On the Scalability of Parallel UCT

Richard B. Segal[⋆]

IBM Research
Yorktown Heights, NY
rsegal@us.ibm.com

Abstract. The parallelization of MCTS across multiple-machines has proven surprisingly difficult. The limitations of existing algorithms were evident in the 2009 Computer Olympiad where ZEN using a single four-core machine defeated both FUEGO with ten eight-core machines, and MOGO with twenty thirty-two core machines. This paper investigates the limits of parallel MCTS in order to understand why distributed parallelism has proven so difficult and to pave the way towards future distributed algorithms with better scaling. We first analyze the single-threaded scaling of FUEGO and find that there is an upper bound on the play-quality improvements which can come from additional search. We then analyze the scaling of an idealized N-core shared memory machine to determine the maximum amount of parallelism supported by MCTS. We show that parallel speedup depends critically on how much time is given to each player. We use this relationship to predict parallel scaling for time scales beyond what can be empirically evaluated due to the immense computation required. Our results show that MCTS can scale nearly perfectly to at least 64 threads when combined with virtual loss, but without virtual loss scaling is limited to just eight threads. We also find that for competition time controls scaling to thousands of threads is impossible not necessarily due to MCTS not scaling, but because high levels of parallelism can start to bump up against the upper performance bound of FUEGO itself.

1 Introduction

The parallelization of UCT across multiple-machines has proven surprisingly difficult. The limitations of existing parallel algorithms was evident in the 19x19 Computer Go tournament at the 2009 Computer Olympiad. In this competition ZEN took first place running on a single four-core machine, FUEGO came in second running on ten, eight-core machines, and MOGO placed third running on twenty, thirty-two core machines.

The most successful distributed parallelization algorithm to date has been the periodic, partial tree-synchronization model employed by MOGO-TITAN[1]. This algorithm has been employed successfully in a number of high-profile wins

[⋆] This work was supported in part by the DARPA GALE project, Contract No. HR0011-08-C-0110.

against high-ranking human players, but was defeated by much smaller systems in the Olympiad.

The limits of existing parallelism techniques has been reflected in our own experiments attempting to scale Fuego to a 16,384 core Blue Gene/P supercomputer. Our system, BlueFuego, incorporates several distributed parallelism algorithms that have been reported in the literature, and so far none of them have achieved meaningful scaling beyond 32 cores.

This paper investigates the limits of parallel MCTS in order to understand why distributed parallelism has proven so difficult and to pave the way towards future distributed algorithms with better scaling. We analyze the scaling of an idealized N-core shared memory machine to determine the maximum amount of parallelism available. This study extends on previous work by considering very large-scale parallelism; up to 512 parallel threads or more; and by doing so in a manner that factors out processor specific differences. We also demonstrate that speedup of MCTS cannot be analyzed in isolation, but is highly dependent on both search time and the inherent scaling of the particular search under consideration. Finally, we add to the growing literature showing the importance of virtual loss for parallel MCTS.

The next section describes Fuego and its use of MCTS search. The following section presents our experimental methodology including how we simulate an idealized N-core machine in Fuego and how we use the simulator to analyze very large-scale parallelism. We then analyze Fuego's single-threaded scaling both to bound Fuego's overall performance and to provide a reference point for measuring multi-threaded performance. The subsequent section analyzes the scaling of parallel MCTS and presents our primary results. The remainder of the paper discusses the implications of these results for both multi-threaded and multi-machine parallelism and concludes by discussing related work and summarizing our results.

2 Fuego

Fuego is an open source Computer Go program developed by Martin Müller, Markus Enzenberger, and Broderick Arneson. Fuego implements the UCT[2] Monte-Carlo tree search algorithm with many Go specific optimizations and tweaks to maximize overall play quality. Fuego incorporates many of the technologies that have been shown to be effective in other UCT-based Go programs including RAVE [3], node priors, and a Mogo-style playout policy [4].

Algorithm 1 shows the pseudo-code for Fuego's main search algorithm. Each pass through the loop in GenerateMove() implements a single UCT trial. Each trial consists of a call to PlayInTree() to select a leaf of the UCT tree and a call to PlayoutGame() to play a randomized game from the selected leaf. Evaluate() is then called to determine whether the randomized trial resulted in a win or loss. This value is used to update both the UCT and RAVE statistics stored in the UCT search tree.

The function PlayInTree() implements the UCT leaf-selection algorithm. The selection process starts at the root of the search tree. At each level, the

Algorithm 1. FUEGO's search algorithm.

```
1: function GenerateMove
2:     while TimeLeft() $> 0$ do
3:         SelectedPath $\leftarrow$ PlayInTree()
4:         MoveSequence $\leftarrow$ PlayoutGame(SelectedPath)
5:         Value $\leftarrow$ Evaluate(MoveSequence)
6:         UpdateUctValues(SelectedPath, Value)
7:         UpdateRaveValues(SelectedPath, Value, MoveSequence)
8:     end while
9:     return SelectBestMove(TreeRoot().Children())
10: end function
11:
12: function PlayInTree
13:     CurrentNode $\leftarrow$ TreeRoot()
14:     SelectedPath $\leftarrow$ {CurrentNode}
15:     Finished $\leftarrow$ FALSE
16:     while notFinished do
17:         if IsLeaf(CurrentNode) then
18:             ExpandNode(CurrentNode);
19:             Finished = TRUE
20:         end if
21:         CurrentNode = SelectChild(CurrentNode)
22:         SelectedPath.Add(ChildNode)
23:     end while
24:     return SelectedPath
25: end function
```

child that maximizes FUEGO's node evaluation function is selected. The process is then repeated recursively until a leaf node is reached. The leaf node is expanded by adding child nodes for all legal moves available at that point in the search tree. Finally, one child of the newly expanded node is selected and the selection process terminates by returning the entire sequence of nodes that was traversed from the root of the tree.

Multi-threading in FUEGO uses Tree Parallelism [5]. Each thread performs independent trials on a shared UCT tree. FUEGO's lock-free tree design minimizes synchronization costs and allows FUEGO to achieve near perfect play-strength speedup on at least eight cores [6].

One FUEGO detail that will become important later in our discussion is how FUEGO computes UCT bounds. FUEGO computes UCT bounds using the commonly used UCB1 formula [7], but has a parameter to control the weight of the exploration term. The experiments below were all performed using FUEGO's default parameter settings which sets the weight of the UCB1 exploration term to zero thus effectively eliminating it. This setting has empirically been shown to have the best performance for FUEGO and for several competing systems.

3 Methodology

Our goal is to understand the potential parallel scale up of FUEGO on a Blue Gene/P supercomputer for 19x19 competition play. The Blue Gene/P available at our lab has 16,384 PowerPC 450 cores running at 850Mhz. The system can be run in either virtual-node (VN) mode in which all 16,384 processors operate as independent machines, or SMP mode in which the cores are grouped into 4,096 quad-core shared-memory machines. Blue Gene/P features a high-speed point-to-point network organized as a 3D torus and a specialized broadcast network for fast MPI collective operations.

For comparison, we also consider a cluster of ten 8-core Intel Xeon servers running at 3.0Ghz. The total computational power of this cluster is an order of magnitude less than that of Blue Gene/P. However, the advantage of the Xeon cluster is that each Xeon core executes FUEGO approximately eight times faster than a single Blue Gene/P core.

The scaling of FUEGO (and MCTS) on multi-core systems is dependent on many processor specific variables including instruction set, memory bandwidth, memory latency, cache design, number of floating point units, and processor pipeline depth. These processor specific details can limit parallel scaling to a much lower number of threads than the algorithm intrinsically can support. Furthermore, the number of threads available on mainstream processors is limited and far from the thousands of cores that we wish to analyze in this paper.

We have developed an extension to FUEGO that simulates an arbitrary number of threads on a single processing core. This extension models an idealized multi-core machine for which there is no resource contention between threads. When FUEGO is run with multiple threads, each trial is started without knowing the results of the $N-1$ trials that are executing in parallel. We therefore can simulate multiple threads on a serial machine by placing the results of each trial into a processing queue and only updating the tree with the results of a trial until after an additional $N-1$ trials have been started. Algorithm 2 shows the pseudo-code for this algorithm.

Our simulator assumes that each thread proceeds in lockstep such that calls to PLAYINTREE(), EVALUATE(), UPDATEUCTVALUES(), and UPDATERAVEVALUES() are serialized. While this is not entirely accurate, it does provide a good approximation of actual behavior as the number of violations of these conditions is likely to be small on real systems.

Our basic methodology will be to analyze the effectiveness of FUEGO with varying numbers of simulated threads on 1,000 self-play games. The experiments will be conducted on the Blue Gene/P supercomputer to take advantage of its immense total processing power. There are a number of different ways to measure parallel efficiency of MCTS. We will use strength speedup [5] which is defined as the increase in playing time needed to achieve identical strength play. If an 8-core system with a 15 minute time control performed only as well as a single core system with a 60 minute time control, then the strength speedup of the 8-core system is $60/15 = 4$x.

Algorithm 2. Extension of Fuego's search algorithm to simulate N threads

```
1: function GenerateMove($N$)
2:     $PlayoutQueue \leftarrow \emptyset$
3:     while TimeLeft $> 0$ do
4:         $SelectedPath \leftarrow$ PlayInTree()
5:         $MoveSequence \leftarrow$ PlayoutGame($SelectedPath$)
6:         $PlayoutQueue$.AddLast($MoveSequence$)
7:         if $PlayoutQueue$.Length() $= N$ then
8:             $MoveSequence \leftarrow PlayoutQueue$.RemoveFirst()
9:             $Value =$ Evaluate($MoveSequence$)
10:            UpdateUctValues($MoveSequence$, $Value$)
11:            UpdateRaveValues($MoveSequence$, $Value$, $MoveSequence$)
12:        end if
13:    end while
14:    return SelectBestMove(TreeRoot().Children())
15: end function
```

However, it is virtually impossible directly to measure strength-speedup for a large number of threads at the 60 minute per player time controls typical of competition. A real 512 thread system playing with an hour of search time would use 512 hours of total processor time. To simulate this computation on a single processor would require 512 hours to play just a single game. Playing one thousand 512 hour games is clearly infeasible. Instead, we perform several experiments with varying time controls and use the data collected to make projections about what kind of scaling is likely at the time controls of interest.

4 Single-Threaded Scaling

First we measure the scaling of single-threaded Fuego and use the results to build a model of how Fuego scales with playing time. This model will provide a reference point to measure the strength-speedup of our multi-threaded experiments. We follow the method by Chaslot *et al.* [5] to develop our model but extend their technique slightly to handle self-play data.

To measure Fuego's single-threaded scaling we performed a series of self-play experiments where one player is given twice the total move time as its opponent. We arbitrary set the ELO of one minute game to zero and use the results of the self-play experiments to compute the ELO gain at each time point. Figure 1 shows the resulting scaling curve.

Interestingly, our results are qualitatively different from Chaslot *et al.*'s results for Mango. Where Mango displayed linear scaling with time, the amount Fuego gains from each time doubling decreases as search time grows. We suspect that this difference may be an artifact of the small number of empirical points used in the Mango study. Don Dailey performed a similar study with Mogo and FatMan and found a similar decay in the value of a doubling with increased search time [8].

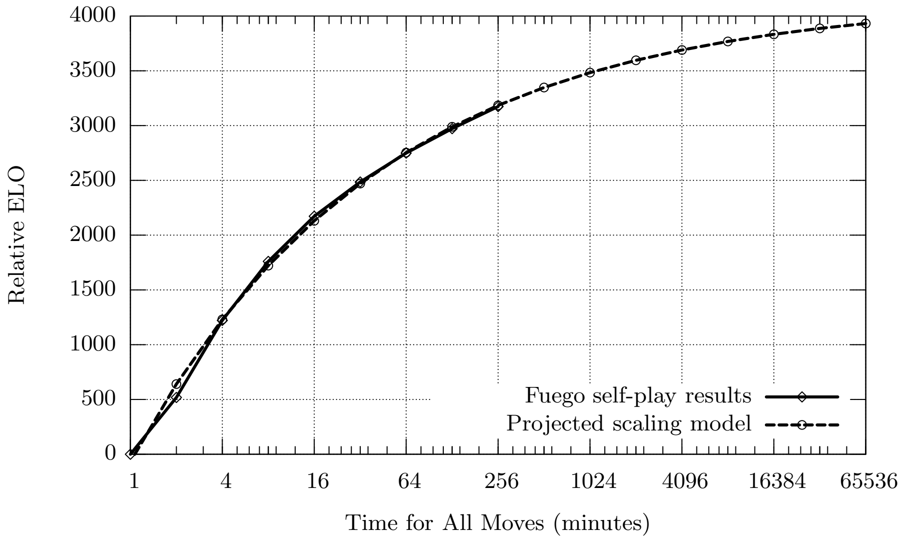

Fig. 1. Actual and projected FUEGO scaling derived from self-play experiments

The decay in the value of each doubling can be accurately modeled as an exponential decay process. Let

$$D = \log_2(t)$$

denote the number of doublings at time t. The ELO gain $\mathcal{G}$ for each doubling can be modeled with an exponential decay function:

$$\mathcal{G}(D) = ab^D$$

with parameters a and b. The total ELO $\mathcal{E}$ for D doublings is then:

$$\begin{aligned} \mathcal{E}(D) &= \int ab^D \, dD \\ &= \frac{ab^D}{ln(b)} + c \\ &= AB^D + C \end{aligned}$$

which gives a parametric equation in A,B, and C. Solving this equation using the non-linear optimization tools in the R statistical package we arrive at the following model for how FUEGO scales with playing time:

$$\mathcal{E}(t) = -4223.6 \times 0.832^{\log_2 t} + 4154.7$$

Figure 1 shows that this curve is indeed a good fit for our empirical data. Interestingly, this curve also suggests an upper bound on FUEGO's total performance of 4,154 ELO above the play quality achieved in a one-minute game on PowerPC 450 processor. As we will discuss in the next section, this upper limit can negatively impact multi-threaded scaling.

5 Multi-threaded Scaling

The scaling of MCTS to thousands of threads is not obvious. When N threads are searching in parallel, each thread must decide which leaf in the UCT tree to expand without knowing the results of the last $N-1$ playouts started, but not completed by other threads. It is entirely possible, if not likely, that once the results of the other $N-1$ threads are known the playout selected by the current thread will not be needed. The issue is more than simply wasting computation on an unneeded playout. Playouts outside the optimal UCT path can dilute UCT's average-value backup such that its convergence to the min-max solution is delayed or even prevented.

We analyze the scaling of MCTS using a three-step process. The first step is to collect empirical data from self-play experiments. Each self-play experiment consists of 1,000 head-to-head matches between Fuego simulating N threads and single-threaded Fuego. Both players are given an equal amount of time to complete all their moves. If Fuego scales perfectly to N threads then both the multi-threaded player and the uni-processor should win about half their games as both players are given identical computation time. If Fuego does not scale perfectly, the multi-threaded version would be expected to win less than 50% of its games and that difference can be converted to a relative ELO rating reflecting the amount of play quality lost using multiple threads. The second step is to fit a curve to the empirical data so that we can estimate the ELO lost for arbitrary length games. The last step is to compute the expected ELO with N threads by subtracting the ELO lost due to threading from the total ELO expected for a given amount of total computation.

We first consider multi-threading without virtual loss. We performed the above self-play experiments using from 1 to 512 threads and from 1 to 128 minutes per player. The results indicate that MCTS scales well to a minimum of 8 threads as the curves for 2, 4, and 8 threads quickly converge to zero loss. The results for 16 threads converge to a constant loss of 110 ELO for all games with time controls greater than 30 minutes. The loss of 110 ELO represents a large fraction of the 162 ELO that could be gained going from 8 to 16 threads and therefore is close to the maximum achievable speedup in this experiment. Figure 3 confirms this analysis by showing the projected scaling curves for a 60-minute time control based on this data.

Figure 4 shows the results of the same experiment when using virtual loss. The graph shows that virtual loss allows Fuego to scale almost perfectly to 32 threads when given a full hour to make its moves. Although, 32 threads is not as effective with a shorter time control such as 15 minutes per side. The shape of the curves suggest that perfect speedup can be achieved with 512 threads on suitable hardware if given sufficient time per move. Remember, in a one-hour

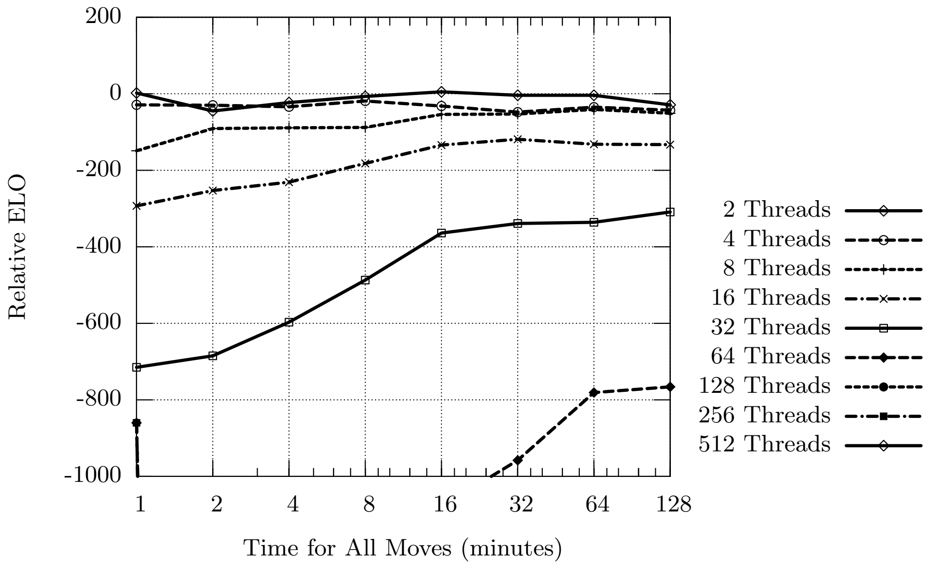

Fig. 2. Self-play of N threads against a uni-processor with equal total computation

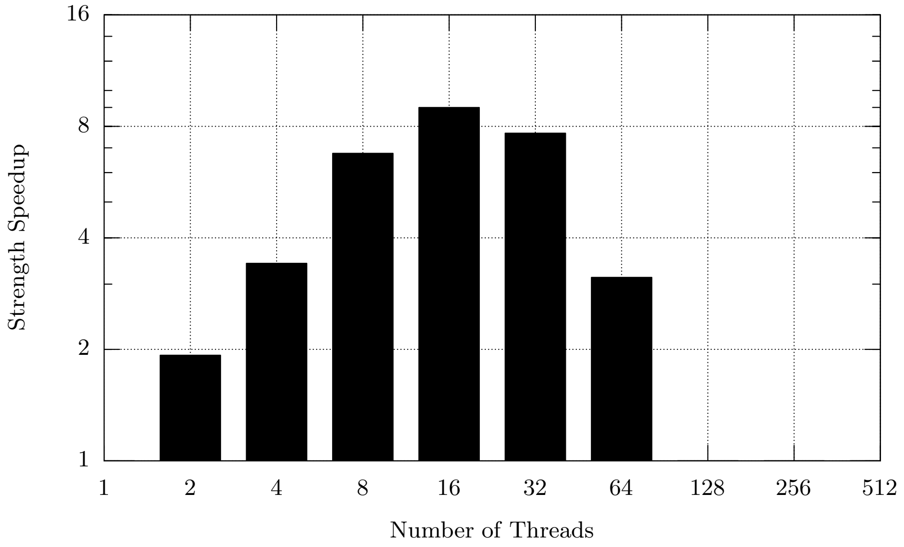

Fig. 3. FUEGO scaling computed from projections of self-play results

game with 512 threads a total of 512 hours of processing is performed. As a result, to achieve perfect speedup with 512 threads the loss needs to converge close to zero at 512 * 60 minutes on this curve.

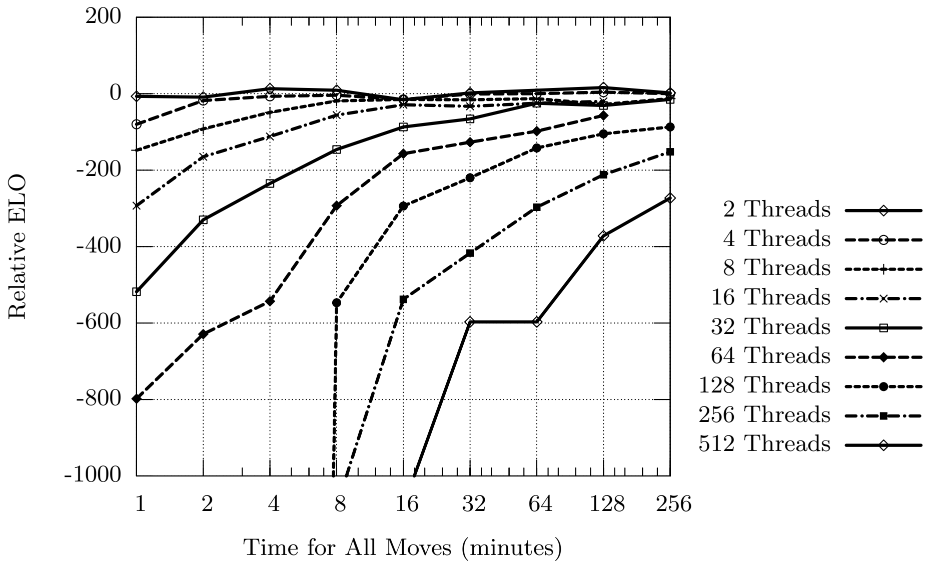

Fig. 4. Self-play of N threads against a uni-processor and virtual loss enabled

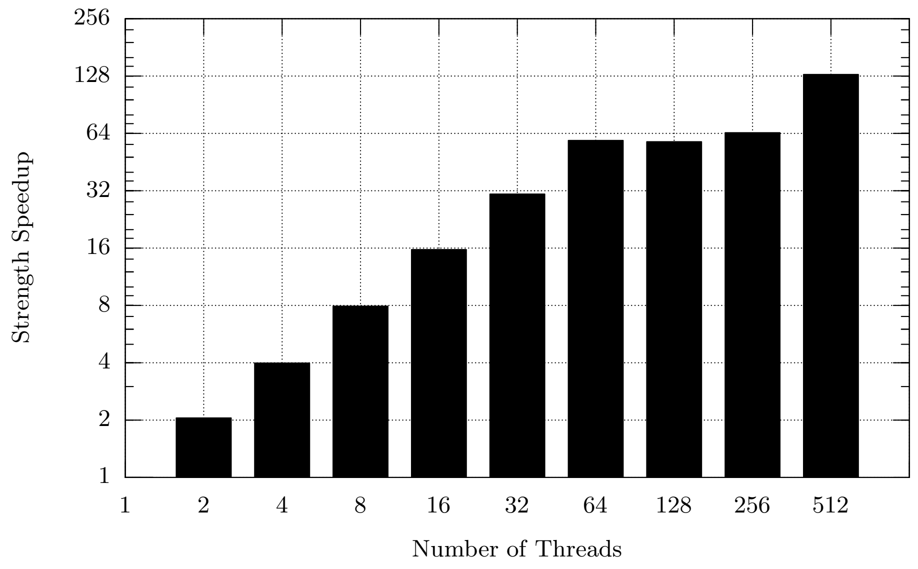

Fig. 5. FUEGO scaling computed from projections of self-play results with virtual loss enabled

This analysis is only partially verified by the results of the projections generated by curve fitting. Figure 5 shows the speedup produced by these projections for 60-minute time controls. Near perfect speedup is only achieved to 64 threads.

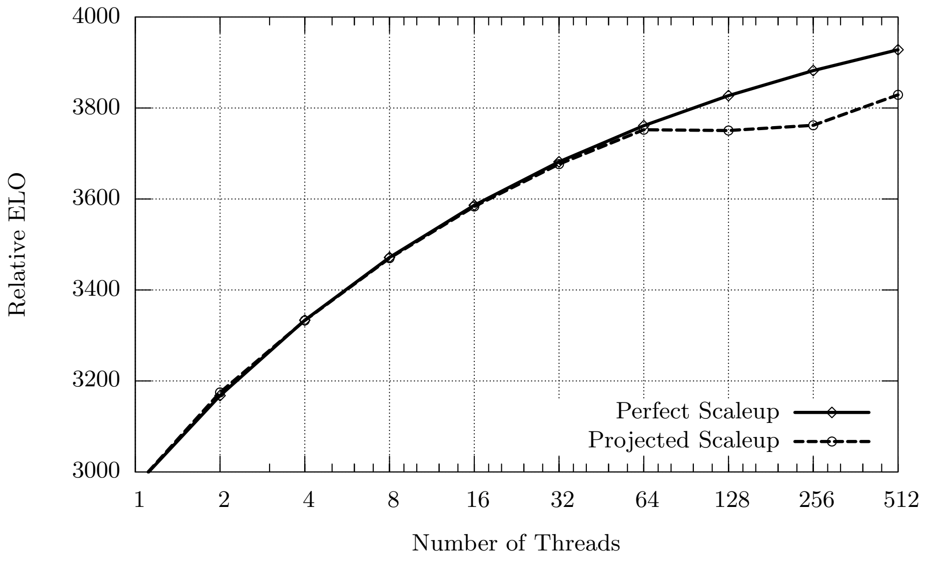

Fig. 6. Projected FUEGO scaling showing that performance at 512 threads is within 300 of our projected maximum of 4154 ELO

The loss curves for 128 to 512 did not converge to zero. Instead, they converged to -78 ELO, -119 ELO, -87 ELO, respectively, suggesting that perfect speedup beyond 64 threads may not be possible. Looking back at Figure 4 it is unclear whether this represents a true scaling failure or inaccuracies that were introduced in the projection process.

Figure 6 displays the same data as Figure 5 as a learning curve. In this graph we can see that using 512 threads FUEGO will be close to 3850 ELO, about 300 ELO away from the maximum possible play quality according to our scaling model. The value of an additional doubling beyond 512 threads is just 46 ELO. Parallel UCT with virtual loss may in fact scale to the 16,384 parallel processors that Blue Gene/P has to offer, but the current FUEGO search algorithm is unlikely to be able to take advantage of it due to the scaling limits of single-threaded FUEGO.

6 Discussion and Related Work

Chaslot *et al.* [5] categorize the types of parallel MCTS into three categories: Tree Parallelization, Leaf Parallelization, and Root Parallelization. Tree Parallelization is the most common. In Tree Parallelization each thread of control performs an independent search on a shared MCTS tree. Leaf Parallelization algorithms use a single thread of control to traverse the MCTS tree, but once a leaf is selected, multiple playouts are performed in parallel on the selected leaf. In Root Parallelization each thread of control updates its own private MCTS tree in parallel. Periodically the threads merge the upper-most portion of their respective trees to achieve mild coordination among the otherwise independent searches.

Each of these algorithms can be implemented on either shared-memory or distributed-memory machines. This paper is primarily concerned with Tree Parallelization as implemented on an idealized shared-memory machine but is applicable to distributed-memory implementations as well. Cazenave and Jouandeau [9] consider distributed Tree Parallelization in which a master process dispatches playouts to be performed remotely. Our results agree with their findings of maximal benefit being achieved with eight parallel threads as they did not use virtual loss in their experiments.

Our work is most similar to Chaslot *et al.*[5] who use a similar methodology to analyze all three types of parallel UCT. They evaluate Tree Parallelization with virtual loss on a real 16-core shared memory machine and discover a maximum strength-speedup of only 8x, far lower than the 130x maximum speedup demonstrated in our experiments. Part of the difference may be due to their consideration of the smaller 13x13 board size which may admit less parallelism than 19x19 Go. But it is likely due to their usage of real hardware rather than an idealized threading model and thereby is more a reflection of the chosen hardware platform than an indication of inherent MCTS limits.

Chaslot *et al.* also evaluate Root Parallelization with very good results; demonstrating near perfect speedup out to sixteen nodes. Our own experiments in this area have been much less successful. We have achieved no more than a 4x speedup with BlueFuego on 8 distributed nodes using virtually the same technique. Interestingly, BlueFuego's speedup appears to depend mostly on the number of distributed nodes and not on the number of threads running on each node. Therefore, BlueFuego achieves a similar 16x speedup when run on Blue Gene/P with 8 SMP nodes with four cores each.

Gelly *et al.* [1] demonstrate positive scaling to at least nine distributed nodes applying Root Parallelization to Mogo. However, a full comparison is not possible as strength-speedup numbers were not provided. Later experiments from the same authors posted to the Computer Go mailing list [10] suggest that with faster networks, Mogo's Root Parallelization can scale nearly perfectly to 32 distributed nodes on 19x19 Go.

The question remains why BlueFuego's Root Parallelism cannot scale beyond eight distributed nodes where others have reported good scaling to either 16 or 32 nodes. The experiments in this paper clearly show that Fuego can achieve greater than 16x strength speedup when using virtual loss. But maybe that is the key. Virtual loss is critical for scaling Fuego to large numbers of threads so a distributed equivalent of virtual loss may be needed to achieve large-scale distributed parallelism.

7 Conclusion

We are interested in developing a parallel version of Fuego that can scale to a 16,384 node Blue Gene/P super computer. This paper investigated the limits of parallel MCTS both to understand whether large-scale parallelism is possible and to provide insight into developing better algorithms for distributed parallelism. We first analyzed the single-threaded scaling of Fuego and found that there is an upper bound on the play-quality improvements that can come from additional search. This in itself limits the potential advantage of Blue Gene/P as its total processing power in a one-hour timed game easily exceeds the computation needed to maximize Fuego's overall play quality.

We then analyzed the scaling of an idealized N-core shared memory machine using empirical data where possible, but using projection to make predictions about larger scale experiments than can be practically performed on existing hardware. Our results show that MCTS can scale nearly perfectly to at least 64 threads when combined with virtual loss, but without virtual loss scaling is limited to just eight threads. This represents the highest degree of scaling for parallel UCT reported in the literature to date. Our results suggest that further scaling to 512 or more nodes may be possible on faster hardware but more data is needed to support this conclusion.

References

1. Gelly, S., Hoock, J.B., Rimmel, A., Teytaud, O., Kalemkarian, Y.: The parallelization of monte-carlo planning - parallelization of mc-planning. In: ICINCO-ICSO, pp. 244–249 (2008)
2. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
3. Gelly, S., Silver, D.: Combining online and offline learning in UCT. In: 17th International Conference on Machine Learning, pp. 273–280 (2007)
4. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France (2006)
5. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel monte-carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
6. Enzenberger, M., Müller, M.: A lock-free multithreaded monte-carlo tree search algorithm. In: Advances in Computer Games 12 (2009)
7. Auer, P., Cesa-Binachi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47, 235–256 (2002)
8. Dailey, D.: 9x9 scalability study (2008), `http://cgos.boardspace.net/study/index.html`
9. Cazenave, T., Jouandeau, N.: A parallel monte-carlo tree search algorithm. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 72–80. Springer, Heidelberg (2008)
10. Teytaud, O.: Parallel algorithms. Posting to the Computer Go mailing list (2008), `http://computer-go.org/pipermail/computer-go/2008-May/015074.html`