We thought this was a reasonable tradeoff between size and speed, though through

preliminary testing the initial 'reserved' size seemed not to matter much, as values of 1 and 40 (max branching factor) performed identically to 10.

Younger Brother Wait Concept:

This implementation is the same as the PVS implementation, but the parallel alpha-beta pruning algorithm is called asynchronously by children, instead of the serial version. Again, once child-subtrees are less than a certain (adjustable) depth, they are executed serially. The size of the future vector, though it was allocated many times more in this version than in parallel PVC, still seemed to have little effect on performance.

Implementation: Distributed (MPI)

Primary Variation Split:

For the implementation of a distributed version of our Othello AI, we decided to start with implementing the PVS algorithm. We chose this starting point because we were already familiar with PVS from the parallel version, and we theorized that moving this simple algorithm to a distributed model would be somewhat straightforward. Using MPI we restarted from the serial version to accomplish our final implementation. To set up the algorithm, our main function launches the desired number of MPI processes at the start of the program. Then the root process queries the user for the desired game start information. With that, it broadcasts the color and depth limit to all of the other threads, so they all have those global constants. They all then complete the precomputation functions independently, with an MPI_Barrier after them to ensure everyone has completed the computations before proceeding. At this point, both the root process and the slave processes all call the make move function. Inside this function, the root process goes into a minimax call, while all the slave processes wait in an infinite true loop for a node assignment. Since we are implementing PVS, the master process will call minimax, diving to the depth limit of the tree. Then it will return upwards, having obtained guess alpha and beta pruning values from what looks to be the best child at each level. With this information, it calls on slave processes at each level to complete a run of minimax with these new values. At each level, it loops through the remaining children nodes, sending one per slave process, and looping back over the list of slaves if there are more children than processes. Because the slave processes are sitting in a while (true) loop waiting for parameters of the minimax algorithm, they will always receive these sends from the master. We only pass the board position, instead of a whole node struct, because for every level except the top, the other properties of node are not used. However, we do have to pass the other values for the parameters of minimax, but they are all ints and doubles. The slave processes will use these parameters to complete a serial run of minimax, and then send the value back to

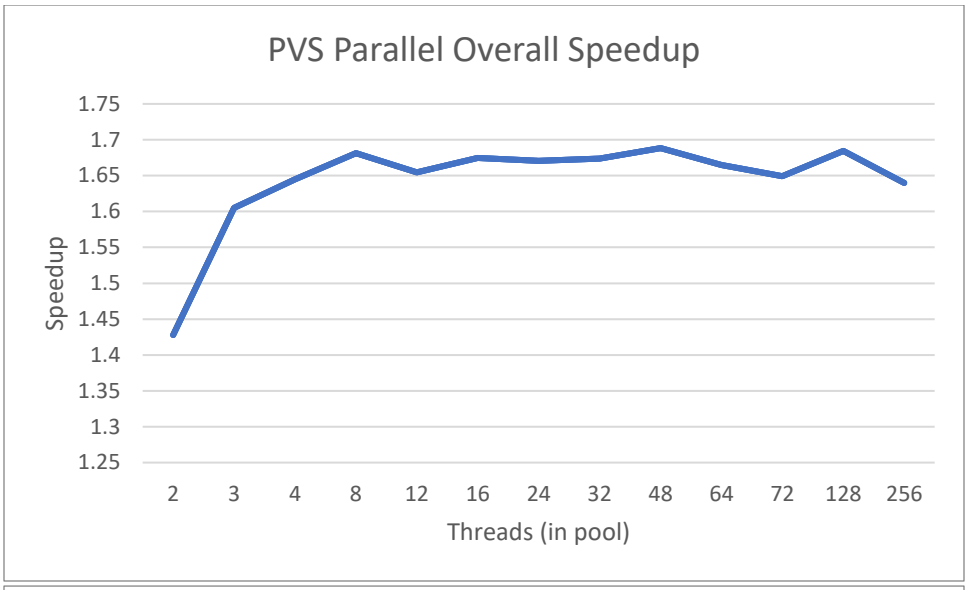
the root process. The root process waits to receive all of the children results on one level, and then chooses the best and returns to the level above to repeat the process.

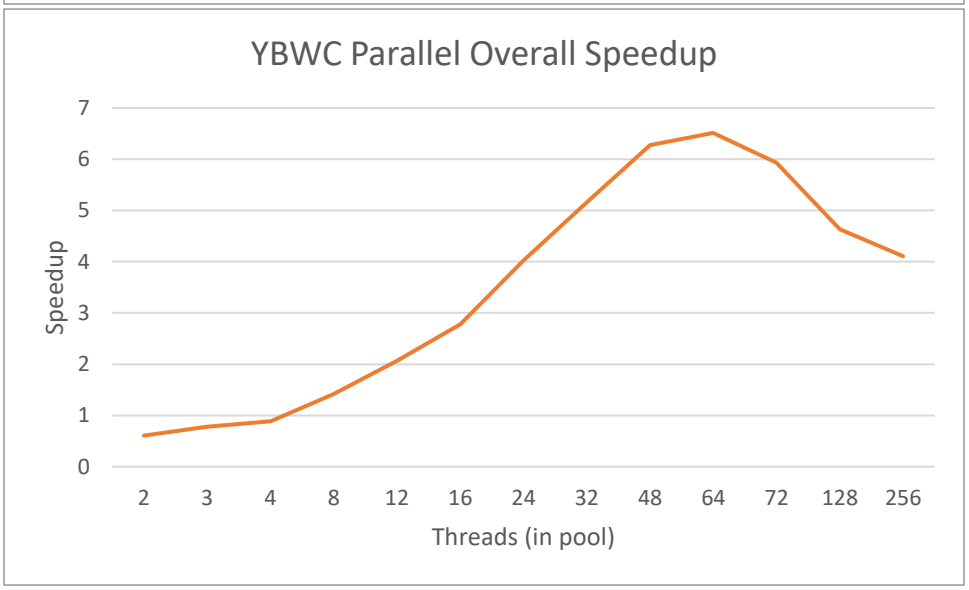
Custom Split:

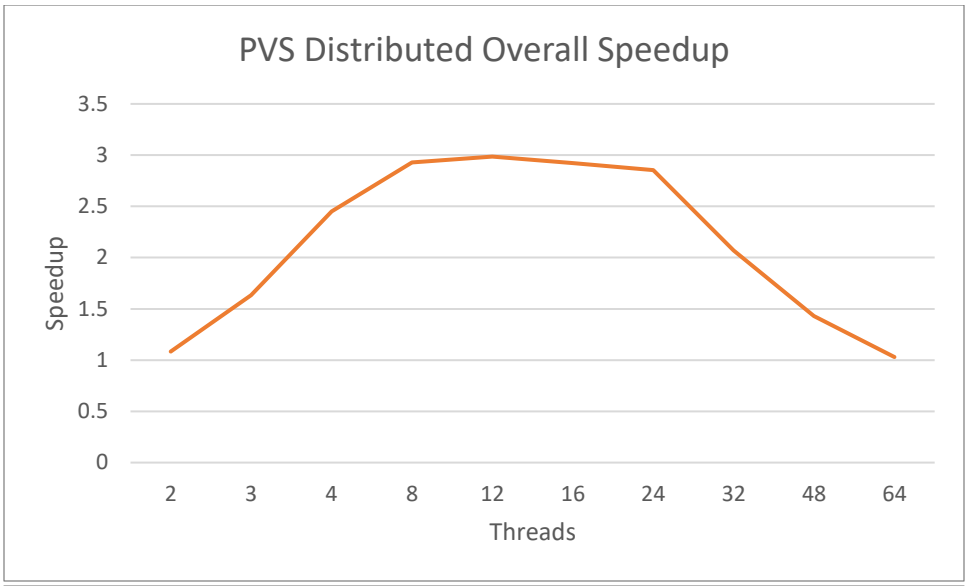
After completing the PVS algorithm in MPI, we decided to try a different implementation of a simple algorithm that would minimize the message passing on the system. We concluded that the best way to do this was to simply split the tree into sections, and then have processes complete independent serial minimax runs. Then the master could simply choose the best from return values of the children. To accomplish this, we used the same initial setup as the PVS version, but we altered the structure of our minimax functions. We returned minimax to its original serial version, and created a second master_minimax function. The slave nodes still loop indefinitely waiting for parameters to feed into the serial minimax algorithm, but now the master calls master_minimax. Inside this function, the master acts as an overseer, simply splitting up the tree amongst the slave nodes. We accomplish this by iterative deepening. The master will expand the root of the tree (initial board state), and put the nodes into a queue. If the size of the queue is greater than the amount of processes, it will distribute them amongst the processes. If the queue has less nodes than the number of processes, it will go back and expand all the nodes into children nodes, and check to see if there are enough again. We did this in order to effectively scale the workload among the changing number of processes. After the master has a satisfactory number of nodes distributed, it will wait for the slaves to return values, and then chooses the best to make the move.

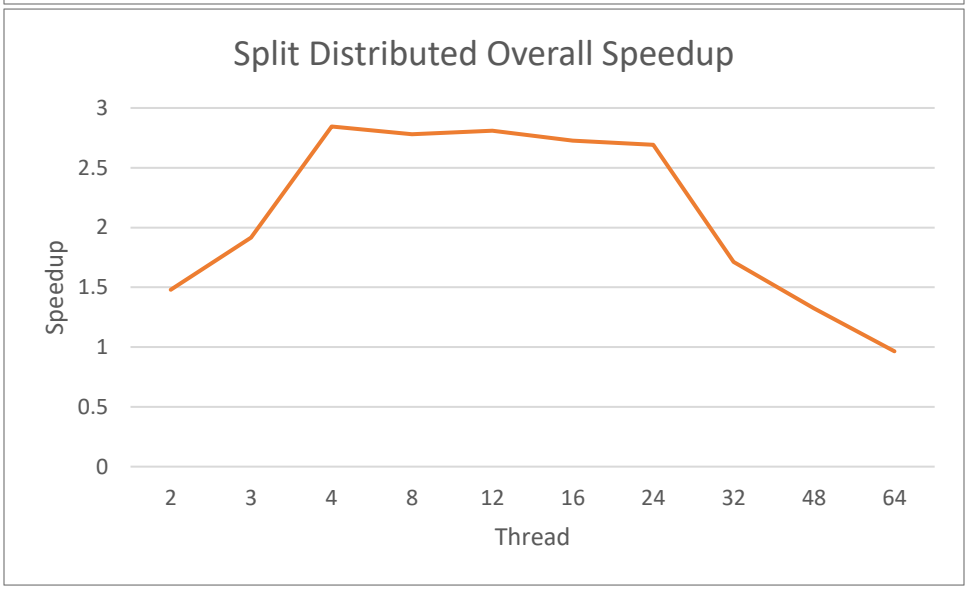
Results

Testing of the parallel code was done on node2x14a, which has 56 hardware threads. Distributed code was tested on the node setup from Assignment 4, using all 6 nodes. This represents 48 hardware threads, 8 per node. Timing was averaged over three runs per data point. Each data point was the sum of the time taken to search to a certain tree depth for a single move from three unique board positions. These board positions were taken from the beginning, middle, and late-stage of a game, respectively. The graphs below are the speedup (sequential_time/parallel_time) for each algorithm. The speedup graphs for each board position individually, can be found in the Appendix.


PVS Parallel Overall Speedup
Speedup
1.75
1.7
1.65
1.6
1.55
1.5
1.45
1.4
1.35
1.3
1.25
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)


YBWC Parallel Overall Speedup
Speedup
7
6
5
4
3
2
1
0
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)





Analysis

Parallel Primary Variation Split:

For parallel PVS, initial speedup was good, but plateaued quickly. This is a problem inherent in the algorithm as it can only utilize a number of threads equal to the number of children at the current node ('current' being the node the main thread is at). Because of this, the algorithm is thought to have an asymptotic speedup of 5 [3]. Coupled with overhead, leveling out at such a low thread count is not unexpected. In fact, up to 8 threads our speedup is fairly similar to that achieved by [2], also using PVS.

There is no drop off in the graph as would be expected as hardware threads are surpassed since the number of threads is simply the threads initialized in the pool. While this is

an initial overhead cost, there is virtually no penalty to having idle, unused threads in the pool, as they are not even woken until an item is placed in the queue.

Parallel Younger Brothers Wait Concept:

In parallel YBWC, the speedup was slower to start off with, but scaled amazingly well. The slow start is likely due to the overhead of splitting threads off at every node, but this splitting also provides almost limitless parallelism. (Theoretically, up to the number of tree nodes, but in our implementation, only the number of leaf nodes, due to the issue discussed below.)

As one can see from the graph, the speedup continues to rise past the number of physical hardware threads. This is explained by how the algorithm generated children. When a parent thread generated children and sent them out asynchronously, it would not take a child for itself unless no idle threads were left in the pool. This means that one thread is wasted blocking at every junction of the tree until no idle threads are left, at which point serial execution is turned to. Forcing a parent to take the last child for itself would likely improve on this, and make the peak of the speedup curve in line with number of hardware threads. Unfortunately, this was not realized until the analysis phase of the project, and such an implementation change was not time-viable. Note that this change also occurs with the parallel PVS implementation, but as there are only splits equal to tree depth, instead of equal to tree nodes, it poses much less of a problem.

Unlike parallel PVS, parallel YBWC does eventually drop off. This is because it can actually have every task in the pool working, even if that work is the blocking described above. Many extra threads are simply used in this 'node holding' capacity, but eventually the number of concurrently active, non-blocking threads exceeds the number of hardware threads, and a drop-off is observed.

Distributed Primary Variation Split:

From the graph, we can see results very similar to the parallel version of PVS. The reason the speedup plateaus quickly is again because the algorithm can only utilize parallel computation for a maximum of the available children at each level. If there are many processes, the ones beyond the number of children will not be used, because the algorithm will distribute a minimum of one child per process. We do see a greater maximum speedup (3 as compared of the parallel versions 1.7). This is most likely due to the fact that as the number of parallel processes gets larger, there is more contention on the cached values in the parallel version. The MPI version has to pass values a lot slower (accounting for the slower start), but does not have to deal with any cache contention, because the processes run independently with passed values. The drop off in speedup at the end, which differs from the parallel version is most likely explained by the fact that once we exceed the number of hardware threads (48)

the processes will have to share threads, and this will cause waiting that will deliver a serious blow to speedup.

Custom Split:

Despite our efforts to create an algorithm that would better mimic the results of YBWC parallel, we ended up with an algorithm with similar pitfalls as the PVS distributed version. The we get better performance initially from the split version due to the fact that the master isn't blocking progress waiting for messages to be passed back and forth at each level. However, it plateaus just like the PVS version. Our efforts to make it scalable beyond the number of children, by allowing the master to continue to break up the tree failed to improve performance with larger process counts. Even though we have more processes running in parallel, we did not see an increase in speed, most likely due to the changed pruning structure, and continuing persistence of load imbalance.

Conclusions

It might have been a better idea to start with code that was thoroughly debugged before trying to parallelize, but it made the project more interesting and exciting to revisit and improve upon code we had previously written. The guarantee that we would not have to learn any new concepts (at worst we'd have to relearn one) to understand the base code was a nice bonus.

The C++ threadpool implementation we used, in the parallel version, left something to be desired. As discussed in the analysis section, the threadpool used was somewhat wasteful of the threads it was given. Ideally, we would've had a `ForkJoinPool` as implemented in the Java concurrent library. This would've allowed a much more efficient use of threads.

The difficulties we encountered with MPI was that the same algorithms that work in parallel do not work in a distributed system. Transposition tables allow you to have different processes working at different depths share results with each other. Without a transposition table, it's not possible to implement the same kind of recursive parallelism that makes the YBWC method so successful. Since we did not have a transposition table, we were forced to stick with PVS, and try to experiment to find a different way of gaining speed. Another shortfall of MPI is the fact that dynamic process allocation does not really exist like it does with something like a parallel thread pool. You can mock dynamic allocation, but you cannot accomplish this in a system with recursion.

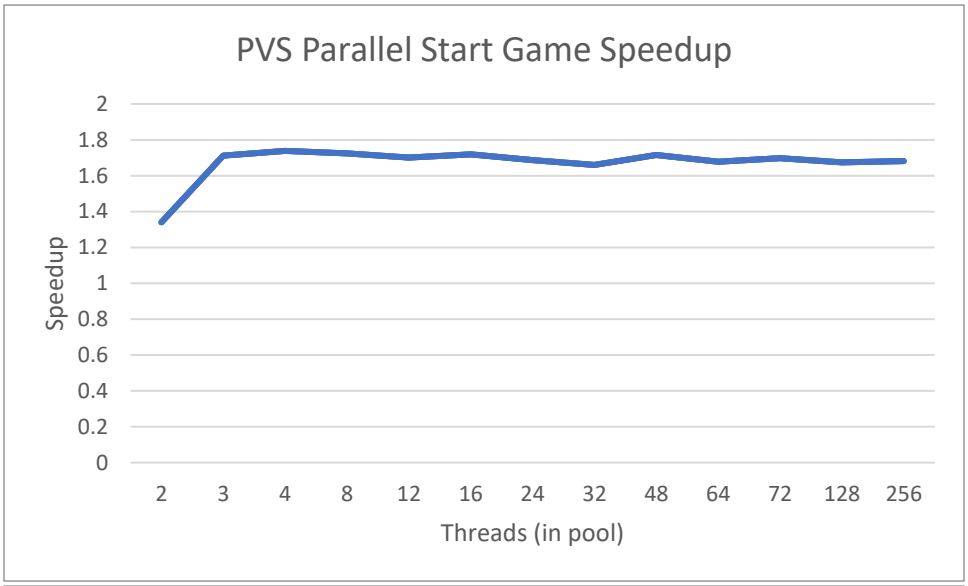
References

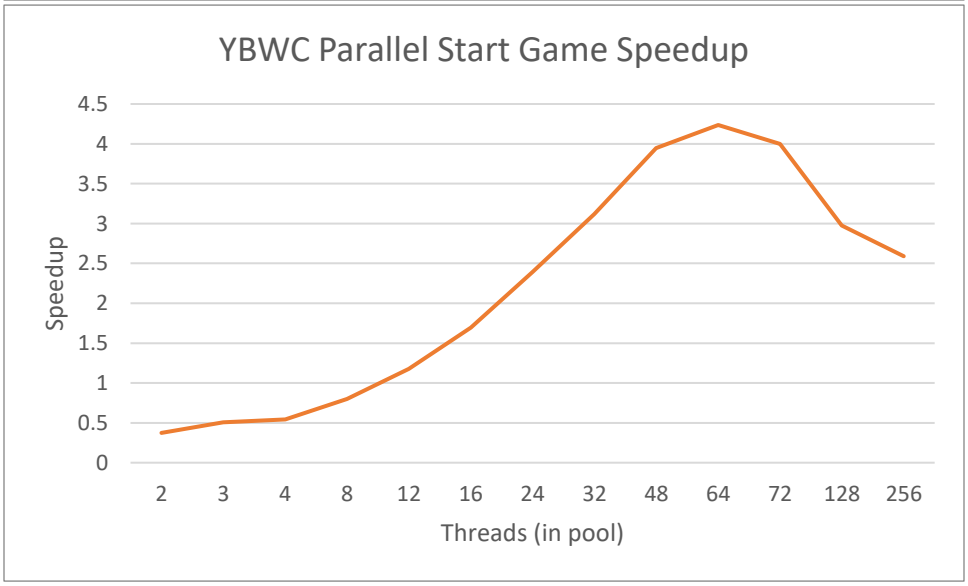
[1]: CTPL: Modern and efficient C++ Thread Pool Library. https://github.com/vit-vit/CTPL/

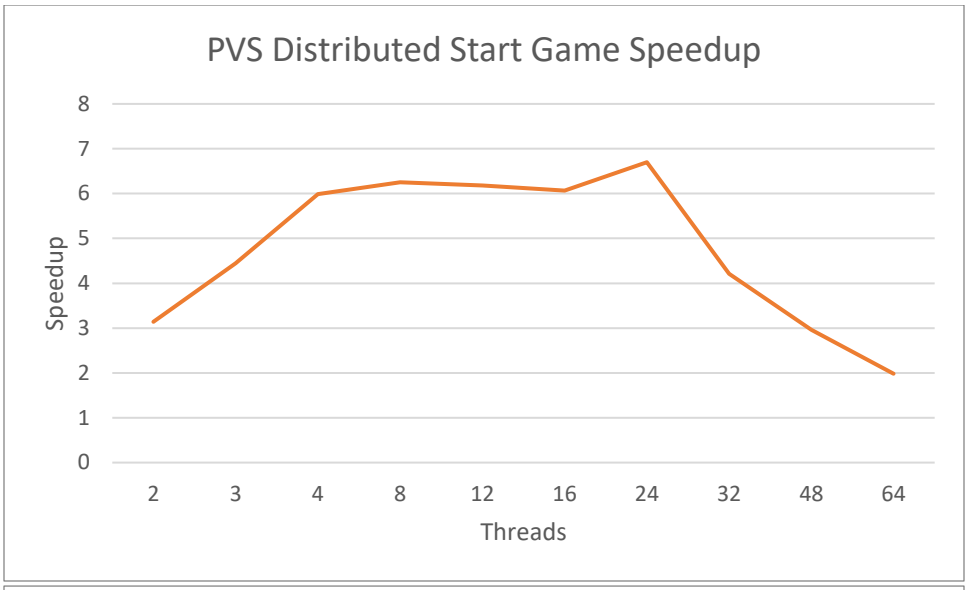
[2]: B. Greskamp. Parallelizing a Simple Chess Program. University of Illinois Urbana-Champaign, Fall 2003. http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf

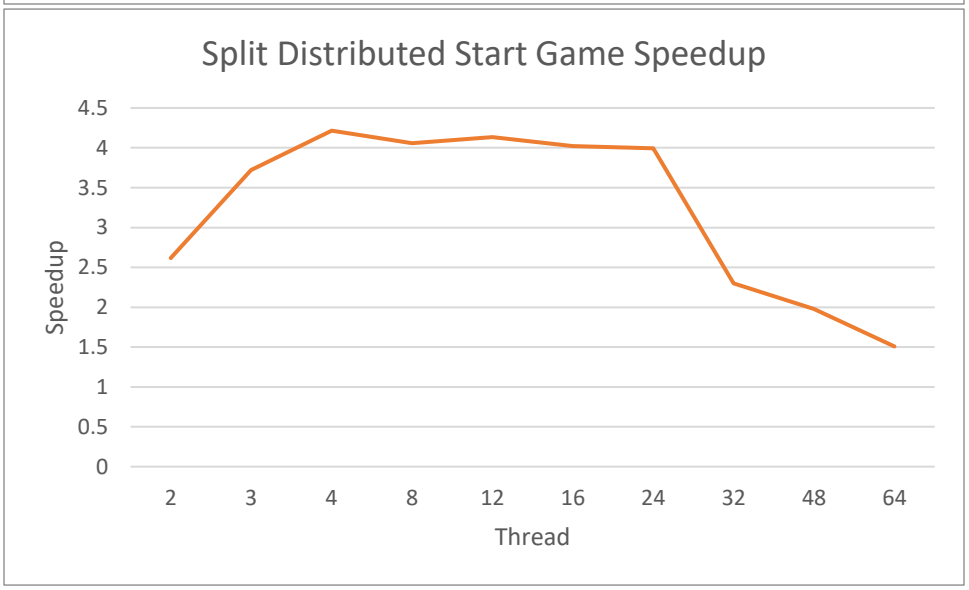
[3]: R. M. Hyatt. The dynamic tree-splitting parallel search algorithm. 20(1):3–19. (Taken from [2])

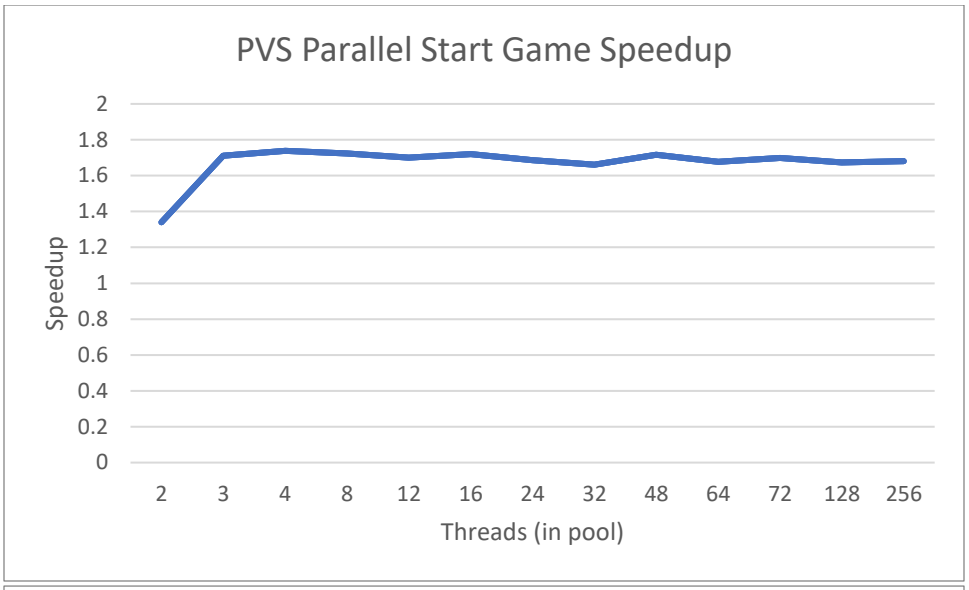
Appendix

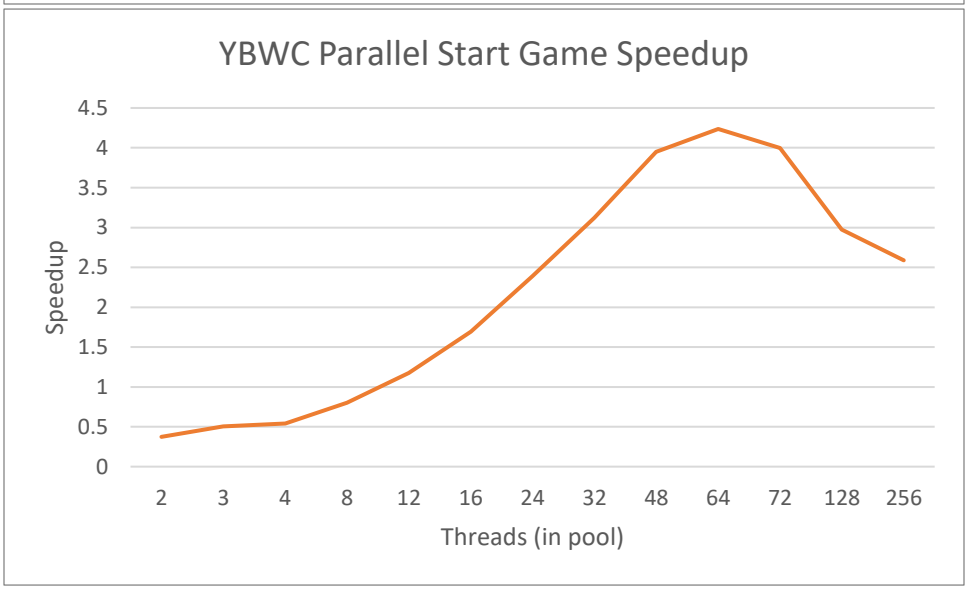


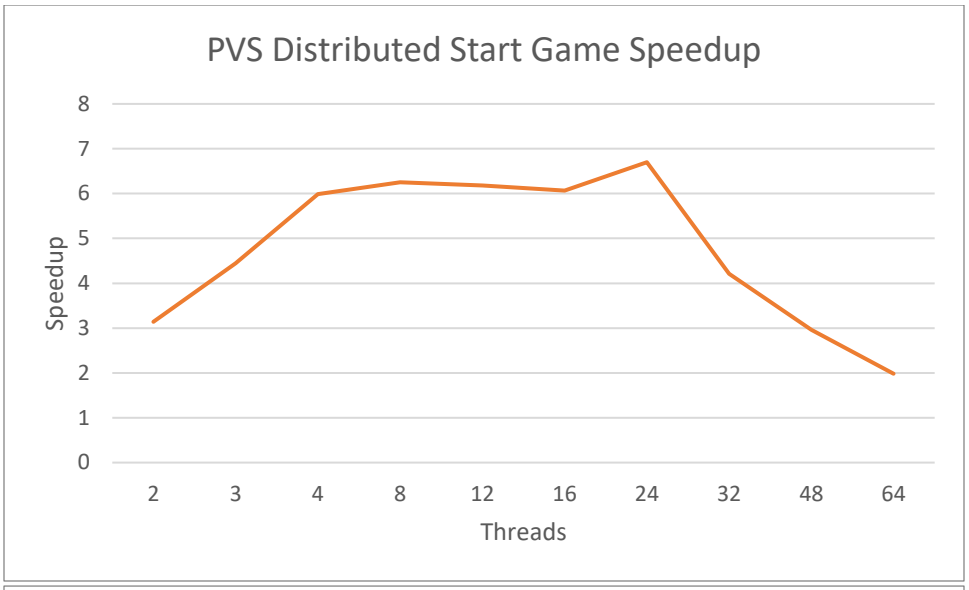


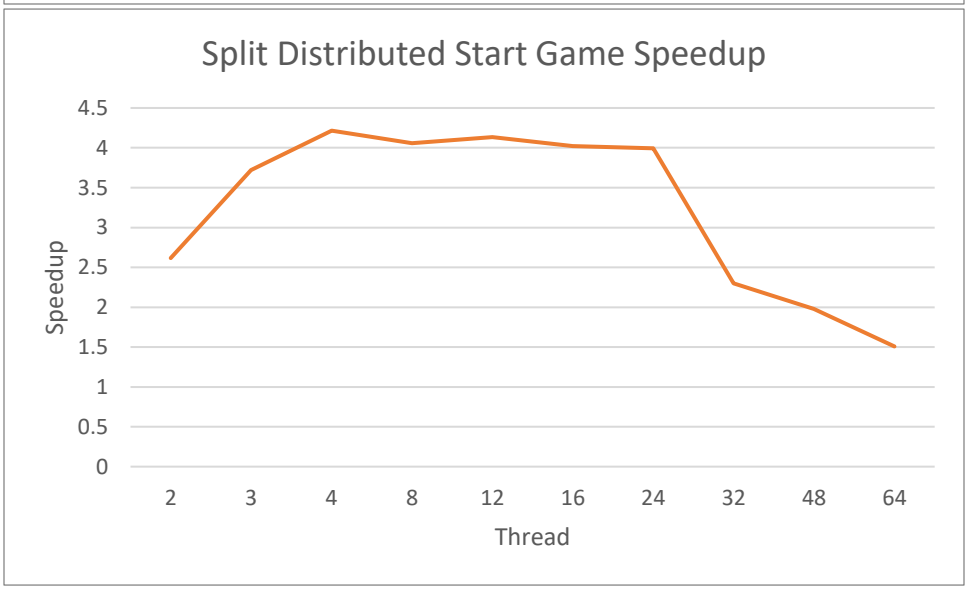

PVS Distributed Start Game Speedup
Speedup
0
1
2
3
4
5
6
7
8
2
3
4
8
12
16
24
32
48
64
Threads

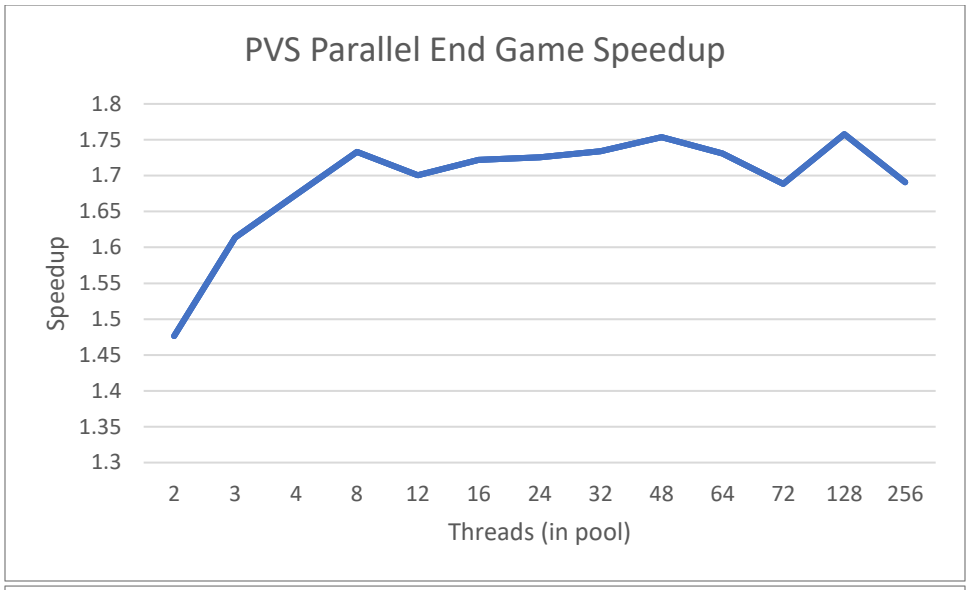

Split Distributed Start Game Speedup
Speedup
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
2
3
4
8
12
16
24
32
48
64
Thread

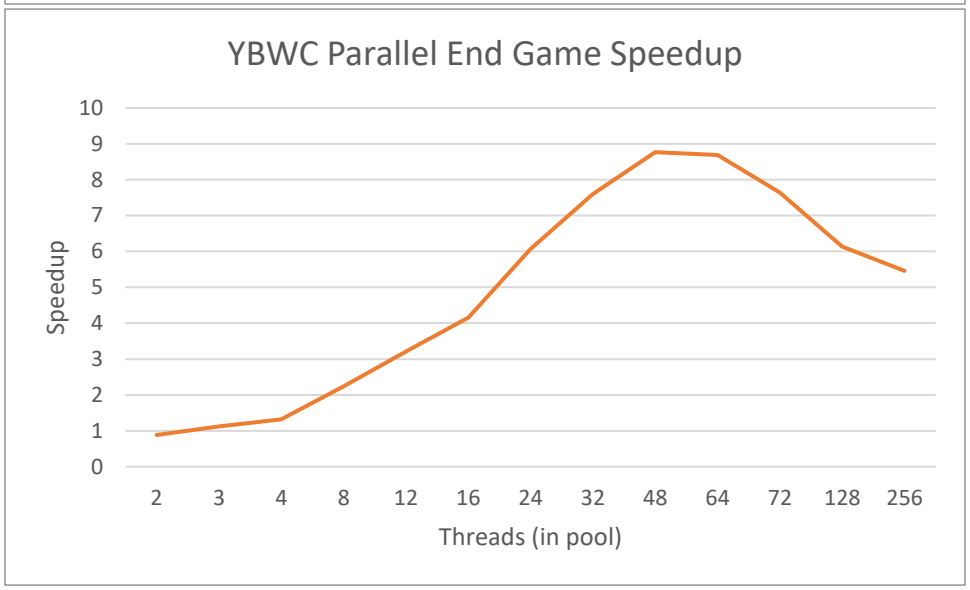

PVS Parallel Start Game Speedup
Speedup
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)

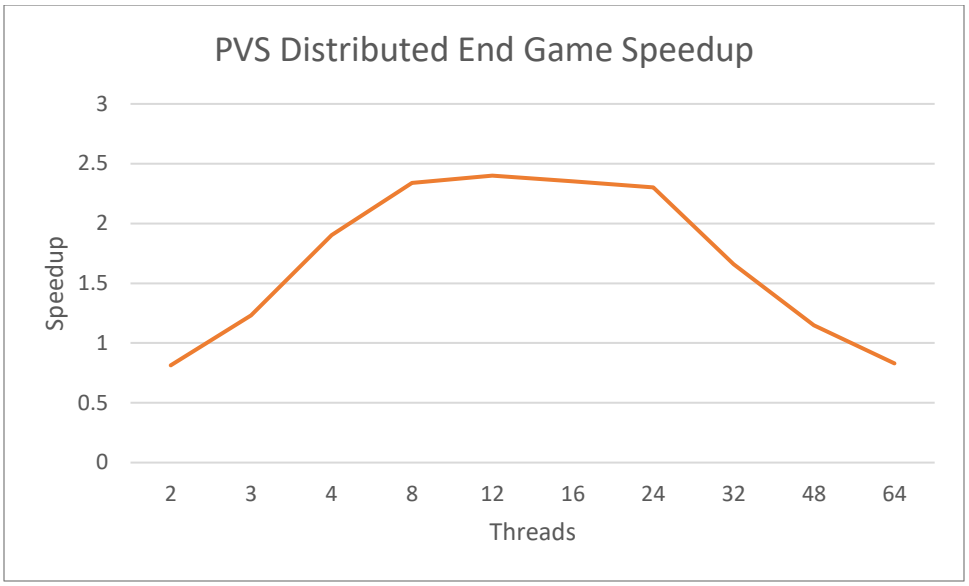

YBWC Parallel Start Game Speedup
Speedup
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)


PVS Distributed Start Game Speedup
Speedup
0
1
2
3
4
5
6
7
8
2
3
4
8
12
16
24
32
48
64
Threads


Split Distributed Start Game Speedup
Speedup
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
2
3
4
8
12
16
24
32
48
64
Thread


PVS Parallel End Game Speedup
Speedup
1.8
1.75
1.7
1.65
1.6
1.55
1.5
1.45
1.4
1.35
1.3
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)


YBWC Parallel End Game Speedup
Speedup
10
9
8
7
6
5
4
3
2
1
0
2 3 4 8 12 16 24 32 48 64 72 128 256
Threads (in pool)


PVS Distributed End Game Speedup
Speedup
3
2.5
2
1.5
1
0.5
0
2
3
4
8
12
16
24
32
48
64
Threads