Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors

S. Ali Mirsoleimani*†, Aske Plaat*, Jaap van den Herik* and Jos Vermaseren†
*Leiden Centre of Data Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
†Nikhef Theory Group, Nikhef
Science Park 105, 1098 XG Amsterdam, The Netherlands

Abstract*—In recent years there has been much interest in the MCTS algorithm, a new, adaptive, randomized optimization algorithm. In fields as diverse as Artificial Intelligence, Operations Research, and High Energy Physics, research has established that MCTS can find good solutions without domain dependent heuristics. However, practice shows that reaching high performance on large parallel machines is not so successful as expected. So far, the reasons are not well understood. This paper investigates the scalability of two popular parallelization approaches (tree parallelization and root parallelization) of the MCTS algorithm, using the Intel Xeon Phi highly multi-threaded shared-memory system. Moreover, we compare the results on a Xeon CPU and a Xeon Phi to *understand* the scalability of the parallel MCTS algorithms, and to understand their absolute performance. We find that tree parallelization can achieve near perfect speedup for up to 16 threads on the Xeon CPU and up to 64 threads on the Xeon Phi. For root parallelization we find that the effect of locks is small. Moreover, we establish the overall parallel speedup of the two parallelization methods of the MCTS algorithm is fundamentally limited on the Xeon Phi for games such as Hex or Go. The limiting factor is not, as might be expected, the parallel algorithm, or its implementation, but the high level of sequential calculations in each thread, for which no vectorization method is known.

I. Introduction

Since its inception in 2006 [1], the Monte Carlo Tree Search (MCTS) algorithm has gained much interest among optimization researchers. Starting with the game of Go, an oriental board game, MCTS has achieved performance breakthroughs in domains ranging from planning and scheduling to high energy physics [2], [3]. MCTS is a sampling algorithm that uses search results to self-guide the algorithm through the search space, obviating the need for domain-dependent heuristics. One way to improve the quality of the search is to increase the number of samples and thus enlarge the size of the MCTS tree. This can be done by parallelizing the search. Earlier work identified three main parallelization approaches: leaf, root, and tree parallelization [4]. With a few modifications, these are still the main parallelization strategies among researchers.

In 2012/2013 Intel introduced the Intel Xeon Phi (Xeon Phi), a new architecture featuring, for the first time, large scale shared memory parallelization, with support for up to 244 hardware threads [5]. It is speculated that future architectures might support even higher numbers of hardware threads. An important question, therefore, is to what extent the parallelization of the three strategies of the MCTS algorithm will be effective on modern parallel processors such as Xeon Phi. Mirsoleimani et al. provided a first study using a state of the art Go program for lock-free tree parallelization [6]. It showed that achieving good performance on the Xeon Phi was a non-trivial task, and that more research was needed to fully understand the complications. One of the complicating factors is that the search in MCTS contains sequence-dependencies, e.g., results in the early part of the search influence the expansion strategy later on in the search. The search sequence of a parallel execution will therefore be different from a sequential execution, and typically be less efficient. The inefficiency is defined as *search overhead*. The complication gives rise to two different speedup-measures by which the efficiency of the MCTS algorithm parallelizations is measured: (1) playout speedup and (2) strength speedup. The first measure corresponds to the improvement in execution time caused by doing more simulations per second (excluding search overhead), and the second measure corresponds to the improvement in quality of playing (including search overhead). However, practice shows that reaching good speedups on shared-memory machines has some limitations. Segal's [7] simulation study of tree parallelization on an ideal shared-memory system suggested that perfect strength speedup beyond 64 threads may not be possible, presumably due to increased search overhead. Baudiš et al. reported almost near perfect strength speedup up to 22 threads for a lock-free tree parallelization [8].

These reports warrant an investigation of parallel MCTS beyond 22 threads on real shared-memory hardware. The Xeon Phi co-processor with 244 parallel threads and 61 cores is a perfect candidate. The goal of this paper is to investigate the performance of parallel MCTS algorithms on Xeon CPU and Xeon Phi and to understand whether a comparable high performance parallelization of the MCTS algorithm can be achieved on Xeon Phi. We answer this question by reporting both playout speedup and strength speedup of root and tree parallelization (the two major and most successful parallel implementations of MCTS).

This paper has three main contributions.

- We have performed an in depth scalability analysis of both root and tree parallelizations of MCTS algorithm on Xeon CPU and Xeon Phi for the game of Hex. Previous works only targeted strength speedup on CPU [9] or used simulated hardware [7], or a complex Go program [6].
- Contrary to previous results [10], we show that the

effect of using locks is not a limiting factor on the performance of a tree parallelization for 16 threads on Xeon CPU and 64 threads on Xeon Phi.

- Instead, the Xeon Phi features a high communicate/compute ratio, a factor of 30 higher than for the standard Xeon CPU and a vector pipeline. We find that this high ratio, in combination with the amount of sequential work in MCTS, fundamentally limits performance of parallel MCTS on Xeon Phi to a playout speedup of 30% of the Xeon CPU.

The remainder of this paper is structured as follows. In section 2 the required background information is briefly discussed. Section 3 discusses related work. Section 4 gives the experimental setup, and section 5 gives the experimental results. Finally, in Section 6 we conclude the paper.

II. Background

MCTS is a tree search method that has been successfully applied in games such as Go, Hex and other applications with a large state space [1], [11], [2]. It works by selectively building a tree, expanding only branches it deems worthwhile to explore. MCTS consists of four steps [12]. (1) In the selection step, a leaf (or a not fully expanded node) is selected according to some criterion (see II-A). (2) In the expansion step, a random unexplored child of the selected node is added to the tree. (3) In the simulation (also called playout step), the rest of the path to a final node is completed using random child selection. At the end a score Δ is obtained that signifies the score of the chosen path through the state space. (4) In the backprogagation step (also called backup step), this value is propagated back through the tree, which affects the average score (win rate) of a node. The tree is built iteratively by repeating the four steps. In the games of Hex and Go, each node represents a player move and in the expansion phase the game is played out, in basic implementations, by random moves. In many MCTS implementations UCT is chosen as the selection criterion [13], [14]. UCT provides a trade-off between exploitation and exploration that is one of the hallmarks of the algorithm.

A. *UCT Algorithm*

The Upper Confidence Bounds for Trees (UCT) algorithm addresses the problem of exploitation and exploration in the selection phase of the MCTS algorithm [14]. A child node j is selected to maximize:

$$UCT(j) = \overline{X}_j + C_p\sqrt{\frac{\ln(n)}{n_j}} \tag{1}$$

where $\overline{X}_j = \frac{w_j}{n_j}$, w_j is the number of wins in child j, n_j is the number of times child j has been visited, n is the number of times the parent node has been visited, and $C_p \geq 0$ is a constant. The first term in the UCT equation is for exploitation of known parts of the tree and the second one is for exploration of unknown parts [13]. The level of exploration of the UCT algorithm can be adjusted by the C_p constant.

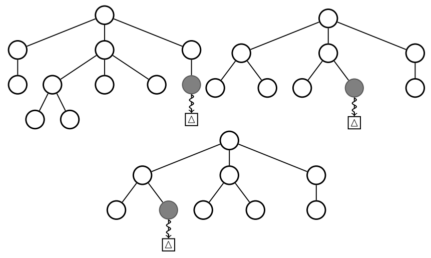


Fig. 1: Different independent MCTS trees are used in root parallelization

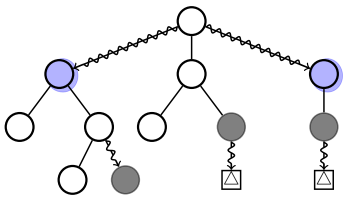


Fig. 2: Tree parallelization with local lock. The curly arrows represent threads. The grey nodes are locked ones. The dark nodes are newly added to the tree.

B. *Parallel MCTS*

In MCTS root parallelization [4], each thread builds simultaneously a private and independent MCTS search tree with a unique random seed. Fig. 1 illustrates root parallelism. When root parallelization wants to select the next move to play, one of the threads collects the number of visits and the number of wins in the first level nodes of all trees and then computes the total sum for each child [4]. Subsequently, it selects a move based on one of the possible policies. Note that UCT with root parallelization is not algorithmically equivalent to plain UCT, but is equivalent to Ensemble UCT [13].

In tree parallelization one MCTS tree is shared among several threads that are performing simultaneous searches [4]. The main challenge in this method is using data locks to prevent data corruption. Figure 2 shows the tree parallelization algorithm with local locks. A lock-free implementation of this algorithm reportedly addresses the aforementioned problem with better scaling than a locked approach [10].

In our implementation of tree parallelization, locks are only used in the expansion phase of the MCTS algorithm in order to avoid the loss of any information (cf. [10]). To allocate all children of a given node, the algorithm uses (1) A pre-allocated vector of children and (2) A single atomic increment instruction updating the index to the next children at the expansion phase. This allows vectorization of the selection and expansion phases in order to achieve the highest performance.

C. *The Game of Hex*

Previous scalability studies used artificial trees [15], simulated hardware [7], or a complex program [6]. None of them allowed a truly realistic performance profiling and analysis. For this reason, we developed from scratch a program specifically for the purpose of generating realistic trees that are sufficiently

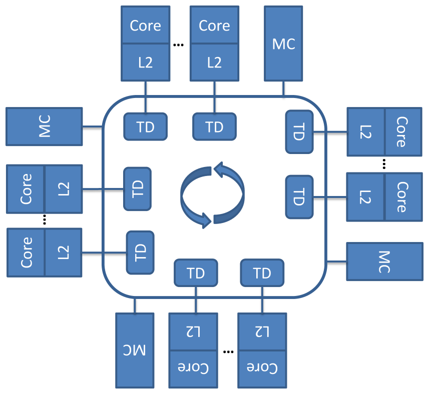


Fig. 3: Abstract of Intel Xeon Phi microarchitecture

clean to allow good performance analysis, using the game of Hex.

Hex is a board game with a board of hexagonal cells [11]. Each player is represented by a color (White or Black). Players take turns placing a stone of their color on a cell on the board. The goal for each player is to create a connected chain of stones between the opposing sides of the board marked by their colors. The first player to complete this path wins the game. Since the first player to move in Hex has a distinct advantage, the swap rule is generally implemented for fairness. This rule allows the second player to choose whether to switch positions with the first player after the first player has made the move.

In our implementation of Hex, a disjoint-set data structure is used to determine the connected stones. Using this data structure the evaluation of the board position to find the player who won the game becomes very efficient [16].

D. Architecture of the Intel Xeon Phi

We will now provide an overview of Xeon Phi co-processor architecture (see Figure 3). A Xeon Phi co-processor board consists of up to 61 cores based on the Intel 64-bit ISA. Each of these cores contains *vector processing units* (VPU) to execute 512 bits of 8 double-precision floating point elements or 16 single-precision floats or 32-bit integers at the same time, 4-way SMT, and dedicated L1 and fully coherent L2 caches [5]. The *tag directories* (TD) are used to look up cache data distributed among the cores. The connection between cores and other functional units such as *memory controllers* (MC) is through a bidirectional *ring interconnect*. There are 8 distributed memory controllers as interface between the ring burst and main memory which is up to 16 GB.

III. RELATED WORK

Below we review related work on MCTS parallelizations. The two major parallelization methods for MCTS are root parallelization and tree parallelization [4]. There are also other techniques such as leaf parallelization [4] and approaches based on transposition table driven work scheduling (TDS) [15] [17].

1) Tree Parallelization: For shared memory machines, tree parallelization is a suitable method. It is used in FUEGO, an open source Go program. In tree parallelization one MCTS tree is shared among several threads that are performing simultaneous searches [4]. It is shown in [4] that the playout speedup of tree parallelization with virtual loss cannot scale perfectly for up to 16 threads. The main challenge is the use of the data locks to prevent data corruption. Figure 2 shows a tree parallelization algorithm with local locks. Moreover, it is shown in [10] that a lock-free implementation of this algorithm provides better scaling than a locked approach. In [10] such a lock free tree parallelization for MCTS is proposed. They intentionally ignored rare faulty updates inside the tree and studied the scalablity of the algorithm for up to 8 threads. In [8], the performance of a lock free tree parallelization for up to 22 threads is reported. The strength speedup is perfect for 16 threads but the improvement drops for 22 threads. There is also a case study that shows good performance of a (non-MCTS) Monte Carlo simulation on the Xeon Phi co-processor [18].

2) Root Parallelization: Chaslot et al. [4] reported results that root parallelization shows perfect playout speedup for up to 16 threads. Soejima et al. [9] analyzed the performance of root parallelization in detail. They showed that a Go player that uses lock free tree parallelization with 4 to 8 threads outperformed the same program with root parallelization which utilizes 64 distributed CPU cores. This result suggests the superiority of tree parallelization over root parallelization in shared memory machines.

IV. EXPERIMENTAL SETUP

In order to generate statistically significant results in a reasonable amount of time, both players do playouts for 1 second to choose a move. The board size is 11x11. To calculate the strength speedup for the first player, we perform matches of two players against each other. Each match consists of 200 games, 100 with White and 100 with Black for each player. A statistical method based on [19] and similar to [6] is used to calculate 95%-level confidence lower and upper bounds on the real winning rate of a player, indicated by error bars in the graphs. C_p is set at 1.0 in all our experiments. To calculate the playout speedup for the first player when considering the second move of the game, the average of the number of playouts per second over 200 games is measured. Taking the average removes (1) The randomized feature of MCTS in game playing and (2) The so called warm up phase on Xeon Phi [20].

The results were measured on a dual socket Intel machine with 2 Intel *Xeon* E5-2596v2 CPUs running at 2.40GHz. Each CPU has 12 cores, 24 hyperthreads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The peak TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program. The machine is equipped with an Intel *Xeon Phi* 7120P 1.238GHz which has 61 cores and 244 hardware threads. Each core has 512KB L2 cache. The co-processor has 16GB GDDR5 memory on board with an aggregate theoretical bandwidth of 352 GB/s. The peak turbo frequency is 1.33GHz. The theoretical performance of the 7120P is 2.416 TFLOPS or TIPS and 1.208 TFLOPS for single-precision or integer and double-precision floating-point arithmetic operations, respectively [21]. Intel's *icc 14.0.1* compiler is used to compile program in *native mode*. A *native application* runs directly on the Xeon Phi and its embedded Linux operating system.

V. Experimental Results

The scalability of parallel MCTS is measured by using empirical data from self-play experiments [4]. Each self-play experiment consists of 200 head-to-head matches between the first player with N threads and the second player with N/2 threads. Both players are given 1 second of time to do a move. If the algorithm scales perfectly then the number of playouts per second for the first player should be two times more than the second player. The percentage of wins (or win rate) for the first player should also be always more than for the second player for a constant rate. The ideal playout speedup is represented by a diagonal straight line and the ideal strength speedup is represented by a horizontal straight line.

A. Playout Speedup

1) Tree Parallelization: In Fig. 4 the scalability of tree parallelization on Xeon CPU and Xeon Phi are compared. Fig. 4a shows playout speedup on Xeon CPU. We see a perfect playout speedup up to 4 threads and a near perfect speedup up to 16 threads. The increase in the number of playouts continues up to 32 threads, although the increase is no longer perfect. There is a sharp decrease in the number of playouts for 48 threads. The available number of cores on the Xeon CPU is 24 cores, with 2 hyperthreads per core available, for a total of 48 hyperthreads. Thus, we see the benefit of hyperthreading up to 32 threads. We surmise that using a lock in the expansion phase of the MCTS algorithm is visible in playout speedup after 4 threads but the effect is not severe. Our results are different from the results in [4] and [11] where the authors reported no speedup beyond 4 threads for locked tree parallelization.

In Fig. 4b the playout speedup on Xeon Phi is shown. A perfect playout speedup is observed up to 64 threads. We see that, using a lock has no effect on the performance of the algorithm up to this point. After 64 threads the performance drops, although the number of playouts per second still increases up to 240 threads. It should be noted that even with playout speedup increasing up to 240 threads, we see that at 240 threads on Xeon Phi still the number of playouts per second is less than on 8 threads on Xeon CPU. In fact the performance for tree parallelization on Xeon Phi is almost 30% of the peak performance on Xeon CPU.

2) Root Parallelization: Next we will discuss the root parallelization, where threads are running independently and where no locking mechanism exists. Root parallelization is well suited to see whether the decrease in playout speedup in tree parallelization is due to locks or not. As is shown in Fig. 5a for Xeon CPU, the playout speedup is perfect for up to 16 threads (while in tree parallelization it is for up to 4 threads). The second difference between these two algorithms is revealed at 48 threads where root parallelization still shows improvement in playout speedup. We may conclude that removing the lock in the expansion phase of tree parallelization improves performance for a high number of threads on Xeon CPU.

The performance of root parallelization on Xeon Phi is shown in Fig. 5b. Here, we require at least 8 threads on Xeon Phi to reach almost the same number of playouts per second as 1 thread on Xeon CPU. On Xeon Phi with root parallelization perfect playout speedup is achieved for up to 64 threads, which implies that the drops on 64 threads in tree parallelization performance are likely not due to locking. However, for 240 threads the number of playouts increases by a higher rate compared to tree parallelization. Overall, the peak performance for root parallelization on Xeon Phi is almost 30% of its counterpart on Xeon CPU.

3) Section Conclusions: To understand the reason for this low performance we performed a detailed timing analysis to find out where the most time of the algorithm has been spent in the selection, expansion, playout, or backup phase. For the Hex board size of 11x11, MCTS spends most of its time in the playout phase. This phase of the algorithm is problem dependent, for example it is different for Go and Hex; and the difference is even different for distinct board sizes. In our program around 80% of the total execution time for performing a move is spent in playout phase.

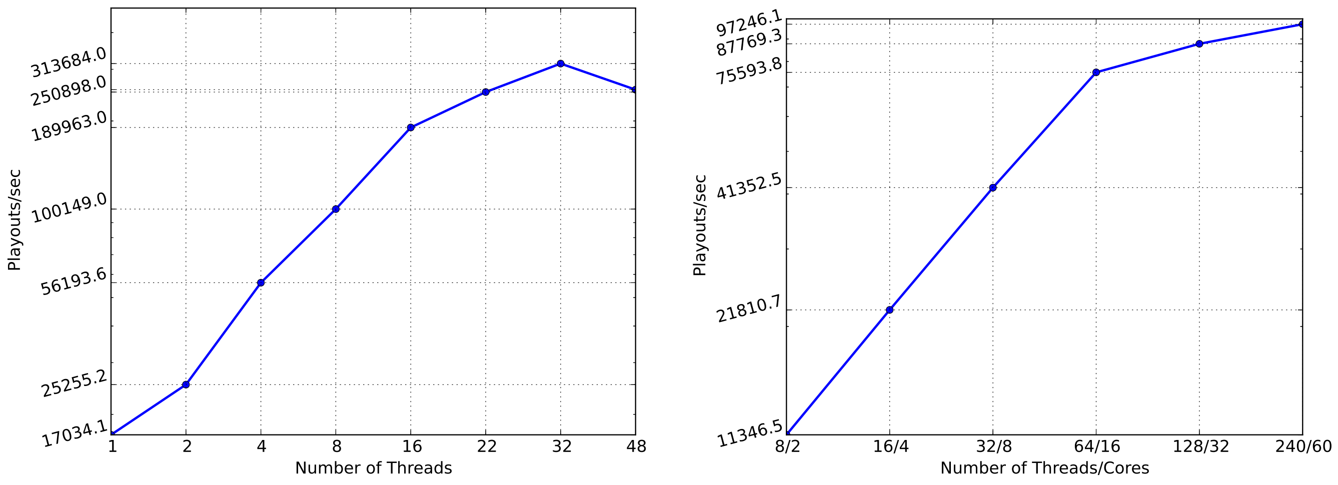
The Xeon Phi co-processor is designed especially for high throughput communication, with a wide memory bandwidth of 352 GB/s, higher than that of ordinary Xeon CPU architectures. Combined with the modest integer performance of 1.33GHz, it has a high communicate/computation ratio, a factor of 30 higher than for the standard Xeon CPU. To achieve full performance on Xeon Phi, programs must make use of communication and vectorization.

Since the playout phase dominates execution time of each thread, Xeon CPU outperforms Xeon Phi significantly because of more powerful cores. No method for vectorization has been devised for the playout phase. Therefore, for the current ratio of Xeon CPU cores versus Xeon Phi cores (24 versus 61) it is not possible to reach the same performance on Xeon Phi because each core of Xeon CPU is more powerful than each core of Xeon Phi for sequential execution. From these results we may conclude that for the current ratio of Xeon CPU cores versus Xeon Phi cores the parallel MCTS algorithms for games such as Hex or Go on Xeon Phi have a limitation. Therefore, it is interesting to investigate the limitation problem in the other domains in which MCTS has been successful such as [2].

B. Strength Speedup

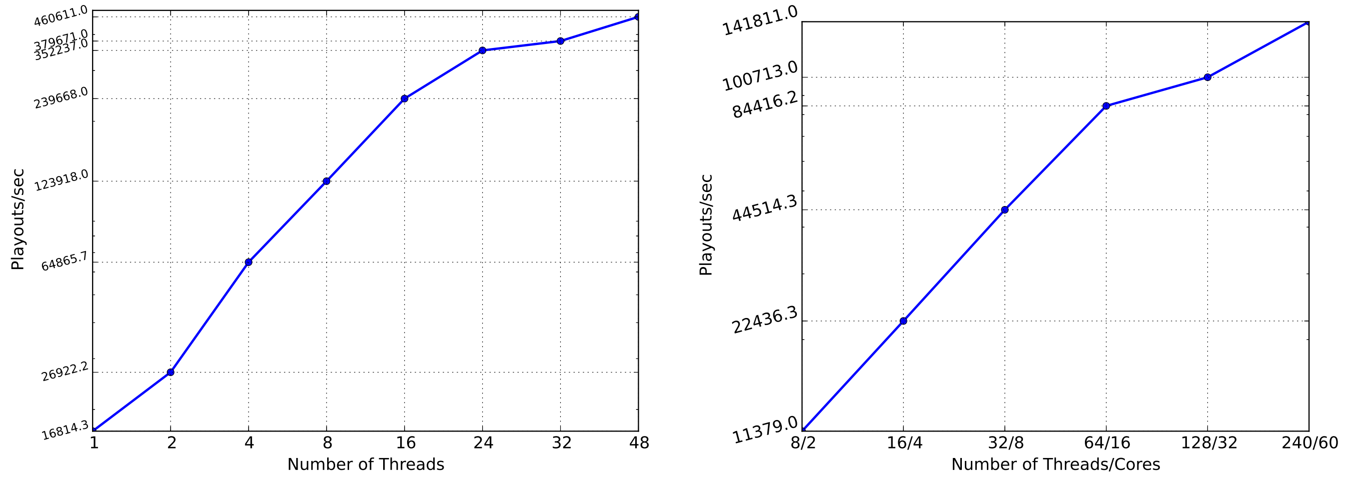
1) Tree Parallelization: As already mentioned, it is also important to evaluate the playing strength of the MCTS player for a game such as Hex. The goal is to see how the increase in the number of playouts per second reflects into a more powerful player. In Fig. 6a strength speedup for tree parallelization on Xeon CPU is shown. Note that, since we compare the performance of N threads against N/2 threads, an ideal perfect strength speedup would give a straight, horizontal line of, say, 60% win rate for the player with more threads.

We see good strength speedup up to 32 threads. The win rate drops to 50 percent for 48 threads. This decrease in win rate is consistent with the drop in the number of playouts per second for 48 threads in Fig. 4a. On the Xeon CPU, strength speedup follows playout speedup closely.



(a) (b)

Fig. 4: Playout speedup for tree parallelization. (a) Xeon CPU (b) Xeon Phi.



(a) (b)

Fig. 5: Playout speedup for root parallelization. (a) Xeon CPU (b) Xeon Phi

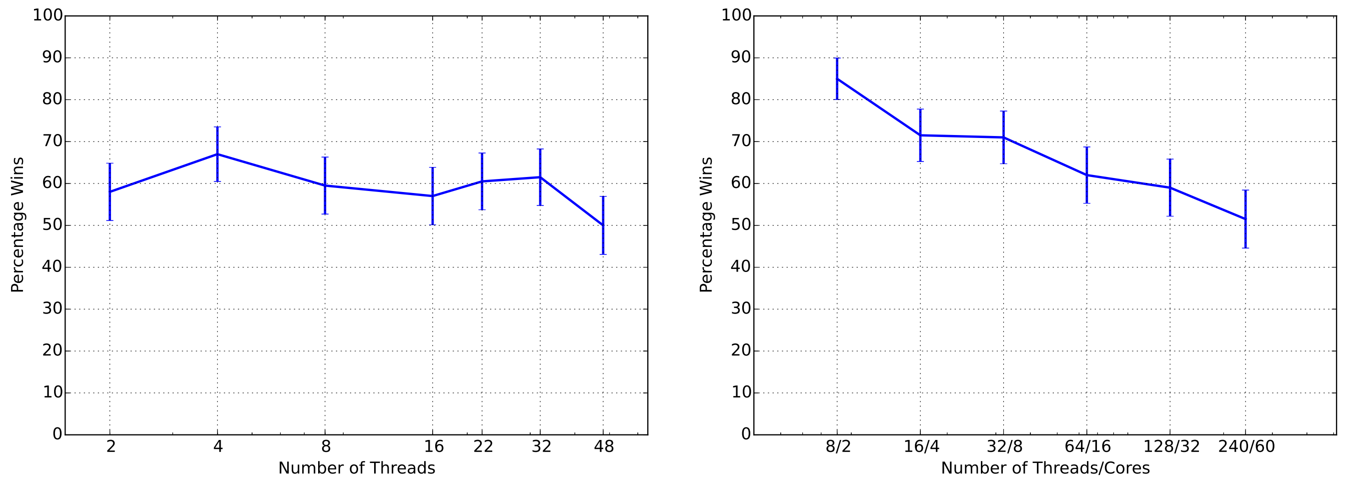
Interestingly, the strength speedup on Xeon Phi is quite different from Xeon CPU. The win rate for 8 threads is more than 80%. This is due to an insufficient number of playouts per second for 4 threads (the opponent player of the player with 8 threads), caused by the slow compute performance of Xeon Phi as described above. For 16 and 32 threads the win rate is consistent with perfect playout speedup (Fig. 6b). After 32 threads the decrease in strength speedup starts and continues to 240 threads.

2) Root Parallelization: In Fig. 7 the strength speed up for root parallelization on the Xeon CPU and Xeon Phi are shown. Again, the shape of the graphs are consistent with the playout speedup.

3) Section Conclusion: In both tree and root parallelization algorithms the differences between strength speedup graphs on Xeon CPU and Xeon Phi is due to an insufficient number of playouts per second on Xeon Phi compared to the Xeon CPU for each player.

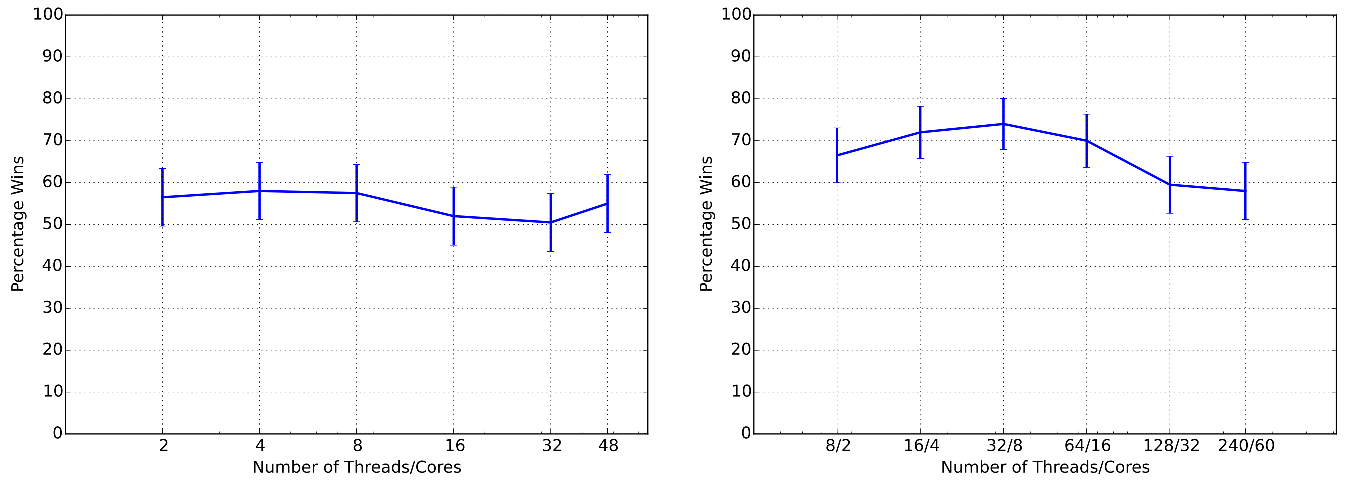
VI. Conclusion and Future Work

In many fields of research, randomized algorithms show a rather strong promise. One of these algorithms is MCTS, which has proven good solutions in Artificial Intelligence, Operations Research, and High Energy Physics [2]. It is an open question whether it is possible to parallelize MCTS efficiently on large scale machines [4]. Previous works have performed simulations or used either artificially generated trees [15], or complicated programs that precluded analysis



(a) (b)

Fig. 6: Strength speedup for tree parallelization. (a) Xeon CPU (b) Xeon Phi.



(a) (b)

Fig. 7: Strength speedup for root parallelization. (a) Xeon CPU (b) Xeon Phi.

and profiling [6]. This paper studies both the scalability and absolute performance of tree parallelization on Xeon CPU and Xeon Phi. We distinguish between playout speedup (no search overhead) and strength speedup (including search overhead).

With a specifically developed clean implementation of a real game-playing program we find that absolute performance of playout speedup on Xeon Phi is dominated by the time spent in the sequential part of the playout, and that current Xeon CPUs at 24 cores substantially outperform the Xeon Phi co-processor on 61 cores.

Finally, we may conclude that performance of parallelizations of the MCTS algorithm on Xeon Phi for games such as Hex or Go is limited by integer performance in sequential parts of the playout phase for which no vectorization method is known. However, since strength speedup follows playout speedup so closely, and since the scaling of the playout speedup continues upward up to 240 threads, we speculate that for a different problem in the other domains where the playout phase could be vectorized more promising results will be found. Our results warrant a more thorough analysis of playout speedup in the 64-240 thread region on Xeon Phi.

For the future work, we are working on a heterogeneous solution for executing parallel MCTS on both Xeon CPU and Xeon Phi similar to [22].

Acknowledgment

The authors would like to thank Jan Just Keijser for his helpful support with the Intel Xeon Phi co-processor. This work is supported in part by the ERC Advanced Grant no. 320651, HEPGAME.

References

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proceedings of the 5th International Conference on Computers and Games*, ser. CG'06. Berlin, Heidelberg: Springer-Verlag, May 2006, pp. 72–83.

[2] H. J. van den Herik, A. Plaat, J. Kuipers, and J. Vermaseren, "Connecting Sciences," *In 5th International Conference on Agents and Artificial Intelligence (ICAART 2013)*, vol. 1, pp. IS–7 – IS–16, 2013.

[3] J. Kuipers, A. Plaat, J. Vermaseren, and J. van den Herik, "Improving multivariate Horner schemes with Monte Carlo tree search," *Computer Physics Communications*, vol. 184, no. 11, pp. 2391–2395, Nov. 2013.

[4] G. Chaslot, M. Winands, and J. van den Herik, "Parallel monte-carlo tree search," *Computers and Games*, vol. 5131, pp. 60–71, 2008.

[5] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Sep. 2013.

[6] S. A. Mirsoleimani, A. Karami, and F. Khunjush, "A Two-Tier Design Space Exploration Algorithm to Construct a GPU Performance Predictor," in *Architecture of Computing Systems–ARCS 2014*. Springer, 2014, pp. 135–146.

[7] R. B. Segal, "On the Scalability of Parallel UCT," in *Proceedings of the 7th International Conference on Computers and Games*, ser. CG'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 36–47.

[8] P. Baudiš and J.-l. Gailly, "Pachi: State of the Art Open Source Go Program," in *Advances in Computer Games 13*, Nov. 2011.

[9] Y. Soejima, A. Kishimoto, and O. Watanabe, "Evaluating Root Parallelization in Go," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 278–287, Dec. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5654650

[10] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," *Advances in Computer Games*, vol. 6048, pp. 14–20, 2010.

[11] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, Dec. 2010.

[12] G. M. J. B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 03, pp. 343–357, 2008.

[13] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[14] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," *Machine Learning: ECML 2006*, 2006.

[15] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa, "Scalable Distributed Monte-Carlo Tree Search," *Fourth Annual Symposium on Combinatorial Search*, pp. 180–187, May 2011.

[16] Z. Galil and G. F. Italiano, "Data Structures and Algorithms for Disjoint Set Union Problems," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 319–344, Sep. 1991. [Online]. Available: http://doi.acm.org/10.1145/116873.116878

[17] J. Romein, A. Plaat, H. E. Bal, and J. Schaeffer, "Transposition Table Driven Work Scheduling in Distributed Search," in *In 16th National Conference on Artificial Intelligence (AAAI'99)*, 1999, pp. 725–731.

[18] S. Li, "Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework," 2013. [Online]. Available: https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise

[19] E. Heinz, "New self-play results in computer chess," in *Computers and Games*, 2001, pp. 262–276.

[20] J. Reinders, J. Jeffers, I. Meyerov, A. Sysoyev, N. Astafiev, and I. Burylov, *High Performance Parallelism Pearls*. Elsevier, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128021187000194

[21] Intel, "Intel Xeon Phi Product Family Highly parallel processing to power your breakthrough innovations," 2013. [Online]. Available: http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html

[22] S. A. Mirsoleimani, A. Karami, and F. Khunjush, "A parallel memetic algorithm on GPU to solve the task scheduling problem in heterogeneous environments," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO '13. New York, NY, USA: ACM, 2013, pp. 1181–1188. [Online]. Available: http://doi.acm.org/10.1145/2463372.2463518