Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

3

Thread-level Parallelization for MCTS

This chapter [1] addresses *RQ1* which is mentioned in Section 1.7.

- ***RQ1***: *What is the performance and scalability of thread-level parallelization for MCTS on both multi-core and many-core shared-memory machines?*

The recent successes of MCTS has led to even more investigations in closely related areas. Among them, considerable research has been put into improving the performance of parallel MCTS algorithms. Obviously, a high-performance parallelization in combination with additional computing power means that MCTS can investigate a larger part of the search space. As a direct consequence, MCTS *performance studies* (see Section 1.5) have become important in their own right [CWvdH08a, YKK+11, BCC+11, Seg11, SP14, SHM+16, SSS+17]. Besides the performance studies, there also exist *scalability studies* (see Section 1.5). A scalable parallelization means that performance of the algorithm scales on future architectures (e.g., a transition from multi-core to many-core). With respect to thread-level parallelization for MCTS we focus on both performance and scalability.

We do so on shared-memory machines for multi-core and many-core architectures. So far, only multi-core types of studies have been performed, and all of them were in

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, J. Vermaseren, and H. J. van den Herik, Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi, in Proceedings of the 2014 European Simulation and Modeling Conference (ESM 2014), 2014, pp. 88--94.
- S. A. Mirsoleimani, A. Plaat, H. J. van den Herik, and J. Vermaseren, Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors, in Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, vol. 3, pp. 77--83.

some sense limited. The two most important limitations were: (1) a limited number of cores on multi-core machines; and as a result of the first limitation, (2) the studies had to simulate a large number of the cores on a simulated environment instead of real hardware [Seg11]. In the first decade of this century, typically 8-24 core machines were used [CWvdH08a]. Recently, a scalability study of MCTS in AlphaGo has been performed with 40 threads on a 48 cores shared-memory machine [SHM+16]. The advent of the Intel® Xeon Phi™ in 2013 did allow to abandon both (a) a limited number of cores and the simulated environment and start (b) executing experiments in a real environment with a large number of cores. Indeed, the new development enabled us for the first time to study performance and scalability of the parallel MCTS algorithms on actual hardware, up to 244 parallel threads and 61 cores on shared-memory many-core machines. Hence, we designed an experimental setup with the above hardware and three benchmark programs.

In the first experiment (see Section 3.1), we executed operations related to matrix calculations using a micro-benchmark program on the Xeon Phi. The purpose of the first experiment was to measure the actual performance of the Xeon Phi and to understand the characteristics of its memory architecture. The results from this experiment were used as the input to execute the next two experiments.

In the second experiment (see Section 3.2), we ran the game of Go using the FUEGO program [EM10] that was also used in other studies [Seg11, SHM+16], on the Xeon CPU and for the first time on the Xeon Phi. FUEGO was one of the strongest open source programs in that time (2016--2017). It was based on a high performance C++ implementation of MCTS algorithms [SHM+16]. The purpose of the second experiment was to measure performance and scalability of FUEGO on both the Xeon CPU and the Xeon Phi. In this way, a direct comparison between our study on actual hardware with 244 parallel threads and other studies was possible.

In the third experiment (see Section 3.3), we carried out the game of Hex using our ParallelUCT program on both the Xeon CPU and the Xeon Phi. ParallelUCT is our highly optimized C++ library for parallel MCTS (see Section 2.6). The purpose of the third experiment was to measure performance and scalability of ParallelUCT on both the Xeon CPU and the Xeon Phi. In this way a direct comparison between our implementation of parallel MCTS and the FUEGO program was possible.

In the experiments, both FUEGO and ParallelUCT use thread-level parallelization for parallelizing MCTS. It is worth to mention that, even in all of the current parallelization approaches, the parallelism technique for implementing a parallel MCTS algorithm is thread-level parallelization [CWvdH08a, EM10, SKW10, SHM+16]. It means that multiple threads of execution which are equal to the number of available cores, are used. The advent of many-core machines, such as the Xeon Phi with many cores that are communicating through a complex interconnect network, did raise an

important question called *RQ1a*.

- **RQ1a**: *Can thread-level parallelization deliver a comparable performance and scalability for many-core machines compared to multi-core machines for parallel MCTS?*

The research goals of this chapter are twofold: (1) to investigate the performance and scalability of parallel MCTS algorithms on the Xeon CPU and the Xeon Phi (i.e., *RQ1*) and (2) to understand whether a comparable high-performance parallelization of the MCTS algorithm can be achieved on Xeon Phi using thread-level parallelization(i.e., *RQ1a*). We present and compare the results of the three experiments in the Section 3.1 to 3.3 to answer both research questions, *RQ1a* and *RQ1*. In Subsection 3.2.5 we answer *RQ1a* for FUEGO. In Subsection 3.3.5 we answer *RQ1a* for ParallelUCT. In Section 3.5 we answer *RQ1*. Our performance measures on which we will report are (A) the playout speedup and (B) the improvement of playing strength.

In summary, the chapter is organized as follows. In three sections, we provide answers to the research questions. Section 3.1 provides the performance of a micro-benchmark code for matrix calculations on the Xeon Phi. A study for the performance of FUEGO for the game of Go on a 9×9 board is presented in Section 3.2. Section 3.3 provides the performance of ParallelUCT for the game of Hex on an 11×11 board. Section 3.4 discusses related work. Finally, Section 3.5 contains our answer to *RQ1*.

3.1 Micro-benchmark Code Performance

The first experiment is about using a micro-benchmark code to measure the actual performance of Xeon Phi and to understand the characteristics of its memory architecture. We first provide an overview of Xeon Phi co-processor architecture in Subsection 3.1.1. Then, the experimental setup is discussed in Subsection 3.1.2, and it is followed by experiments in Subsection 3.1.3. We provide the results in Subsection 3.1.4 and conclude by presenting our findings in Subsection 3.1.5.

3.1.1 Xeon Phi Micro-architecture

A Xeon Phi co-processor board consists of up to 61 cores (of which 8 are shown in Figure 3.1a) based on the Intel 64-bit Instruction Set Architecture (ISA). Each of these cores contains Vector Processing Units (VPUs) to execute 512 bits. This means eight double-precision or 16 single-precision floating-point elements or 32-bit integers at the same time. The core also contains 4-way Simultaneous Multithreading (SMT), a dedicated L1 (it is not shown in the figure) and fully coherent L2 caches [Rah13]. The

Vector Processing Units (VPUs) are used to look up cache data distributed among the cores. The theoretical performance of the Xeon Phi card for double-precision floating-point operations is 1208 Giga Floating Point Operations per Second (GFLOPS). This is equal to 2416 GFLOPS for single-precision floating-point operations.

The connection between cores and other functional units such as a Memory Controller (MC) is through a bidirectional *ring interconnect*. There are eight distributed MCs as an interface between the ring burst and main memory (four MCs are shown in the figure). The main memory is up to 16 GB. To reduce hot-spot contention for data among the cores, a distributed Tag Directories (TD) is implemented so that every physical address that the co-processor can reach, is uniquely mapped through a reversible one-to-one address hashing function. This memory architecture provides a maximum transfer rate of 352 GB/s.

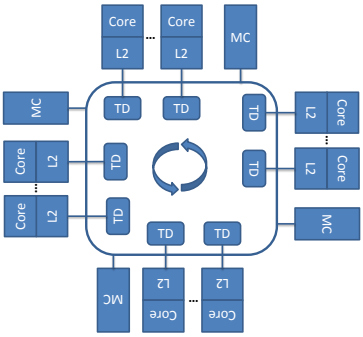
Thread affinity policies: On Xeon Phi, there are three predefined thread affinity policies for assigning threads to a core for obtaining improved or predictable performance [RVW+13]. These three policies are given in Table 3.1. A user can select one of the three policies for assigning threads or even none for assigning threads randomly. Figure 3.1b shows how each of the three thread affinity policies works for an exemplary case of eight threads and four cores. Thread affinity binds each thread to run on a specific subset of cores, to take advantage of memory locality. In the *compact* policy, the eight threads are bound to the first two cores, which means they are as close together as possible. The *scatter* policy distributes the eight threads as evenly as possible across the entire series of cores. *Scatter* is the opposite of *compact*. The *balanced* policy is between *compact* and *scatter*. The same set of rules applies when the number of threads is 244, and the number of cores is 61. However, we remark that when using the maximum number of threads (244 threads for 61 cores), the *compact* policy is equivalent to the *balanced* policy.

Definition 3.1 (Thread Affinity Policy) *A thread affinity policy is a bit vector mask in which each bit represents a logical processor that a thread is allowed to run on.*

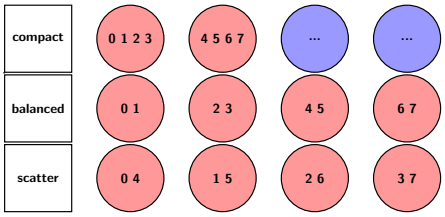
3.1.2 Experimental Setup

Below, we are using two micro-benchmark programs of two nested loops for doing Fused Multiply Add (FMA) operations to measure the maximum performance and memory bandwidth on the Xeon Phi. The primary target use for FMA is matrix operations [SBDD+02, JR13].

The first micro-benchmark code is the computation of 16 FMA vector operations (constitutes the inner loop) for ITR times (constitutes the outer loop). Listing A.1 shows the micro-benchmark program for measuring performance. The key line of the



(a) Abstract microarchitecture



(b) Thread-to-core assignment with three different thread affinity policies when mapping 8 threads to 4 cores.

Figure 3.1: Intel Xeon Phi Architecture.

Compact	It uses all four threads of a core before it begins using the threads of subsequent cores.
Balanced	It maps threads on different cores until all the cores have at least one thread, as done in the *scatter* policy. However, when multiple threads need to use the same core, the *balanced* policy ensures that threads with consecutive IDs are close to each other, in contrast to what is done by the *scatter* policy.
Scatter	It allocates the threads as evenly as possible over the whole processor such that consecutive threads are executed in different cores.

Table 3.1: Thread affinity policies

code in the inner loop is $c[j] = a[j] * b[j] + c[j]$. The outer loop is distributed among the available threads using OpenMP. For example, having 48 threads and $ITR = 48 * 10^6$ each of them executes $10^6 * 16$ operations. The inner loop is unrolled to optimize the program execution speed.

The second micro-benchmark code is performing $48 * 10^6$ three reads, and one write memory access pattern (constitutes the inner loop) for ITR times (constitutes the outer loop). Listing A.2 shows the micro-benchmark program for measuring bandwidth. The key line of the code in the inner loop is $c[j] = a[j] * b[j] + c[j]$. The inner loop is distributed among the available threads using OpenMP.

We measure the computation cost of arithmetic operations on different data formats (i.e., double-precision floating-point and integer) and provide the performances of the micro-benchmark code.

Definition 3.2 (Double-Precision Floating-Point Format) *The double-precision floating-point format is a computer number format, usually occupying 64 bits in computer memory.*

Definition 3.3 (Integer Format) *The integer format is a computer number format, consisting of 4 bytes.*

Henceforth, we will call the operations calculated on the double-precision floating-point format double-precision operations; likewise, we speak of integer operations.

3.1.3 Experimental Design

In our experiments, the benchmark code is compiled with the highest level of optimization (i.e., level three). The turbo mode is also on for the Xeon Phi. First, we set the thread affinity policy via the $KMP\_AFFINITY$ environment variable. Second, we set the number of threads via the $OMP\_NUM\_THREADS$ environment variable. Finally, we run the micro-benchmark code. We rerun the code while increasing the number of threads methodically from one to 244 for each of the three thread affinity policies (i.e., *compact*, *balanced*, and *scatter*).

3.1.4 Experimental Results

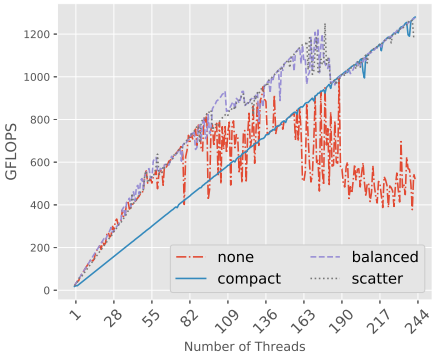
Figure 3.2, 3.3, and 3.4 show the results of our experiment. We will discuss the results for (A) double-precision operations and (B) integer operations.

A: Double-precision operations

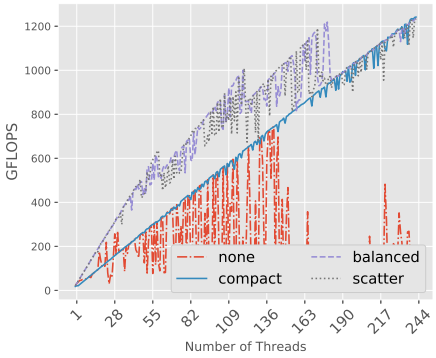
Below we discuss three issues: performance, scalability, and bandwidth.

Performance Figure 3.2 (a and b) shows the effect of three different thread affinity policies (*compact*, *balanced*, and *scatter*)[2] on the performance of the Xeon Phi for double-precision arithmetic operations for 244 data points, grouped into intervals of 27 data points. From the experiments we may provisionally conclude that using the *compact* policy, the maximum performance of $\sim$1200 GFLOPS is reached with 244 threads (see the blue line). Both the *balanced* (see the purple line) and *scatter* (see the gray line that is intermingled with the purple line) policies can reach the maximum performance of $\sim$1200 GFLOPS at 183 threads.

[2]In Figure 3.2 and the subsequent similar figures of this chapter, caption *none* (see the red line) means none of the three thread affinity policies is used.



(a) Number of iterations $960 * 10^6$.



(b) Number of iterations $48 * 10^6$.

Figure 3.2: Performance and scalability of double-precision operations for different numbers of iterations.

Scalability Figure 3.2b again shows the effect of three different thread affinity policies on the scalability of double-precision arithmetic operations on the Xeon Phi. We split 244 threads into four regions: (1) from 1 to 61 threads, (2) from 62 to 122 threads, (3) from 123 to 183 threads, and (4) from 184 to 244 threads. In the *compact* policy (see the blue line) the performance was steadily scaled until it reaches the maximum performance of ∼1200 GFLOPS at the end of the fourth region. The performance for both the *balanced* policy (see the purple line) and the *scatter* policy (see the gray line that is intermingled with the purple line) scales up to more than ∼600 GFLOPS at the end of the first region. By entering the second region, the performance suddenly drops to around 500 GFLOPS and starts increasing until it reaches ∼1000 GFLOPS at the end of the second region (i.e., 122 threads or 2 threads per core). The beginning of the third region (i.e., 123 threads) shows a drop in performance again, resulting in ∼800 GFLOPS. The third region is completed by a performance of ∼1200 GFLOPS. The very same pattern occurs in the fourth region, starting from ∼1000 GFLOPS for 184 threads and ending in more than ∼1200 GFLOPS for 244 threads.

Bandwidth Figure 3.3 shows the effect of thread affinity policies on the bandwidth of the Xeon Phi for executing the benchmark program in double-precision data types for 244 data points, grouped into intervals of 27 data points. In the *compact* policy (see the red line) the memory bandwidth is continuously increased until it reaches a plateau. The memory bandwidth graph has four regions in the *balanced* policy (see the blue line): (1) from 1 to 61 threads, (2) from 62 to 122 threads, (3) from 123 to 183 threads, and (4) from 184 to 244 threads. The maximum bandwidth of ∼180 GB

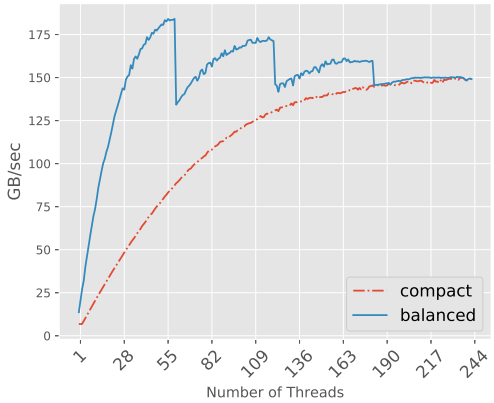


Figure 3.3: Memory bandwidth of double-precision operations on the Xeon Phi for increasing numbers of threads. Each interval contains 27 points.

per second (GB/sec) is reached for 61 threads. By using more threads, the bandwidth continuously decreased and never reached the same level as in the previous region. Therefore we may conclude that the memory bandwidth measurement shows that the maximum bandwidth is available for small numbers of threads (i.e., around 55) for the *balanced* policy.

B: Integer Operations

Below we discuss two issues: performance and scalability. We do not have a bandwidth graph for integer operations.

Performance Figure 3.4 shows the effect of four different thread affinity policies on the performance of the Xeon Phi for integer arithmetic operations. The first policy is *compact*. In the *compact* policy (see the blue line), the maximum performance of ∼1500 Giga Integers per Second (GIPS) is reached for around 244 threads. In the *balanced* and *scatter* policies depending on how many threads are assigned to each core, three different maxima for integer performances exist. As shown in Figure 3.4, for the both *balanced* and *scatter* policies, between 122 threads and 244 threads three peak points exist (i.e., 122, 183, and 244). At each of the peak points, a maximum performance of around 1500 GIPS is reached.

Scalability Figure 3.4 shows the effect of four different thread affinity policies on the scalability of the Xeon Phi for integer arithmetic operations. The first policy is

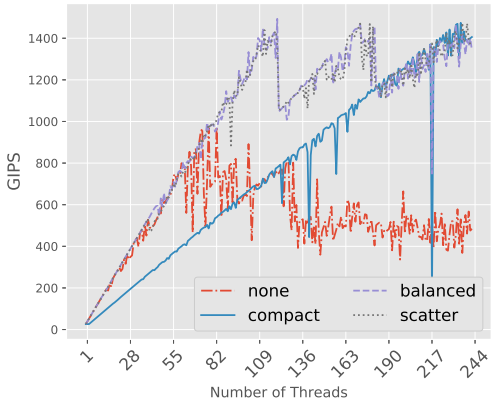


Figure 3.4: Performance and scalability of integer operations of the Xeon Phi for different numbers of threads.

compact. In the *compact* policy the performance was steadily increased (see the blue line). In the *balanced* and *scatter* policies depending on how many threads are assigned to each core, three different regions for integer performance exist. For example, as shown in Figure 3.4 between 122 threads and 183 threads some cores have two threads and some others have three threads in the *balanced* policy. The asymmetry in assigning threads to cores degraded the performance drastically at the beginning of the region (i.e., 123 threads) and later at the end of the region (i.e., 183), when thread assignment becomes more symmetric, performance started to increase.

3.1.5 Section Conclusion

We have performed micro-benchmarking on the Xeon Phi and found unexpected sensitivity of performance to thread affinity policies, which we attribute to a complex interconnect architecture. Although the theoretical performance for the Xeon Phi is reached, from the results of experiments we may conclude that the performance of a parallel program on the Xeon Phi is susceptible to the number of threads and the thread affinity policy.

3.2 FUEGO Performance and Scalability

The second experiment measures the performance and scalability of an open source library for parallel MCTS which is based on thread-level parallelization. FUEGO is an open source, tournament level Go-playing program, developed by a team at the Uni-

versity of Alberta [EM10]. It is a collection of C++ libraries for developing software for the game of Go and includes a Go player using MCTS. Using FUEGO would be a good benchmark for measuring the performance of the Xeon Phi because it has been used for similar scalability studies on CPUs [CWvdH08a, EM10, SHM+16]. It is essential to know what settings of the number of threads and the thread affinity policy will bring the best performance that an algorithm such as MCTS can reach when taking both computation time (i.e., for doing simulations) and memory bandwidth (i.e., for updating the search tree) into account.

Below we provide the experimental setup in Subsection 3.2.1. In Subsection 3.2.2 we explain the experiment. Then, the experimental results are discussed in Subsection 3.2.3. Subsection 3.2.4 provides our findings in this experiment.

3.2.1 Experimental Setup

To determine the performance and scalability of FUEGO on the Xeon Phi, we have performed a set of self-play experiments. The program with N threads plays as the first player against another instance of the same program but now with $N/2$ threads. It is a type of experiment that has been widely adopted for performance and scalability studies of MCTS [CWvdH08a, BG11]. We carry out the experiments on both the Xeon Phi co-processor and the Xeon CPU. Our results will allow a comparison between the two.

Performance Metrics

In our experiments, the performance of FUEGO is reported by (A) playout speedup (see Eq. 2.6) and (B) playing strength (see Eq. 2.7). We defined both metrics in Section 2.5. Here we operationalize the definitions. The scalability is the trend that we observe for these metrics when the number of resources (threads) is increasing.

3.2.2 Experimental Design

To generate statistically significant results in a reasonable amount of time most setups use the setting of 1 second per move, and so did we, initially. Appendix B provides details of the statistical analysis method which we used to analyze the result of a self-play tournament. The experiments were conducted with FUEGO SVN revision 1900, on a 9×9 board, with komi 6, Chinese rules, the alternating player color was enabled, the opening book was disabled. The win-rate of two opponents is measured by running at least a 100-game match. A single game of Go typically lasts around 81 moves. The games were played using the Gomill Python library for tournament play [Woo14]. Intel's *icc 14 .1* compiler is used to compile FUEGO in *native mode*. A

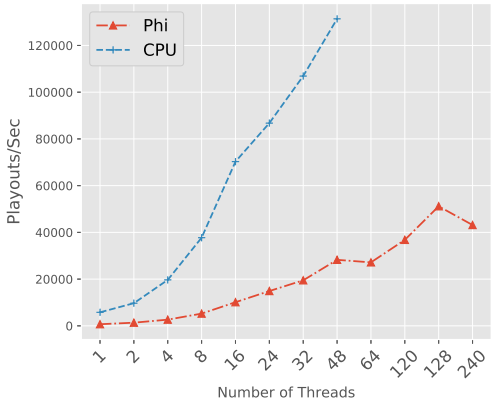


Figure 3.5: Performance and scalability of FUEGO in terms of PPS when it makes the second move. Average of 100 games for each data point. The board size is 9×9.

native application runs directly on the Xeon Phi and its embedded Linux operating system.

3.2.3 Experimental Results

This subsection reports on the performance of FUEGO by using two metrics: (A) playout speedup and (B) playing strength. The first metric corresponds to the improvement in the number of playouts or simulations per second (excluding search overhead), and the second metric corresponds to the improvement in the PW (including search overhead).

A: Playout Speedup

Figure 3.5 shows the performance and scalability of FUEGO on both the Xeon CPU (see the blue line) and the Xeon Phi (see the red line) in terms of PPS versus the number of threads. In the following, the results for the experiments on (A1) the multi-core Xeon CPU and (A2) the many-core Xeon Phi are discussed.

A1: Experiment on multi-core

Table 3.2 describes details of Figure 3.5 for the performance of FUEGO on the multi-core Xeon CPU. Although FUEGO does not show a linear speedup on the Xeon CPU it scales up to 48 threads. It reaches a speedup of 23 times for 48 threads on a 24 core machine.

# threads	1	8	16	32	48
count	100	100	100	100	100
mean	5788	37723	70286	106912	131378
std	241	2552	6078	9137	6008
min	4154	28246	40966	69129	97085
max	5979	40480	77210	121989	143630
speedup	1	7	12	18	23

Table 3.2: Performance of FUEGO on the Xeon CPU. Each column shows data for N threads. The board size is 9×9.

A2: Experiment on many-core

Figure 3.5 shows the PPS versus the number of threads for FUEGO for 12 data points where for each data point the number of threads is a power of 2 except for 24 and 48 threads that are selected to compare the Xeon Phi performance with the Xeon CPU. Moreover, 120 and 240 threads are chosen to find behavior of the curve around 128 threads. Figure 3.5 shows that even using 128 or more threads of the Xeon Phi cannot reach the performance of 16 threads on the Xeon CPU.

Table 3.3 describes details of Figure 3.5 for the performance of FUEGO on the Xeon Phi. The maximum speedup versus one core of the Xeon Phi is 74 times for 128 threads. The slow down from 128 threads to 240 threads shows that FUEGO cannot scale beyond 128 threads. The table also shows that FUEGO achieves only nine times speedup for 128 threads versus one core of the Xeon CPU. It should be noted that the number of PPS for eight threads on the Xeon Phi is equal to one thread on the Xeon CPU (see Table 3.3 where speedup versus CPU equals one for eight threads).

A3: Conclusion

In Paragraph A of Subsection 3.2.3, we reported on the performance and the scalability of FUEGO in terms of playout speedup. The maximum relative speedup on the multi-core Xeon CPU is 23, and on the many-core Xeon Phi it is 74. The thread-level parallelization method used by FUEGO scales up to 48 threads on the multi-core Xeon CPU and up to 128 threads on the many-core Xeon Phi. Due to the higher clock speed, the amount of work by each core of the Xeon CPU is much more than that by the Xeon Phi core. However, the difference in clock speed is only a factor of two, whereas the results show that the difference is more than a factor of 5 for 32 threads and more than 8 for one thread.

# threads	1	8	16	32	48	128	240
count	100	100	100	100	100	100	100
mean	694	5236	10112	19482	28251	51169	43149
std	14	67	128	255	387	810	2513
min	650	5055	9780	18721	27028	48930	39810
max	723	5361	10428	19976	29162	52957	64959
speedup	1	8	15	28	41	74	62
speedup vs CPU	-	1	2	3	5	9	7

Table 3.3: Performance of FUEGO on the Xeon Phi. Each column shows data for N threads. The board size is 9×9.

B: Playing Strength

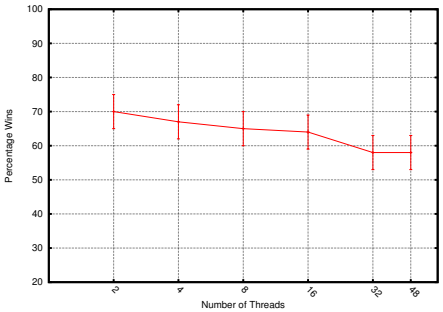
Figure 3.6 shows the scalability of FUEGO on both the Xeon CPU and the Xeon Phi in terms of the PW versus the number of threads. The graph shows the win-rate of the program with N threads as the first player. A straight line means that the program is scalable in terms of PW. In the following, the results for the experiments on (B1) the multi-core Xeon CPU and (B2) the many-core Xeon Phi are discussed.

B1: Experiment on multi-core

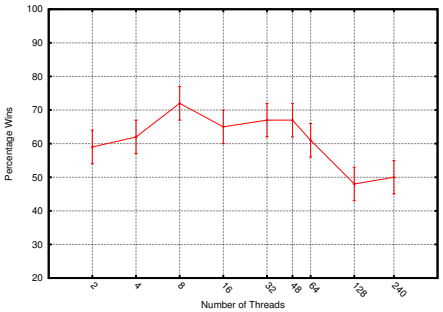
Figure 3.6a shows the results of the self-play experiments for FUEGO on the Xeon CPU. For the 9×9 board, the win-rate of the program with double the number of threads is better than the base program, starting at 70%, decreasing to 58% at 32 threads and then becomes flat. These results are entirely in line with results reported in [EMAS10] for 16 vs. eight threads. The phenomenon of search overhead explains the slightly decreasing lines.

B2: Experiment on many-core

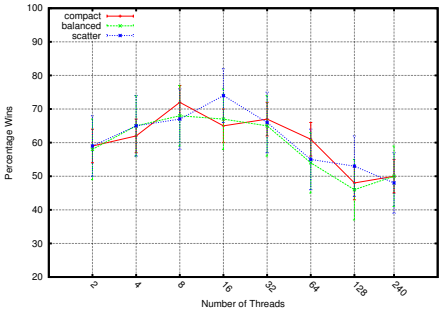
Figure 3.6b shows the performance and scalability of FUEGO in terms of PW on the many-core Xeon Phi. The scalability for the playing strength of FUEGO on the Xeon Phi differs notably from the Xeon CPU in Figure 3.6a. The Xeon CPU shows a smooth, slightly decreasing line. The Xeon Phi shows a more ragged line that first slopes up, and then slopes down. The maximum win-rate on the Xeon Phi is for eight threads (i.e., 72), while on the Xeon CPU it is for two threads (i.e., 70). The playing strength remains above the break-even point of 50% for the first player until 48 threads and then sharply decreases until 128 threads and becomes 50% for 240 threads. Up to 64 threads, these results confirm the simulation study by Segal [Seg11]. However, beyond 64 threads the performance drop is unexpectedly large. In the following two



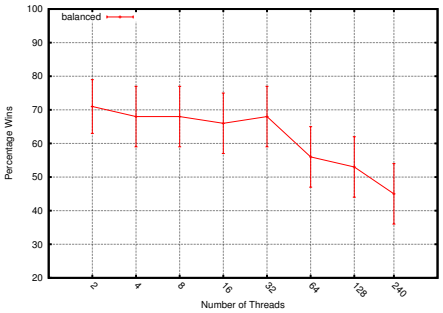
(a) Xeon CPU processor, with 200 games for each data point.



(b) Xeon Phi co-processor, with 300 games for each data point.



(c) Three thread affinity policies on the Xeon Phi, with 100 games for each data point.



(d) 10 second per move on the Xeon Phi, with 100 games for each data point.

Figure 3.6: Scalability of FUEGO in terms of PW with N threads against FUEGO with $N/2$ threads. The board size is 9×9.

paragraphs we report the results of our experiment on many cores (B2a) using different thread affinity policies and (B2b) increasing the time limit for making a move.

B2a: Different thread affinity policies

Figure 3.6c shows the effect of different thread affinity policies on the performance of FUEGO. For the FUEGO self-play experiments the *compact* affinity policy has been used. To show the effect of different thread affinity policies on FUEGO, the three different policies have been run. The PW for *balanced* policy shows more stability compared to the two other thread affinity policies. The best win-rate is for 4 threads (1 core) in the *compact* policy and for 16 threads (16 cores) in the *scatter* policy.

B2b: Increasing time limit

Figure 3.6d shows the results when FUEGO can make a move with 10 seconds for doing a simulation on the Xeon Phi. The low PPS numbers of the Xeon Phi suggest inefficiencies due to the small problem size. Closer inspection of the results on which Figure 3.5 is based suggests that FUEGO is not able to perform sufficient simulations on the Xeon Phi for a small number of threads in just 1 second. Therefore, we increased the time limit per move to 10 seconds. We see that now the graph is approaching that of the Xeon CPU. The win-rate behavior for the low number of threads is now much closer to that of the CPU (Figure 3.6b), and the counter-intuitive hump-shape has changed to the familiar down-sloping trend. However, we still see a fluctuation in the *balanced* policy. Up to 32 threads, the performance is still reasonable (close to 70% win-rate for the $2\times$ thread program), but up to 240 threads the performance deteriorates. The maximum win-rate is for eight threads, and there is still a marginal benefit for using 128 threads.

B3: Conclusion

In Paragraph B of the Subsection 3.2.3, we reported the performance of FUEGO in terms of PW. The maximum PW on the multi-core Xeon CPU is around 70 for two threads and on the many-core Xeon Phi it is around 72 for eight threads. The thread-level parallelization method used by FUEGO does not scale very well on both the multi-core Xeon CPU and on the many-core Xeon Phi. For a time limit equal to 10 seconds per move, FUEGO scales only up to 32 threads on the many-core Xeon Phi.

3.2.4 Section Conclusion

We have carried out, to the best of our knowledge, the first performance and scalability study of a strong open source program for playing Go using MCTS called FUEGO on the Xeon Phi. Previous work only targeted scalability on a CPU [SKW10, BG11, SHM$^+$16] or used simulation [Seg11]. Our experiments showed the difference in performance of an identical program in an identical setup on the Xeon CPU versus the Xeon Phi using the standard experimental settings of the 9×9 board and 1 second per move. We found (1) a good performance up to 32 threads, confirming a previous simulation study and (2) a deteriorating performance from 32 to 240 threads (see Figure 3.6).

3.2.5 Answer to RQ1a for FUEGO

In this subsection we answer *RQ1a* for FUEGO. We repeat *RQ1a* below.

- **RQ1a**: *Can thread-level parallelization deliver a comparable performance and scalability for many-core machines compared to multi-core machines for parallel MCTS?*

Using FUEGO, which uses thread-level parallelization for implementing the Tree Parallelization algorithm, we have found in Subsection 3.1.3 that we cannot reach the same performance on the Xeon Phi as on the Xeon CPU. The maximum performance in terms of PPS for Tree Parallelization on the Xeon CPU is around three times more than the one on the Xeon Phi (see Figure 3.5). Moreover, the scalability of the Tree Parallelization algorithm in terms of PPS is better on the Xeon CPU (for up to 32 threads) than the Xeon Phi (for up to 240 threads). Our experiments show that the performance of the algorithm drops after 128 threads on the Xeon Phi (see Figure 3.5). For the performance in terms of PW, the Xeon CPU shows a steadily decreasing PW (see Figure 3.6a), as expected, where the Xeon Phi shows a hump-like shape (see Figure 3.6b). Hence, our answer to *RQ1a* reads: with thread-level parallelization we cannot reach the same performance of a multi-core machine on a many-core machine.

3.3 ParallelUCT Performance and Scalability

The third experiment is using the ParallelUCT library (see Section 2.6). The open source MCTS libraries of FUEGO add additional ideas to the simple MCTS algorithm to improve gameplay. In contrast, the ParallelUCT is solely developed to focus on MCTS as a general algorithm not only for games but for general optimization problems. ParallelUCT is our highly optimized parallel C++ library for MCTS. It is developed to use thread-level parallelization to parallelize MCTS. Therefore, it is chosen for this study. We provide the experimental setup in Subsection 3.3.1. Then, in Subsection 3.3.2 we explain the experiment. The experimental results are discussed in Subsection 3.3.3. Finally, Subsection 3.3.4 provides our findings of this experiment.

3.3.1 Experimental Setup

In order to generate statistically significant results for the game of Hex (board size 11×11) in a reasonable amount of time, both players do playouts of 1 second for choosing a move. To calculate the playing strength 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 [Hei01] and similar to [MKK14] 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. The parameter C_p is set at 1 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 PPS 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 the Xeon Phi [RJM+15].

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 14.1* 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 [Int13]. Intel's *icc 14.1* compiler is used to compile the program in *native mode*. A *native application* runs directly on the Xeon Phi and its embedded Linux operating system.

Performance Metrics

In our experiments, the performance of ParallelUCT is reported by (A) playout speedup (see Eq. 2.6) and (B) playing strength (see Eq. 2.7). We defined both metrics in Section 2.5. Here we operationalize the definitions. The scalability is the trend that we observe for these metrics when the number of resources (threads) is increasing.

3.3.2 Experimental Design

In all of our experiments, we perform self-play Hex games in a tournament to measure performance and scalability. Each tournament 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 to make a move.

3.3.3 Experimental Results

The performance of the algorithms is reported by (A) playout speedup and (B) playing strength.

A: Playout Speedup

Figure 3.7 shows the performance and scalability of (A1) Tree Parallelization and (A2) Root Parallelization on both the multi-core Xeon CPU and the many-core Xeon Phi in terms of PPS versus the number of threads.

A1: Tree Parallelization

In Figure 3.7 the scalability of Tree Parallelization on the Xeon CPU and the Xeon Phi are compared. In the following the results for the experiments on (A1a) the multi-core Xeon CPU and (A1b) the many-core Xeon Phi are discussed.

A1a: Experiment on multi-core

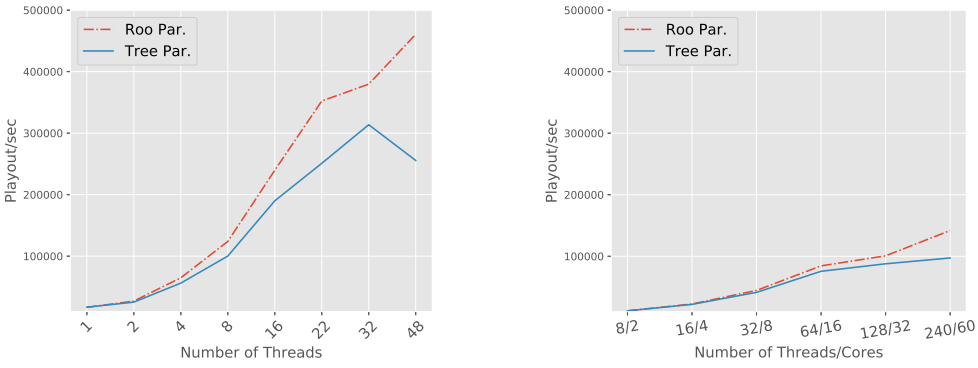
Figure 3.7a shows playout speedup on the Xeon CPU. We see a perfect playout speedup up to 4 threads and a near perfect speedup up to 16 threads. The increase in the number of playouts continues up to 32 threads, although the increase is no longer perfect. There is a sharp decrease in the number of playouts for 48 threads. The available number of cores on the Xeon CPU is 24 cores, with two 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 four threads, but the effect is not severe. The conclusion here is that our results are different from the results in [CWvdH08a] and [AHH10] where the authors reported no speedup beyond four threads for locked Tree Parallelization.

A1b: Experiment on many-core

In Figure 3.7b the playout speedup on the Xeon Phi is shown. A perfect playout speedup is observed up to 64 threads. We see that using a lock does not affect the performance of the algorithm up to this point. After 64 threads the performance drops, although the number of PPS 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 the Xeon Phi still, the number of PPS is less than on eight threads on the Xeon CPU. Our provisional conclusion here is that the performance for Tree Parallelization on the Xeon Phi is almost 30% of the peak performance on the Xeon CPU.

A2: 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



(a) Xeon CPU

(b) Xeon Phi

Figure 3.7: Performance and scalability of ParallelUCT in terms of PPS for both Tree and Root Parallelization.

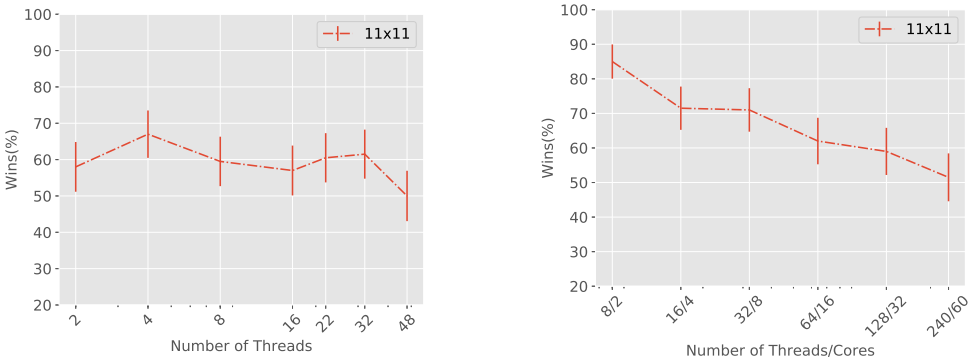
see whether the decrease in playout speedup in Tree Parallelization is due to locks or not. In Figure 3.7 the scalability of Root Parallelization on the Xeon CPU and the Xeon Phi are compared. In the following the results for the experiments on (A2a) the multi-core Xeon CPU and (A2b) the many-core Xeon Phi are discussed.

A2a: Experiment on multi-core

As is shown in Figure 3.7a for the 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 the Xeon CPU.

A2b: Experiment on many-core

The performance of Root Parallelization on the Xeon Phi is shown in Figure 3.7b. Here, we require at least eight threads on the Xeon Phi to reach almost the same number of PPS as one thread on the Xeon CPU. On the 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 the Xeon Phi is almost 30% of its counterpart on the Xeon CPU. To understand the reason



(a) Xeon CPU

(b) Xeon Phi

Figure 3.8: Scalability of ParallelUCT in terms of PW for Tree Parallelization.

for this low performance we did a detailed timing analysis to find out where most of the time of the algorithm has been spent in the selection, expansion, playout, or backup phase. For the Hex board size of 11×11, 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 (9×9) and Hex (9×9) because they have different rules; 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 the playout phase.

A3: Conclusion

In both Tree and Root Parallelization algorithms, the performance of the parallel algorithm is less on the Xeon Phi compared to the Xeon CPU. Comparing the differences between scalability graphs of both algorithms in terms of PPS on both the Xeon CPU and the Xeon Phi shows the limited scalability of Tree Parallelization when using more threads compared to Root Parallelization. Here we may conclude that the performance of thread-level parallelization for both Tree and Root Parallelization algorithms on the Xeon CPU is better than the one on the Xeon Phi (see *RQ1a*). In terms of scalability, the thread-level parallelization for the Root Parallelization algorithm shows better scalability comparing to the Tree Parallelization algorithm on both the Xeon CPU and the Xeon Phi.

B: Playing Strength

Figure 3.8 shows the performance and scalability of Tree Parallelization on both the Xeon CPU and the Xeon Phi in terms of the PW versus the number of threads. Figure

3.9 shows the scalability of Root Parallelization on both the Xeon CPU and the Xeon Phi in terms of the PW versus the number of threads. The graph shows the win-rate of the program with N threads as the first player. A straight line means that the program is scalable in terms of the playing strength.

B1: Tree Parallelization

As already mentioned, it is also essential 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 PPS reflects in a more dominant player. In the following the results for the experiments on (B1a) the multi-core Xeon CPU and (B1b) the many-core Xeon Phi are discussed.

B1a: Experiment on multi-core

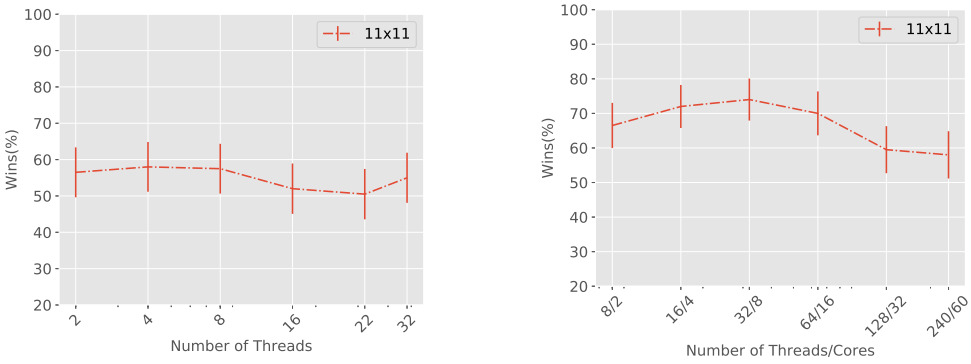
In Figure 3.8a playing strength for Tree Parallelization on the Xeon CPU is shown. Note that, since we compare the performance of N threads against $N/2$ threads, an ideal perfect playing strength would give a straight, horizontal line of, say, 60% win rate for the player with more threads. We see good playing strength 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 PPS for 48 threads in Figure 3.7a. Our provisional conclusion here is that on the Xeon CPU, the playing strength follows playout speedup closely.

B1b: Experiment on many-core

Interestingly, the playing strength on the Xeon Phi is entirely different from that on the Xeon CPU. The win rate for eight threads is more than 80%. This is due to an insufficient number of PPS for four threads (the opponent player of the player with eight threads), caused by the slow computing performance of the Xeon Phi as described above. Our provisional conclusion is that for 16 and 32 threads the win rate is consistent with perfect playout speedup (Figure 3.8b). After 32 threads the decrease in strength speedup starts and continues to 240 threads.

B2: Root Parallelization

In Figure 3.9 the scalability in terms of PW for Root Parallelization on the Xeon CPU and the Xeon Phi are shown. In the following the results for the experiments on (B2a) the multi-core Xeon CPU and (B2b) the many-core Xeon Phi are discussed.



(a) Xeon CPU

(b) Xeon Phi

Figure 3.9: Scalability of ParallelUCT in terms of PW for Root Parallelization.

B2a: Experiment on multi-core

In Figure 3.9a the scalability in terms of PW for Root Parallelization on the multi-core Xeon CPU is shown. The shape of the scalability graph shows that Root Parallelization does not scale beyond 8 threads in spite of good scalability in terms of the number of PPS on the Xeon CPU (see Figure 3.7a).

B2b: Experiment on many-core

In Figure 3.9a the scalability in terms of PW for Root Parallelization on the many-core Xeon Phi is shown. The shape of the scalability graph shows that Root Parallelization scales up to 32 threads no beyond that in spite of good scalability in terms of the number of PPS on the Xeon Phi (see Figure 3.7b).

B3: Conclusion

In both Tree and Root Parallelization algorithms, the differences between scalability graphs in terms of PW on the Xeon CPU and the Xeon Phi is due to an insufficient number of PPS on the Xeon Phi compared to the Xeon CPU.

3.3.4 Section Conclusions

We have performed an in-depth scalability study of both Tree and Root Parallelizations of the MCTS algorithm on the Xeon CPU and the Xeon Phi for the game of Hex. It is the first large-scale (up to 240 threads and 61 cores) study of Tree Parallelization on a real shared-memory many-core machine. Contrary to previous results [EM10], we

show that the effect of using data locks is not a limiting factor on the performance of a Tree Parallelization for 16 threads on the Xeon CPU and 64 threads on the Xeon Phi.

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 step, expansion step, playout step, or update step. For the Hex board size of 11×11, 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; 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 the playout phase.

Since the playout phase dominates execution time of each thread, the Xeon CPU outperforms the 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 the Xeon Phi because each core of the Xeon CPU is more powerful than each core of the 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 the 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 those mentioned in [vdHPKV13].

3.3.5 Answer to RQ1a for ParallelUCT

Using our ParallelUCT package, which uses thread-level parallelization for implementing two parallel MCTS algorithms (i.e., Root Parallelization and Tree Parallelization), we cannot reach the same performance on the Xeon Phi as on the Xeon CPU (see Figure 3.7). The maximum performance in terms of PPS for both Root Parallelization and Tree Parallelization on the Xeon CPU is around three times more than the one on the Xeon Phi. The Root Parallelization algorithm scalability in terms of PPS on both the Xeon Phi and the Xeon CPU are similar. The Tree Parallelization algorithm scalability in terms of PPS is better on the Xeon Phi than on the Xeon CPU, since the performance of the algorithm is dropped after 32 threads on the Xeon CPU. We find that three obstacles limit performance and scalability on both the Xeon CPU and the Xeon Phi: (1) the time spent in the sequential part of the algorithm, (2) the thread management overhead, and (3) the synchronization overhead due to using locks to protect the shared search tree.

3.4 Related Work

Below we review related work on MCTS parallelizations. The two major parallelization methods for MCTS are Tree Parallelization and Root parallelization [CWvdH08a]. There are also other techniques such as leaf parallelization [CWvdH08a] and approaches based on transposition table driven work scheduling [YKK+11, RPBS99].

- Tree Parallelization: For shared-memory machines, Tree Parallelization is a suitable method. It is used in FUEGO, an open source Go program. It is shown in [CWvdH08a] 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. Moreover, it is shown in [EM10] that a lock-free implementation of this algorithm provides better scaling than a locked approach. In [EM10] such a lock-free Tree Parallelization for MCTS is proposed. The authors intentionally ignored rare faulty updates inside the tree and studied the scalability of the algorithm for up to 8 threads. In [BG11], the performance of a lock-free Tree Parallelization for up to 22 threads is reported. The playing strength is perfect for 16 threads, but the improvement drops for 22 threads. There is also a case study that shows good performance of a (no-MCTS) Monte Carlo simulation on the Xeon Phi co-processor [Li13]. Segal's [Seg11] simulation study of Tree Parallelization on an ideal shared-memory system suggested that perfect playing strength beyond 64 threads may not be possible, presumably due to increased search overhead. Baudiš et al. reported almost near perfect playing strength up to 22 threads for a lock-free tree parallelization [BG11].

- Root Parallelization: Chaslot et al. [CWvdH08a] reported results that Root Parallelization shows perfect playout speedup for up to 16 threads. Soejima et al. [SKW10] 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.

3.5 Answer to RQ1

In this chapter we presented the thread-level parallelization for parallelization of MCTS. We have already answered *RQ1a* in Subsection 3.2.5 and Subsection 3.3.5. This section proposes an answer for *RQ1*.

- ***RQ1***:*What is the performance and scalability of thread-level parallelization for MCTS on both multi-core and many-core shared-memory machines?*

For thread-level parallelization, our study shows that the performance of MCTS on the many-core Xeon Phi co-processor with its MIC architecture is less than its performance on the NUMA-based multi-core processor (see Subsections 3.2.3 and 3.3.3). The results show that current Xeon CPUs at 24 cores substantially outperform the Xeon Phi co-processor on 61 cores. Our study also shows that the scalability of thread-level parallelization for MCTS on the many-core Xeon Phi co-processor is limited.