15418 Project - Final Report

By Kevin Lee and Manik Panwar

INTRODUCTION

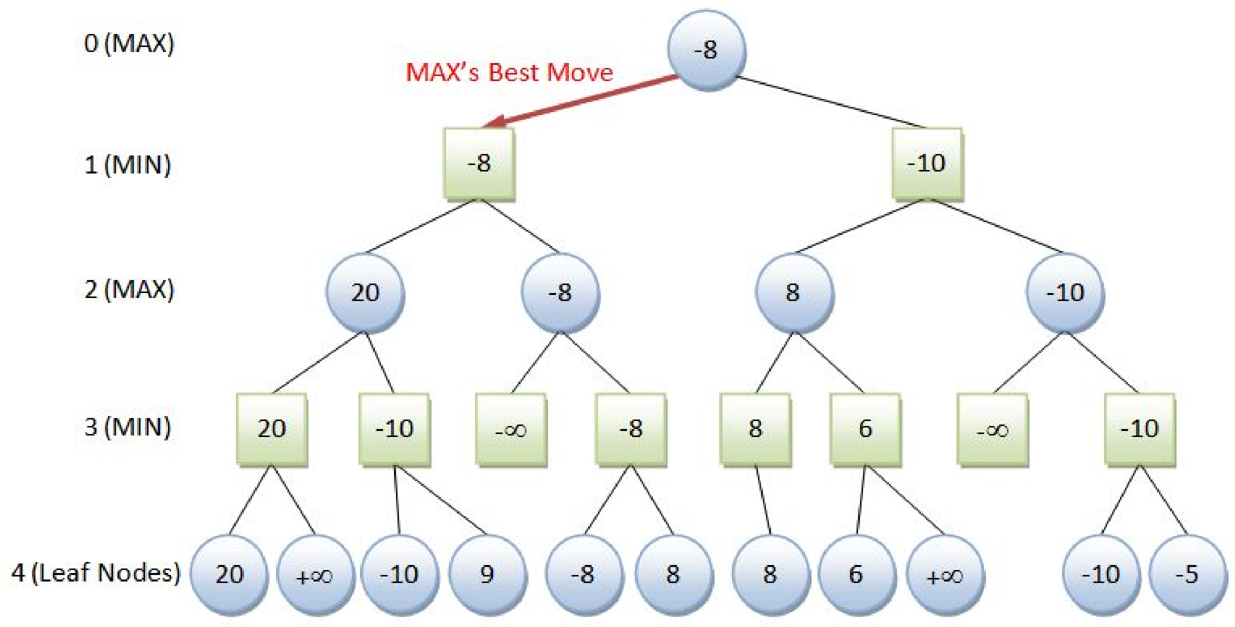
For our project, we investigated the problem of coming up with fast parallel artificial intelligence for the game of Chess. We tried multiple approaches and algorithms to tackle this problem.

Currently, the accepted best algorithm to write a chess AI is minimax augmented with alpha-beta pruning. This algorithm creates a tree of possible moves, and alpha-beta pruning is simply used to ensure that no time is wasted in exploring worse moves than have already been seen. However, the tree of possible moves from a given state in chess is way too large for a computer to completely examine, so we will have to stop the algorithm at some point and judge the state using various heuristics. We will take advantage of parallelism when we traverse the game tree in order to reduce the amount of time spent calculating each move.

We tried multiple iterations to land on our final algorithm - starting off with regular minimax to realising how much pruning and parallelism are essentially for good speedup and times to finally ending up implementing the PVSplit AI algorithm which gave us a good mix of pruning and parallelism. The AI algorithms have previously been explored before but our project was able to parallelise them using multiple approaches and come up with data on which one lended itself best to the problem. We were also able to gain valuable insight on what type of hardware would be the best to run this AI on by getting familiar with the problem and comparing different approaches.

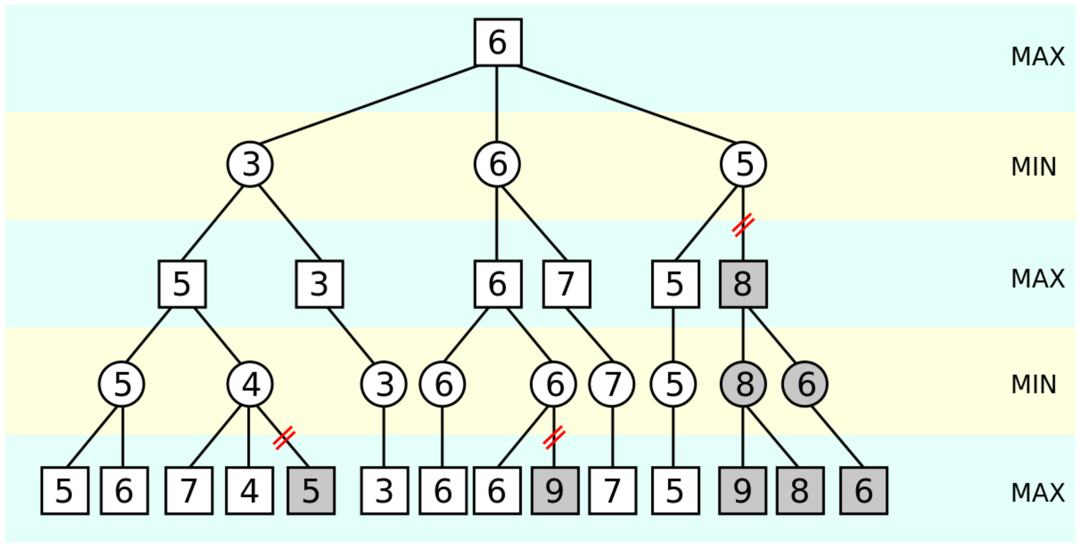
DESIGN / APPROACH

Our entire project was done in C++. We started off with coming up with a complete Chess Game engine implementation which was specifically tailored to our AI needs (had functions such as appleMove, isLegalMove and undoMove). This took a significant amount of time and we worked on this in parallel with coming up with a sequential implementation of an the Minimax algorithm for the AI. The Minimax algorithm essentially simulates the game with at the current state up to a certain depth of moves (which we specify) and draws a game tree of all the possible options. It needs a heuristic to score each state when we reach a leaf node in the game tree and it returns the best possible move which gives the current player the best score given current state (it essentially tries to minimise the loss for the current player in the worst case when the opponent makes all the right decisions). Note that higher depths allow the algorithm to look further into the future and make better decisions but this also significantly increases time taken by the algorithm since increasing depth increases the size of the game tree exponentially.



Minimax algorithm sample tree
(https://www.ntu.edu.sg/home/ehchua/programming/java/images/GameTTT_minimax.png)

Once we had an implementation of Minimax working we converted that to alpha beta pruning. Alpha beta pruning essentially stops looking the the subtrees in the search tree which it knows the opponent would never let you get to because they already have a better move that would give them a better score rather than letting you get here and lowering their score. So it essentially 'prunes' away these subtrees leading to a significant decrease in the number of nodes that need to be searched. It is to be noted that alpha beta pruning still returns the same result that minimax would have but it is just faster.



Alpha Beta Pruning: Greyed out nodes are pruned here
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#/media/File:AB_pruning.svg

Once Alpha beta pruning was implemented we started looking at possible approaches of parallelising. One of the big challenges we faced was that alpha beta pruning is inherently a sequential algorithms since it waits for search on one child of the current node to finish before it moves on to the next child of the current node (and so on, possible pruning too as it gets updated Alpha beta bounds on each of these searches finishing). On the other hand the Minimax algorithm lends itself very well to parallelism because each of the subtrees at the current level can be searched in parallel and do not have to wait for the other subtrees at its level to finish. However, we didn’t do just parallel minimax because we wanted pruning to be happening too. As we increase depths time taken by Minimax increases exponentially and even parallel Minimax wouldn’t be able to compete with the ordinary sequential Alpha beta algorithm. We observed this phenomenon in the timings we did too: when we timed the subtree searches for all the 20ish subtrees at the top level we found that in Alpha beta the leftmost subtree would take the most time but once we had decent Alpha beta bounds pruning would start to show its effects and all the other subtrees would finish more than 10x faster. So it was clear that our final algorithm needed to be a mix of pruning and parallelism. Before we landed on our final algorithm we still tried out different approaches which gave us great insights.

One problem we had to deal with was shared game state:

- In our implementations at each node we would update the game state, recurse down a level and then undo our changes. If we were to port this behavior directly to our parallel algorithm we would have to lock at each node which would essentially lead to a sequential algorithm. So we came up with two approaches to address this issue
    - Create a new game state for each thread and each thread just modifies its own local game state. This is what we used in our final implementation.
    - Keep track of moves that the AI makes as it goes down a path in the tree and at the leaf nodes (i.e when you reach maxDepth), when you have to score the game:
        - Lock the shared game state
        - Apply the moves from the moves list that led to this leaf node
        - Score the game
        - Undo the moves from the moves list in reverse order
        - Unlock the game state
    - This implementation ensures that even though we were keeping a shared game state we would have to lock only at the leaf nodes. We didn’t end up using

moves list in our final implementation because we timed the game state deep copy calls we were making in our other implementation (which created a game state for each thread) and they were less than a millisecond so we decided to use that because locking would introduce its own overhead on top of the overhead of the additional complexity introduced by moves lists.

Parallelism Timeline:

1. Parallelizing by running a POSIX thread for each of the first ~20 ish children of the root node and running alpha beta search in each of these threads. This really didn't give us good time improvements because we were still doing most of the work sequentially and on top of that had to deal with the overheads of pthreads and waiting for them to join. Another factor was that the pruning was happening one level down so each of the top 20 subtrees did not have the best alpha beta bounds so there was overall less pruning happening.
2. Afterwards, we implemented the PVSplit algorithm (described later) which incorporates alpha beta pruning and lets us search the subtrees in parallel.
    a. Initial implementation for this was using POSIX threads. This did not give good speedup because of static assignment so lots of threads were running idle waiting on others. Another problem with POSIX threads was that it would be very difficult to create a static number of threads and dynamically assign work to them. Moreover, there was also lot of overhead in pthread_create for all these threads and then waiting for them to join.

b. So, we decided to port our implementation of the PVSplit algorithm to OpenMp. This was very useful because it allowed us to dynamically work-balance the threads to ensure threads weren't running idle for most of the time and allow us to use a static number of threads. This gave us good speedup and this was the implementation we went with.

3. ***Final Algorithm*** : PVSPLIT ALGORITHM with OpenMP

    a. Description:

        i. We call PVSplit recursively on the left most child of current node.

        ii. We use the alpha beta bounds returned by this PVSplit call to run regular Alpha beta pruning on the other children of the current node in parallel. This way we can still have pruning on these trees but still run them in parallel.

    b. Using OpenMp was useful because we could use its workload balancing so it ensured that we did not have most of our threads running idle so that as they got over (or were pruned) they could take over other remaining work.

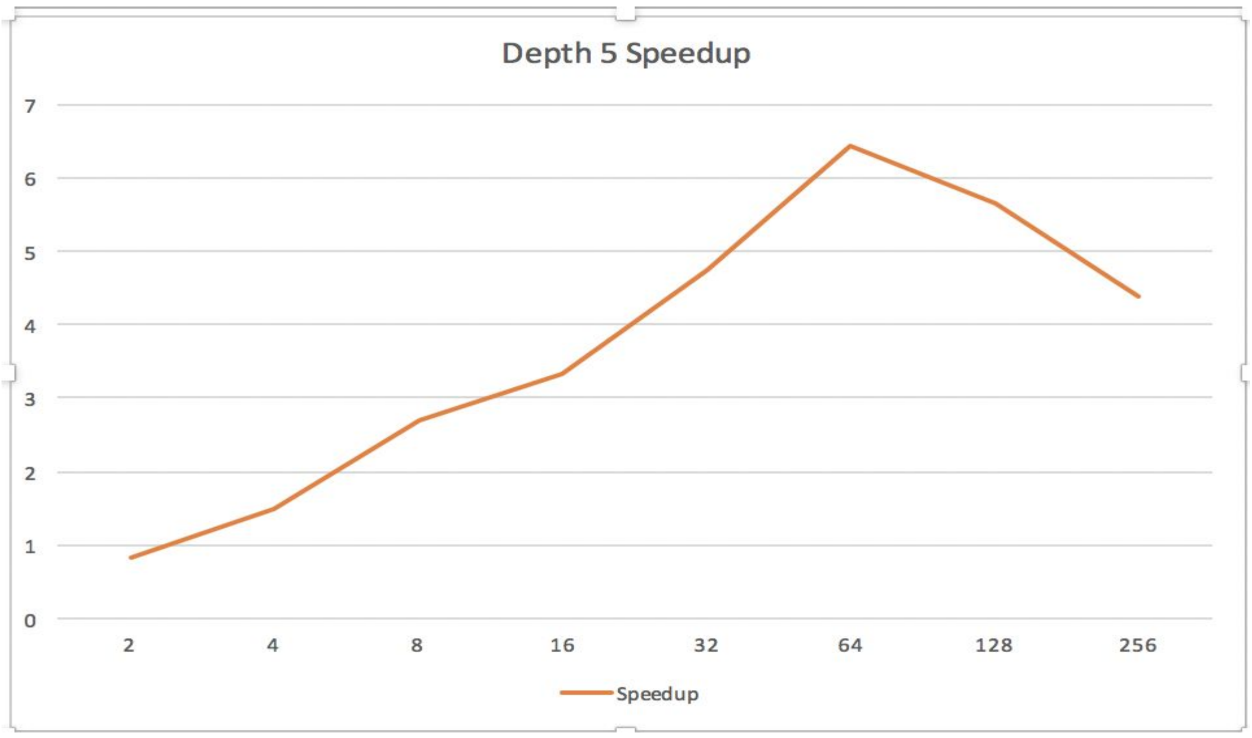
EXPERIMENTAL SETUP

We ran our timing code on unix.andrew.cmu.edu.
Compile command: g++ -o pvsplitOpenMPChess chess.cpp pvsplitOpenMp.cpp -fopenmp -O3 -m64
Run: ./pvsplitOpenMPChess

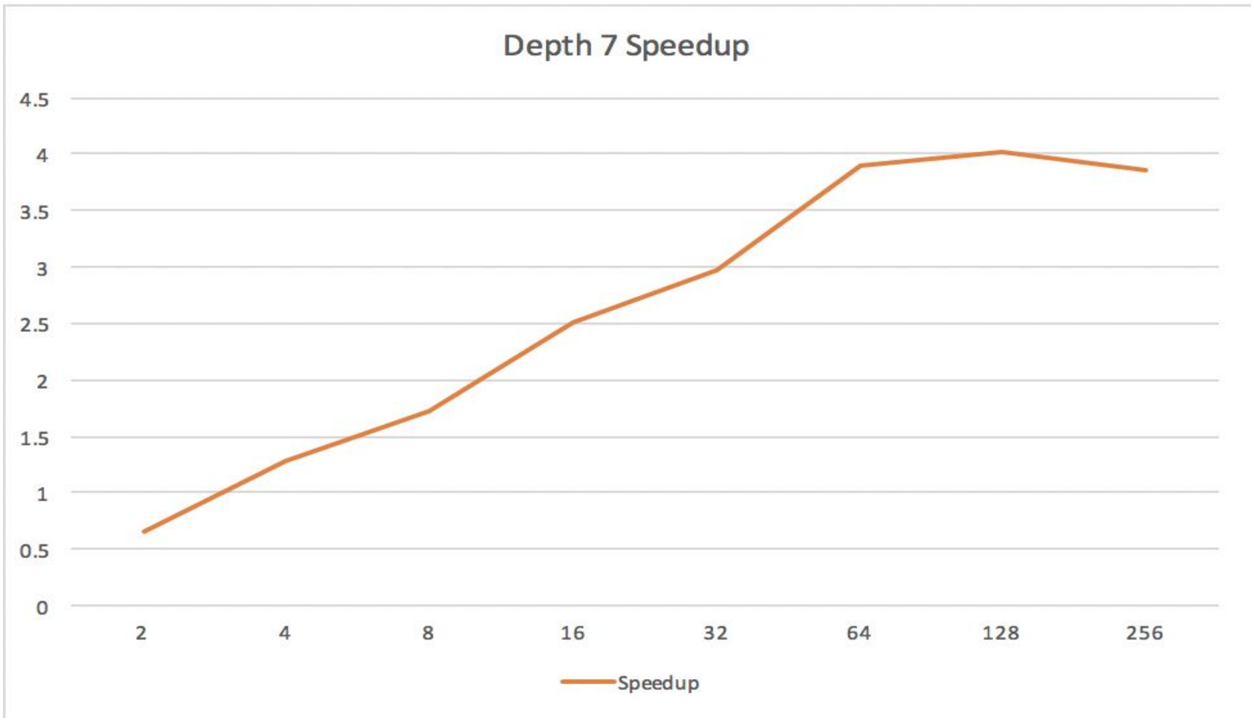
Doing this will give you White’s opening move as well as the time it took to compute that move.

EXPERIMENTAL EVALUATION

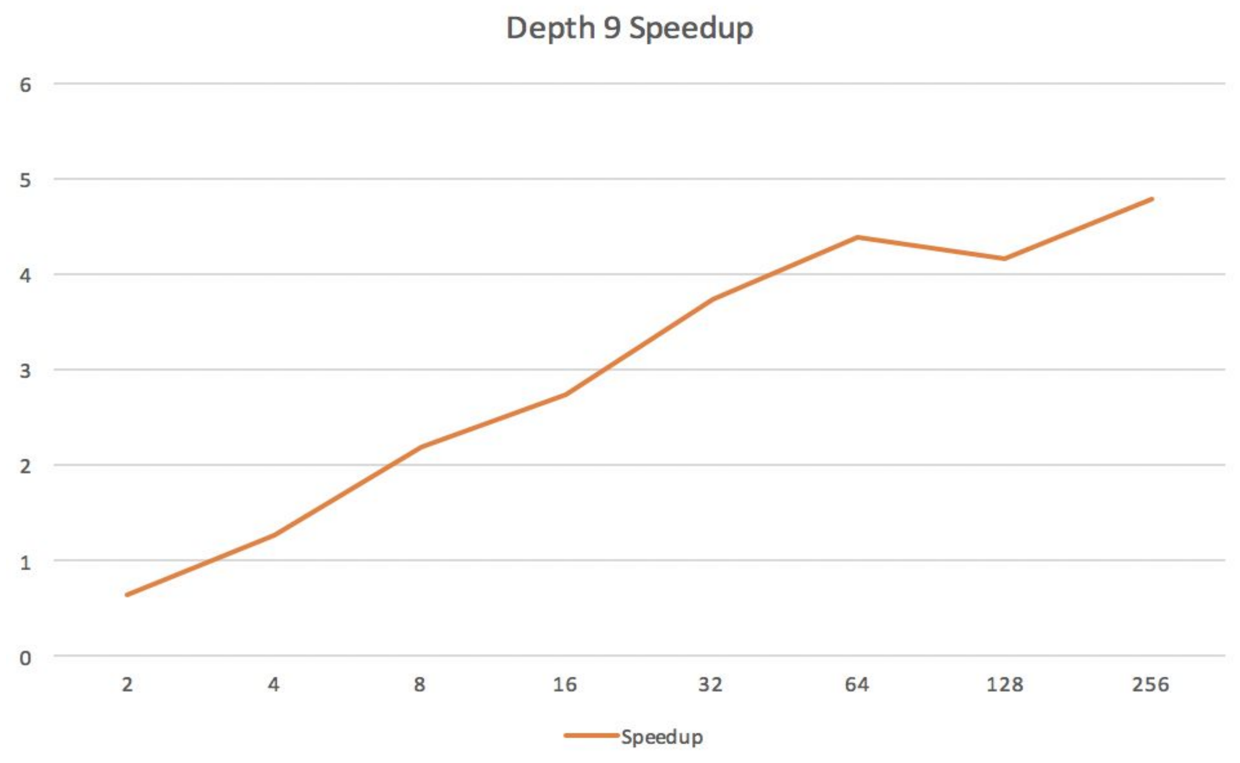
Depth 5		
Threads	Time (ms)	Speedup
1	27	
2	33	0.818181818
4	18	1.5
8	10	2.7
16	8.14	3.316953317
32	5.7	4.736842105
64	4.2	6.428571429
128	4.79	5.636743215
256	6.162	4.381694255



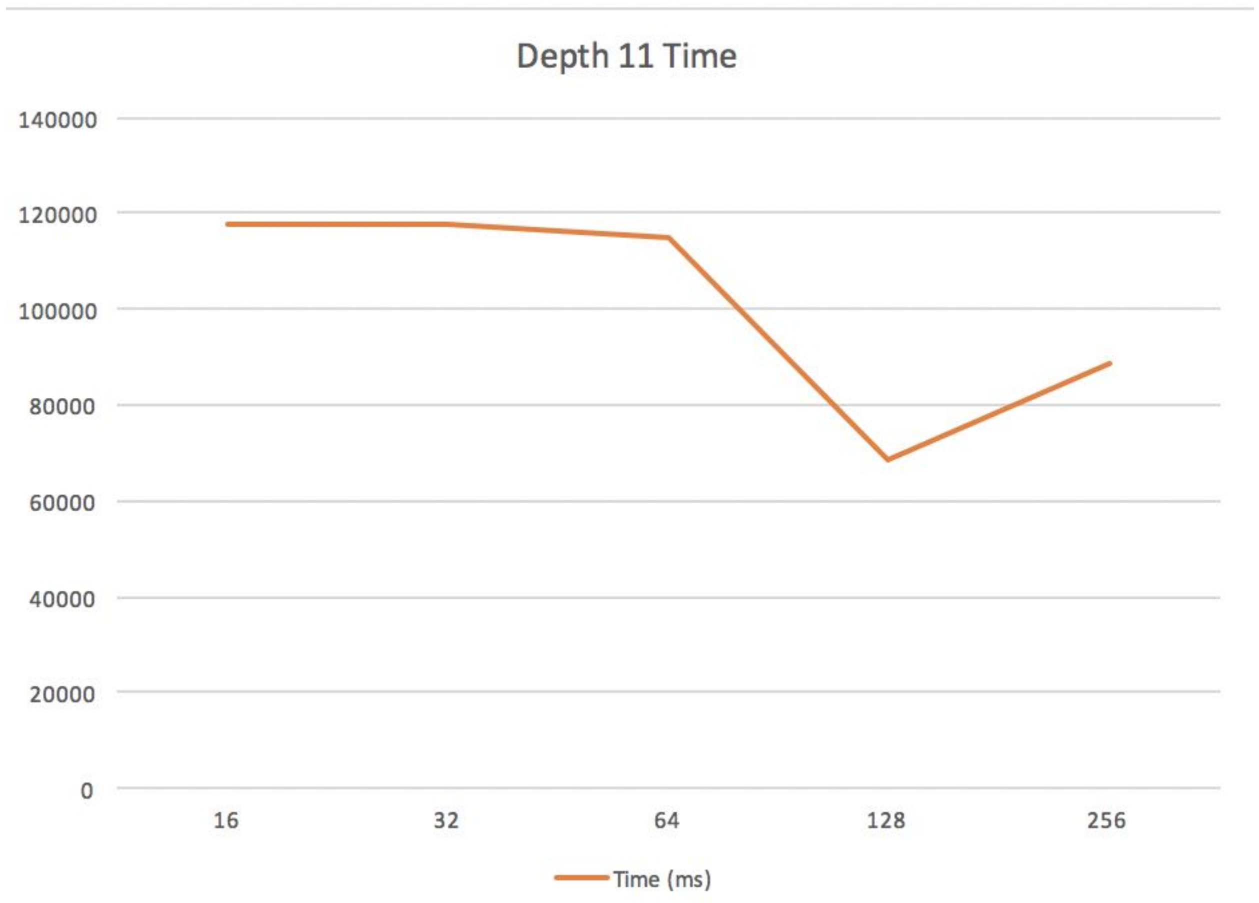
Depth 7		
Threads	Time (ms)	Speedup
1	372.8	
2	567.5	0.6569163
4	292.3	1.275401984
8	215.6	1.729128015
16	148.3	2.513823331
32	125.47	2.971228182
64	95.56	3.901213897
128	92.76	4.018973696
256	96.7	3.855222337



Depth 9		
Threads	Time (ms)	Speedup
1	9901	
2	15213	0.650824952
4	7776	1.273276749
8	4544	2.178917254
16	3635	2.723796424
32	2660	3.722180451
64	2263	4.375165709
128	2382	4.1565911
256	2071	4.780782231



Depth 11	
Threads	Time (ms)
1	-
2	-
4	-
8	-
16	117627
32	117681
64	114834
128	68705
256	88513



*for depth 11, we couldn't plot speedup because running on one thread took way too long

For timing, we found the minimum time it took for the AI to compute its opening move for various maximum depths. We plotted the speedup vs. the number of threads. For depth 5, we can see that there is good speedup up until 64 threads, after which the speedup starts to go down. This is because there are still relatively few nodes in the game tree at depth 5, since for each additional depth the amount of nodes increases exponentially. Since there aren't that many nodes, the benefits to parallelism that more threads give us is overshadowed by the cost of maintaining those additional threads (i.e. creation, synchronization, and destruction). However, for depth 7 and 9, we can see that there isn't a falloff in speedup after 64 threads, and for depth 9 we actually continue to get increasing speedup. This is again because each additional level we traverse increases the number of nodes in the game tree exponentially, so we benefit more and more from increasing the number of threads. For depth 11, we couldn’t plot the speedup because running the algorithm on a low number of threads took way too long, so instead we plotted the amount of time it took our AI to compute the opening move. Again, we get faster times with more threads, which indicates that there is good parallelism.

We were very happy with the results of our design, since clearly there is a lot of parallelism that is occurring. This can be seen by the increasing speedup as we increased the number of threads used.

SURPRISES/LESSONS LEARNED

We had initially wanted to run the AI on a GPU using CUDA. But as we went through our iterations and landed on a final algorithm, we realised that there was still a lot of sequential computation inherent to the problem because we wanted to do Alpha beta search because of its pruning characteristic. So, even though we were running parallel Alpha beta search, each of the individual searches still had a decent amount of work so we actually wanted faster and more powerful sequential processors rather than lots of smaller processors which weren't as fast. So we decided to use the CPU rather than the GPU for this problem. Another reason we ended up using OpenMP over CUDA was because of how well it's abstractions of parallelism lent to this problem and that we could do dynamic workload balancing using OpenMP.

CONCLUSION

We went through many iterations to land on our final algorithm, which was PVSplit running on OpenMP. Future work on this may include adding hash tables to store game states (nodes) that have been seen before, so that if we come across the same game state again we can query the hash table. This can be tried in OpenMp with a shared hash table where both locking and lock free data structure alternatives can be explored. Another option is to try running this solution using MPI message passing distributed across machines and comparing the timing data from these implementations. A solution in Cuda can also be developed just to see how it compares to the solutions running on CPU.

REFERENCES

- http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf
- http://www.contrib.andrew.cmu.edu/~akavka/Final_Report_418-PDF.pdf
- https://github.com/jordancunningham/Chess
- https://chessprogramming.wikispaces.com/Evaluation
- https://en.wikipedia.org/wiki/Alpha–beta_pruning

DISTRIBUTION OF WORK: 50 - 50