Cover Page

The handle http://hdl.handle.net/1887/119358 holds various files of this Leiden University dissertation.

Author: Mirsoleimani, S.A.
Title: Structured parallel programming for Monte Carlo tree search
Issue Date: 2020-06-17

4 Task-level Parallelization for MCTS

This chapter addresses *RQ*2 which is mentioned in Section 1.7.

- ***RQ2***: *What is the performance and scalability of task-level parallelization for MCTS on both multi-core and many-core shared-memory machines?*

In this chapter[1], we investigate how to parallelize irregular and unbalanced tasks efficiently on the Xeon Phi using MCTS. MCTS performs a search process based on a large number of random samples in the search space. The nature of each sample in MCTS implies that the algorithm is considered as a good target for parallelization. Much of the effort to parallelize MCTS has focused on using parallel threads to do tree-traversal in parallel along separate paths in the search tree [CWvdH08a, YKK$^+$11]. Using software threads makes it difficult to deal with irregular parallelism because creating too many threads for load balancing and better utilization of processors would cause a problem regarding thread creation overhead and memory usage. It is often hard to find sufficient parallelism in an application when there is a large number of cores available as in the Xeon Phi. To find adequate parallelism, we need to adapt MCTS to use logical parallelism, called tasks. Therefore, we use tasks due to their increased machine independence, safety, and scalability over threads. Below we present a task parallelism approach for MCTS which allows controlling the granularity (or grain size) of a task.

Three main contributions of this chapter are as follows.

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, H. J. van den Herik, and J. Vermaseren, Scaling Monte Carlo Tree Search on Intel Xeon Phi, in Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2015, pp. 666--673.

1. A first detailed analysis of the performance of three widely-used threading libraries on a highly optimized program with high levels of irregular and unbalanced tasks on the Xeon Phi is provided.
2. A straightforward First In, First Out (FIFO) scheduling policy is shown to be equal or even to outperform the more elaborate threading libraries Cilk Plus and Threading Building Blocks (TBB) for running high levels of parallelism for high numbers of cores. This is surprising since Cilk Plus was designed to achieve high efficiency for precisely these types of applications.
3. The first parallel MCTS with grain size control is proposed. It achieves, to the best of our knowledge, the fastest implementation of a parallel MCTS on the 61-core Xeon Phi 7120P (using a real application) with a 47 times speedup compared to sequential execution on the Xeon Phi itself (which translates to 5.6 times speedup compared to the sequential version on the regular host CPU, Xeon E5-2596).

The rest of this chapter is organized as follows. Section 4.1 describes the complications for irregular parallelization of MCTS. Section 4.2 describes how to achieve task-level parallelization. Section 4.3 explains the threading libraries. Section 4.4 describes the Grain Size Controlled Parallel MCTS algorithm. Implementation considerations are given in Section 4.5. A scalability study for the proposed algorithm is presented in Section 4.6. Section 4.7 provides the experimental setup, Section 4.8 describes the experimental design, and Section 4.9 gives the experimental results (with five possible implementations of GSCPM). Section 4.10 presents our analysis of results. Finally, Section 4.11 discusses related work.

4.1 Irregular Parallelism Challenge

One of the obstacles for parallelizing MCTS is parallel execution of iterations with an asymmetric search tree, resulting in *irregular parallelism* (see Subsection 1.4.1). We aim to address this challenge using task-level parallelization with a task scheduler that ensures a maximum concurrency level with minimum load balancing overhead (see Section 4.2 to Section 4.4).

4.2 Achieving Task-level Parallelization

Reaching task-level parallelization for the MCTS loop depends on a precise arrangement of the iterations into tasks (Subsection 4.2.1). It also requires us to understand data dependencies between different iterations of the loop (Subsection 4.2.2).

4.2.1 Decomposition of Iterations into Tasks

The MCTS loop which is based on *iteration* control flow pattern loop is the best first place to look to create parallel tasks because considering every iteration of the MCTS loop to be a task can often keep a large number of threads active. To create tasks, we use a parallel pattern, namely the *fork-join pattern*.

Definition 4.1 (Iteration Pattern) *In the iteration pattern, a condition c is evaluated. If it is true, a task a is evaluated, then the condition c is evaluated again, and the process repeats until the condition becomes false [MRR12].*

Definition 4.2 (Fork-join Pattern) *A pattern of computation in which new (potential) parallel flows of control are created/split with **forks** and terminated/merged with **joins**.*

4.2.2 Ignoring Data Dependencies among Iterations

The body of the MCTS loop depends on previous invocations of itself. The source of this dependency comes from the fact that the results of computation associated with each iteration update a single search tree. The constructed search tree guides the next iterations of search towards a possibly existing global minimum of the search space avoiding local minima. This type of dependencies is called Iteration-Level Dependencies (ILDs). We can ignore this type of dependency.

4.3 Threading Libraries

In this section, we discuss two threading libraries which allow task-level parallelization of loops. Some parallel programming models provide programmers with thread pools, relieving them of the need to manage their parallel tasks explicitly [LP98, Rob13]. Creating threads each time that a program needs them can be undesirable. To prevent overhead, the program has to do two things: (1) managing the lifetime of the thread objects, and (2) determining the number of threads appropriate to the problem and the current hardware. The ideal scenario would be that the program could just (1) divide the code into the smallest logical pieces that can be executed concurrently (called tasks), and (2) pass them over to the compiler and library, to parallelize them. This approach uses the fact that the majority of threading libraries does not destroy the threads once created so that they can be resumed much more quickly in subsequent use. This is known as creating a *thread pool*.

A thread pool is a group of shared threads [NBF96]. Tasks that can be executed concurrently are submitted to the pool and are added to a queue of pending work. Each task is then taken from the queue by one of the worker threads that execute the

task before looping back to take another task from the queue. The user specifies the number of worker threads.

Thread pools use either a work-stealing or a work-sharing scheduling method to balance the workload. Examples of parallel programming models with work-stealing scheduling are TBB and Cilk Plus [Rei07]. Below we discuss these two threading libraries: Cilk Plus in Subsection 4.3.1 and TBB in Subsection 4.3.2.

4.3.1 Cilk Plus

Cilk Plus is an extension to C and C++ designed to offer a quick and easy way to harness the power of both multi-core and vector processing. Cilk Plus is based on MIT's research on Cilk [BJK+95]. Cilk Plus provides a simple yet powerful model for parallel programming, while runtime and template libraries offer a well-tuned environment for building parallel applications [Rob13].

The function calls in an MCTS can be tagged with the first keyword *cilk_spawn*, which indicates that the function can be executed concurrently. The calling function uses the second keyword *cilk_sync* to wait for the completion of all the functions it spawned. The third keyword is *cilk_for*, which converts a serial for loop (e.g., the main loop of MCTS) into a parallel for loop. The runtime system executes the tasks within a provably efficient work-stealing framework. Cilk Plus uses a double-ended queue per thread to keep track of the tasks to perform and uses it as a stack during regular operations conserving a sequential semantic. When a thread runs out of tasks, it steals the most in-depth half of the stack of another (randomly selected) thread [LP98, Rob13]. In Cilk Plus, thief threads steal *continuations*.

4.3.2 Threading Building Blocks

Threading Building Blocks (TBB) is a C++ template library developed by Intel for writing software programs that take advantage of a multi-core processor [Rei07]. TBB implements work-stealing to balance a parallel workload across available processing cores to increase core utilization and therefore, to scale. The TBB work-stealing model is similar to the work-stealing model applied in Cilk, although in TBB, thief threads steal *children* [Rei07].

4.4 Grain Size Controlled Parallel MCTS

This section discusses the Grain Size Controlled Parallel MCTS (GSCPM) algorithm. The pseudo-code for GSCPM is shown in Algorithm 4.1. In the MCTS loop (see Algorithm 2.1 and Algorithm 2.2), the computation associated with each iteration is

independent. Therefore, *these are candidates* to guide a task decomposition by mapping a chunk of iterations onto a task for parallel execution on separate processors. This type of task is called Iteration-Level Task (ILT) and this type of parallelism is called Iteration-Level Parallelism (ILP) [CWvdH08a, SP14, MPvdHV15a].

Definition 4.3 (Iteration-level Task) *The iteration-level task is a type of task that contains a chunk of MCTS iterations.*

Definition 4.4 (Iteration-level Parallelism) *Iteration-level parallelism is a type of parallelism that enables task-level parallelization to assign a chunk of MCTS iterations as a separate task for execution on separate processors.*

The MCTS loop can be implemented with two different loop constructs (i.e., *while* and *for*). If we cannot predict how many iterations will take place (e.g., the search continues until a goal value has been found), then this is a *while* loop. In contrast, if the number of iterations is known in advance, then this can be implemented in the form of a *for* loop. The GSCPM algorithm is designed for the modern threading libraries. For many threading libraries, it is necessary for parallelizing loops to know the total number of iterations in advance. Therefore, the *outer* loop in GSCPM is a counting *for* loop (see Line 5 in Algorithm 4.1) which iterates as many times as the number of available tasks (*nTasks*). Then, the search budget (*nPlayouts*) can be divided into chunks of iterations to be executed by an *inner* serial loop (see Line 7 in Algorithm 4.1 and details of the UCTSEARCH function in Algorithm 2.2). A *chunk* is a sequential collection of one or more iterations. The maximum size of a chunk is called *grain size*. Therefore, the grain size is the number of playouts divided by the number of tasks (*nPlayouts/nTasks*) and it could be as small as one iteration or as large as the total number of iterations. Controlling the number of tasks (*nTasks*) allows to control the grain size in GSCPM. The design of GSCPM is based on *fork-join* parallelism. The *outer* loop forks instances of the *inner* loop as tasks (see Line 7) and the runtime scheduler allocates the tasks to threads for execution. With this technique, we can create more tasks than threads. This is called fine-grained task-level parallelism.

Definition 4.5 (Fork-join Parallelism) *In fork-join parallelism, control flow splits into multiple flows that combine later.*

By increasing the number of tasks, the grain size is reduced, and we provide more parallelism to the threading library [RJ14]. Finding the right balance is the key to achieve proper scaling. The grain size should not be too small because then spawn overhead reduces performance. It also should not be too large because that reduces parallelism and load balancing (see Table 4.1).

Table 4.1: The conceptual effect of grain size.

Large grain size (nTasks $\ll$ nCores)	Speedup bounded by tasks (not sufficient parallelism)
Right grain size	Good speedup
Small grain size (nTasks $\gg$ nCores)	Spawn and scheduling overhead (reduces performance)

Algorithm 4.1: The pseudo-code of the GSCPM algorithm.

```
1  Function GSCPM(State s_0, nPlayouts, nTasks)
2      v_0 := create a shared root node with state s_0;
3      grain_size := nPlayouts/nTasks;
4      t := 1;
5      for t ≤ nTasks do
6          s_t := s_0;
7          fork UCTSEARCH(v_0, s_t, grain_size) as task t;
8          t := t+1;
9      wait for all tasks to be completed;
10     return action a of best child of v_0;
```

4.5 Implementation Considerations

The performance of a shared search tree in combination with random number generation is significant for the overall performance of the GSCPM algorithm. Therefore, we explain our choices for implementing the shared search tree in Subsection 4.5.1, and for the random number generator in Subsection 4.5.2. For more details on the implementation of GSCPM, see Appendix C.

4.5.1 Shared Search Tree Using Locks

In a serial MCTS loop, the manipulations are called in order manipulations. Therefore, the data inside the tree remains valid during the full execution. However, parallelization of the loop causes out of order manipulations, which may cause data corruption. In our implementation of tree parallelism, a lock is used in the expansion phase of the MCTS algorithm to avoid (1) the loss of any information and (2) corruption of the tree data structure [EM10]. To allocate all children of a given node, a pre-allocated vector of children is used. When a thread tries to append a new child to a node, it increments an atomic integer variable as the index to the next possible child in the vector of children. The values of $w_j = Q(v_j)$ and $n_j = N(v_j)$ are also defined to be atomic integers (see Algorithm 2.2).

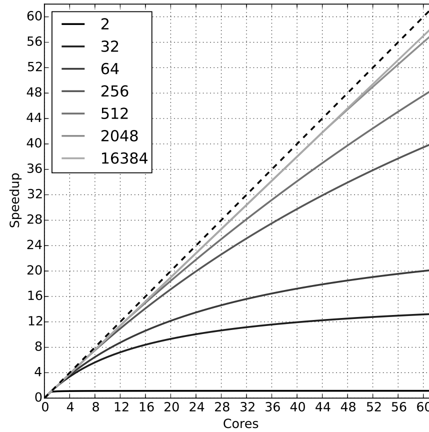


Figure 4.1: The scalability profile produced by Cilkview for the GSCPM algorithm. The number of tasks is shown. Higher is more fine-grained.

4.5.2 Random Number Generator

The efficiency of Random Number Generation (RNG) is a crucial performance aspect of any Monte Carlo simulation [Li13]. In our implementation, the highly optimized Intel MKL is used to generate a separate RNG stream for each task with a single seed. One MKL RNG interface API call can deliver an arbitrary number of random numbers. In our program, a maximum of 64 K random numbers is provided in one call [WZS+14]. A thread generates the required number of random numbers for each task.

4.6 Performance and Scalability Study

Many-core processors such as the Xeon Phi have a large number of cores. Therefore, it is important to study how GSCPM scales as the number of processing cores increases. The Cilkview scalability analyzer is a software tool for estimating the scalability of multithreaded Cilk Plus applications. Cilkview will estimate parallelism and predict how the application will scale with the number of processing cores [HLL10].

Figure 4.1 shows the scalability profile produced by Cilkview that results from a single instrumented serial run of the GSCPM algorithm for different numbers of tasks. The curves show the amount of available parallelism in our algorithm; they are lower bounds indicating an estimation of the potential program speedup with the given grain size. As can be seen, fine-grained parallelism (many tasks) is needed for MCTS

to achieve good intrinsic parallelism. The performance of the GSCPM algorithm for more than 2048 tasks on 61 cores shows near-perfect speedup. Therefore, GSCPM has adequate parallelism. However, the actual performance of a parallel application is determined not only by its intrinsic parallelism but also by the performance of the runtime scheduler. Therefore, it is important to have an efficient implementation using modern threading libraries. In the following three sections, we will present the experimental setup (Section 4.7), the experimental design (Section 4.8), and the experimental results (Section 4.9) of five methods for parallel implementation with the help of four different threading libraries for GSCPM.

4.7 Experimental Setup

The performance evaluation of GSCPM is carried out 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. 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 Xeon Phi has 16GB GDDR5 memory on board with an aggregate theoretical bandwidth of 352 GB/s.

The Intel Composer XE 2013 SP1 compiler was used to compile for both Intel Xeon CPU and Intel Xeon Phi. Five methods for parallel implementation from four different threading libraries were used: (1) standard thread library comes from C++11 libraries, (2) Thread Pool with FIFO scheduling (TPFIFO) is based on Boost C++ libraries 1.41, (3,4) *cilk_spawn* and *cilk_for* come from Intel Cilk Plus, and (5) *task_group* comes from Intel TBB 4.2. We compiled the code using the Intel C++ Compiler with a -$O3$ flag.

4.8 Experimental Design

The goal of this chapter is to study the performance and scalability of *task-level parallelization* for MCTS as an irregular unbalanced algorithm on the Xeon Phi (see also RQ2). We do so using the ParallelUCT package (see Section 2.6). The package implements, a highly optimized, Hex playing program to generate realistic real-world search spaces.

To generate statistically significant results in a reasonable amount of time, 2^{20} playouts are executed to choose a move. The board size is 11×11. The UCT constant C_p is set at 1.0 in all of our experiments. To calculate the playout speedup, the average of time over ten games is measured for making the first move of the game

Table 4.2: Sequential baseline for GSCPM algorithm. Time in seconds.

Processor	Board Size	Sequential Time (s)
Xeon CPU	11×11	21.47 ± 0.07
Xeon Phi	11×11	185.37 ± 0.53

when the board is empty. The empty board is used because it has the most significant playout time; it is the most time-consuming position (since the whole board should be filled randomly). The results are within less than 3% standard deviation which is an acceptable tolerance.

4.9 Experimental Results

In Paragraph A, the performance of a sequential implementation of the MCTS algorithm on both the Xeon CPU and the Xeon Phi is reported. The performance and scalability of task-level parallelization of MCTS are measured on the Xeon CPU in Paragraph B and on the Xeon Phi in Paragraph C.

A: Sequential Performance

Table 4.2 shows the sequential time to execute the specified number of playouts. The time values in Table 4.2 are used to calculate playout speedup values (i.e., sequential time divided by parallel time to execute the equal number of playouts) in Figure 4.2a and Figure 4.2b. The sequential time on the Xeon Phi is almost eight times slower than on the Xeon CPU. This is because each core on the Xeon Phi is slower than each one on the Xeon CPU. (The Xeon Phi has in-order execution, the CPU has out-of-order execution, hiding the latency of many cache misses.)

The time of execution in the first game is longer on the Xeon Phi than on a Xeon CPU. Therefore the overhead costs for thread creation may include a significant contribution to the parallel region execution time. This is a known feature of the Xeon Phi, called the warm-up phase [RJ14]. Therefore, the first game is not included in the results to remove that overhead. The majority of threading library implementations do not destroy the threads created for the first time [RJ14].

B: Performance and Scalability on Xeon CPU

The graph in Figure 4.2b shows the performance and scalability of task-level parallelization for MCTS in terms of playout speedup for different threading libraries on a

Xeon CPU, as a function of the number of tasks. We recall that going to the right of the graph, finer grain parallelism is observed.

For the C++11 implementation, the number of threads is equal to the number of tasks. The best performance for the C++11 implementation is around 18 times speedup for 128 threads/tasks. The C++11 method does not scale after 128 threads or tasks. For the other methods, the number of threads is fixed and equal to the number of cores while the number of tasks is increasing. The best performance for the *cilk_spawn* and the *cilk_for* methods is around 13 times speedup for 32 tasks. The scalability of both the *cilk_spawn* method and the *cilk_for* method drops after 32 tasks and becomes almost stable around 12 times speedup after 128 tasks. For the *task_group* implementation the best performance is around 19 times speedup for 2048 tasks. The scalability of the *task_group* method becomes almost stable around 19 times speedup after 512 tasks.

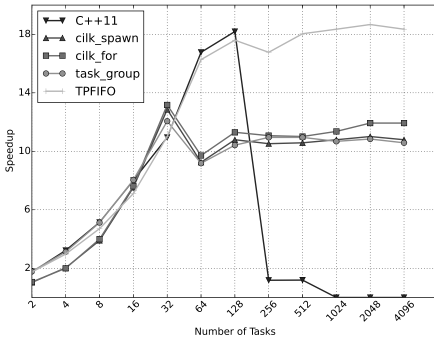
C: Performance and Scalability on Xeon Phi

The graph in Figure 4.2b shows the performance and scalability of task-level parallelization for MCTS in terms of playout speedup for different threading libraries on a Xeon Phi, as a function of the number of tasks. We recall that going to the right of the graph, finer grain parallelism is observed.

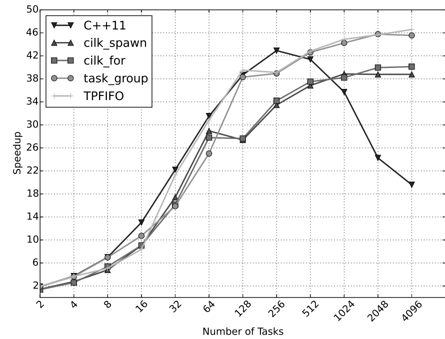
Creating a number of threads in C++11 that is equal to the number of tasks is the first approach that comes to mind. The best performance for the C++11 method is around 43 times speedup for 256 threads/tasks. However, the limitation of this approach is that creating larger numbers of threads has a large overhead. Thus, the method does not scale beyond 256 threads/tasks. For the *cilk_spawn* method and the *cilk_for* method, the best performance is achieved for fine-grained tasks. The best performance of *cilk_spawn* and *cilk_for* is close to a speedup of around 39 times (for 1024 tasks), and to a speedup of around 40 times (for 2048 tasks), respectively. The best performance of *task_group* and TPFIFO is also quite close to a 46 and 47 times speedup, respectively. TPFIFO and *task_group* scale well for up to 4096 tasks. The reason for the similarity between TBB and TPFIFO on the Xeon Phi is explained in Appendix C.

4.10 Discussion and Analysis

We have studied the performance of GSCPM on both the Xeon CPU (see Figure 4.2a) and the Xeon Phi (see Figure 4.2a) for five parallel implementation methods. These methods use a range of scheduling policies, ranging from a work-sharing FIFO work queue, to state-of-the-art work-sharing and work-stealing techniques in Cilk Plus and



(a) Speedup on the Intel Xeon CPU with 24 cores and 48 hyperthreads.



(b) Speedup on the Intel Xeon Phi with 61 cores and 244 hardware threads.

Figure 4.2: Speedup for task-level parallelization utilizing five methods for parallel implementation from four threading libraries. Higher is better. Left: coarse-grained parallelism. Right: fine-grained parallelism.

TBB libraries. Therefore, we compare our results on the Xeon Phi to the results on the Xeon CPU for analyzing and understanding the performance of each method on these two hardware platforms to find out the best method of implementation for each of these processors. The following contains three of our main observations on (A) scaling behavior, (B) performance, and (C) range of tasks.

A: Scaling behavior

First, it is noticeable that we achieve good scaling behavior, a speedup of 47 on the 61 cores of the Xeon Phi and a speedup of 19 on the 24 cores of the Xeon CPU. Surprisingly, this performance is achieved using one of the most straightforward scheduling mechanisms, a work-sharing FIFO thread pool. We expected to observe a similar or even better performance for Cilk Plus methods (*cilk_spawn* and *cilk_for*). These methods are designed explicitly for irregular and unbalanced (divide and conquer) parallelism using a work-stealing scheduling policy. The performance of the TBB method (*task_group*) is close to the FIFO method on the Xeon Phi, but its performance on the Xeon CPU is definitely worse than TPFIFO.

B: Performance

Second, the performance of each method depends on the hardware platform. We see five interesting facts.

- B1: It is shown that on the Xeon CPU (see Figure 4.2a), by doubling the numbers of tasks the running time becomes almost half for up to 32 threads for C++11, Cilk Plus (*cilk_spawn* and *cilk_for*), TBB (*task_group*), and TPFIFO. It means that all of these methods at least scale for up to 32 threads on the Xeon CPU. It is also shown that on the Xeon Phi (see Figure 4.2b), all of these methods achieve very close performance for up to 64 tasks. It means that they at least scale for up to 64 threads on the Xeon Phi.

- B2: The best performance for C++11 is observed for 128 threads/tasks on the Xeon CPU and 256 threads/tasks on the Xeon Phi. It shows that C++11 does not scale on the Xeon CPU and the Xeon Phi for fine-grained tasks which subsequently reveals the limitation of thread-level parallelization. Moreover, for 64 and 128 threads/tasks, the speedup for C++11 is better than for Cilk Plus and TBB on the Xeon CPU.

- B3: The best performance for *cilk_spawn* and *cilk_for* on the Xeon CPU is observed for coarse-grained tasks, when the numbers of tasks are equal to 32. The best speedup for *cilk_spawn* and *cilk_for* on the Xeon Phi is observed for fine-grained tasks, when the numbers of tasks are more than 2048. It shows the optimal task grain size for Cilk Plus on the Xeon CPU is different from the Xeon Phi. Moreover, the measured performance for Cilk Plus methods comes quite close to TBB on the Xeon CPU, while it never reaches to TBB performance on the Xeon Phi after 64 tasks. Cilk Plus' speedup is less than the other methods up to 16 threads.

- B4: The best performance for *task_group* on the Xeon CPU is measured for coarse-grained tasks, when the number of tasks is equal to 32. The best speedup for *task_group* on the Xeon Phi is observed for fine-grained tasks, when the numbers of tasks are more than 2048. It shows the optimal task grain size for *task_group* on the Xeon CPU is different from that for the Xeon Phi. Moreover, the measured speedup for *task_group* comes quite close to TPFIFO on the Xeon Phi, while it never reaches TPFIFO performance on the Xeon CPU after 32 tasks.

- B5: The best performance for TPFIFO on the Xeon CPU is measured for fine-grained tasks when the number of tasks is equal to 2048. The best speedup for TPFIFO is on the Xeon Phi is also observed for fine-grained tasks, when the number of tasks is 4096. It shows the optimal task grain size for TPFIFO on the Xeon CPU is similar to that on the Xeon Phi.

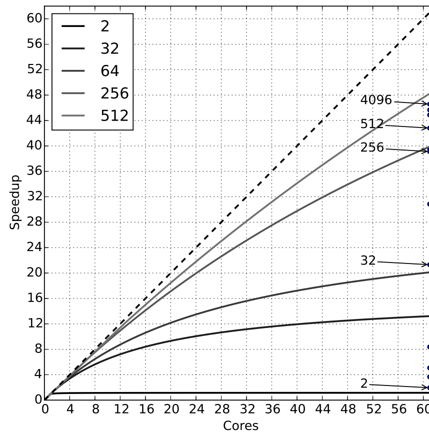


Figure 4.3: Comparing Cilkview analysis with TPFIFO speedup on the Xeon Phi. The dots show the number of tasks used for TPFIFO. The lines show the number of tasks used for Cilkview.

C: Range of tasks

Third, Figure 4.3 shows a mapping from TPFIFO speedup to the Cilkview graph for 61 cores (=244 hardware threads). We see (shown by dots) the speedup for a range of tasks (shown by a line, from 2 to 4096). We remark that the results of Figure 4.2b correspond nicely to the Cilkview results for up to 256 tasks. Thus, the 256-task dot occurs on the 256-task line. However, the 512-task line is above the actual 512-task dot and also the 4096-task dot. After the 256-task dot, the speedup continues to improve but not as expected by Cilkview due to overheads. If the program performs beneath the range of expectation, the programmer can be confident in seeking a cause such as insufficient memory bandwidth, false sharing, or contention, rather than inadequate parallelism or insufficient grain size. The source of the contention in GSCPM is the locked-based shared search tree. Addressing this issue will be the topic of Chapter 5.

In our analysis, we found the notion of grain size to be of central importance to achieve task-level parallelization. The traditional thread-level parallelization of MCTS uses a one-to-one mapping of the logical tasks to the hardware threads to implement different parallelization algorithms (Tree Parallelization and Root Parallelization); see, e.g., [CWvdH08a, MPvdHV15a, MPVvdH14].

4.11 Related Work

Below we discuss four related papers. First, Saule et al. [SÇ12] compared the scalability of Cilk Plus, TBB, and OpenMP for a parallel graph coloring algorithm. They also studied the performance of programming models, as mentioned above, for a micro-benchmark with irregular computations. The micro-benchmark was a *for* loop that is parallelized and specifically designed to be less memory intensive than graph coloring. The maximum speedup for this micro-benchmark on the Xeon Phi was 47 and is obtained by using 121 threads.

Second, authors from [TV15] used a thread pool with work-stealing scheduling and compared its performance to the three libraries: (1) OpenMP, (2) Cilk Plus, and (3) TBB. They used a parallel program that calculates the Fibonacci numbers by concurrent recursion as an example of unbalanced tasks. In contrast to our approach with work-sharing scheduling, their approach with work-stealing scheduling shows no improvement in performance over the selected libraries for Fibonacci before using 2048 tasks.

Third, Baudiš et al. [BG11] reported the performance of lock-free Tree Parallelization for up to 22 threads. They used a different speedup measure. The strength speedup is good up to 16 cores, but the improvement drops after 22 cores.

Fourth, Yoshizoe et al. [YKK+11] study the scalability of the MCTS algorithm on distributed systems. They have used artificial game trees as the benchmark. Their closest settings to our study are 0.1 ms playout time and a branching factor of 150 with 72 distributed processors. They showed a maximum of 7.49 times speedup for distributed UCT on 72 CPUs. They have proposed depth-first UCT and reached 46.1 times speedup for the same number of processors.

4.12 Answer to RQ2

In this chapter, we presented the task-level parallelization for parallelizing of MCTS. We addressed *RQ2*.

- ***RQ2***: *What is the performance and scalability of task-level parallelization for MCTS on both multi-core and many-core shared-memory machines?*

The performance of task-level parallelization to implement the GSCPM algorithm on a multi-core machine with 24 cores was adequate (see Paragraph B of Section 4.9). It reached a speedup of 19, and the FIFO scheduling method showed good scalability for up to 4096 tasks. The performance of task-level parallelization on a many-core co-processor, with the high level of optimization of our sequential code-base, was also

good; a speedup of 47 on the 61 cores of the Xeon Phi was reached (see Paragraph C of Section 4.9). Moreover, the FIFO and *task_group* methods showed good scalability for up to 4096 tasks on the Xeon Phi (see Section 4.10). However, our scalability study showed that there is still potential for improving performance and scalability by removing synchronization overhead. This issue will be the topic for the next chapter.