Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

6

Pipeline Pattern for Parallel MCTS

This chapter[1] addresses *RQ4* which is mentioned in Section 1.7.

- ***RQ4***: *What are the possible patterns for task-level parallelization in MCTS, and how do we use them?*

In recent years there has been much interest in the Monte Carlo Tree Search (MCTS) algorithm. In 2006 it was a new, adaptive, randomized optimization algorithm [Cou06, KS06]. In fields as diverse as Artificial Intelligence, Operations Research, and High Energy Physics, research has established that MCTS can find valuable approximate answers without domain-dependent heuristics [KPVvdH13]. The strength of the MCTS algorithm is that it provides answers with a random amount of error for any fixed computational budget [GBC16]. Much effort has been put into the development of parallel algorithms for MCTS to reduce the running time. The efforts are applied to a broad spectrum of parallel systems; ranging from small shared-memory multi-core machines to large distributed-memory clusters. In the last years, parallel MCTS played a major role in the success of AI by defeating humans in the game of Go [SHM+16, HS17].

The general MCTS algorithm has four operations inside its main loop (see Algorithm 2.1). This loop is a good candidate for parallelization. Hence, a significant effort has been put into the development of parallelization methods for MCTS

[1] Based on:

- S. A. Mirsoleimani S., H. J. van den Herik, A. Plaat and J. Vermaseren, Pipeline Pattern for Parallel MCTS, in Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2, 2018, pp. 614--621.

[CWvdH08a, YKK+11, FL11, SP14, MPvdHV15b]. In Chapter 4, we defined Iteration-Level Parallelism (ILP) to reach task-level parallelization for MCTS [MPvdHV15a]. In ILP the computation associated with each iteration is assumed to be independent. Therefore, we can assign a chunk of iterations as a separate task to each parallel thread for execution on separate processors (see Section 4.4). Close analysis has learned that each iteration in the chunk can also be decomposed into separate operations for parallelization. Based on this idea, we introduce Operation-Level Parallelism (OLP). The main point is to assign each operation of MCTS to a separate task for execution by separate processors. This type of task is called Operation-Level Task (OLT). This leads to flexibility in managing the control flow of the operations in the MCTS algorithm. The main contribution of this chapter is introducing a new algorithm based on the Pipeline Pattern for Parallel MCTS (3PMCTS) and showing its benefits.

Definition 6.1 (Operation-Level Task) *The operation-level task is a type of task that contains one of the MCTS operations.*

Definition 6.2 (Operation-Level Parallelism) *Operation-level parallelism is a type of parallelism that enables task-level parallelization to assign each of the MCTS operations inside an iteration as a separate task for execution on separate processors.*

The remainder of the chapter is organized as follows. In Section 6.1 the data dependencies challenges are described. Section 6.2 provides necessary definitions and explanations for the design of 3PMCTS. Section 6.3 gives the explanations for the implementation the 3PMCTS algorithm, Section 6.4 shows the experimental setup, Section 6.5 describes the experimental design, and Section 6.6 gives the experimental results.

6.1 Data Dependencies Challenges

One of the obstacles for parallelizing MCTS is the two types of data dependencies that exist among the steps in the MCTS algorithm. Parallel execution of the steps without considering related data dependencies may cause danger of getting wrong results. In the following, we explain these two types of data dependencies formally based on two control flow patterns: *sequence* and *iteration*.

6.1.1 Loop Independent Data Dependency

Each iteration of the MCTS algorithm has a sequence pattern. As it is shown in Figure 1.2, function SELECT will execute before function EXPAND, which will execute before function PLAYOUT. In the sequence pattern for MCTS the algorithm text ordering will

be followed, because there are data dependencies between the operations. We define this type of data dependency as Operation-Level Dependency (OLD).

Definition 6.3 (Sequence Pattern) *A sequence pattern is an ordered list of tasks that are executed in a specific order [MRR12]. Each task is finished before the one after it starts.*

The result of violating the operation-level dependencies would be an incorrect algorithm. Therefore, all the approaches for parallelizing MCTS should not break this type of dependency. The consequence of accepting this limitation is that the opportunities for parallelization are restricted only to the iterations of the main loop. There is also a second type of data dependencies among the iterations that we will address in the next subsection.

6.1.2 Loop Carried Data Dependency

The main loop of the MCTS algorithm has an *iteration* pattern. The body of the loop depends on previous invocations of itself because the algorithm needs the past updates to make an optimal selection in the future. We define this type of data dependency as Iteration-Level Dependency (ILD).

Definition 6.4 (Iteration Pattern) *In an iteration pattern, a condition is evaluated. If it is true, a task is executed, then the condition is re-evaluated, and the process repeats until the condition becomes false.*

The result of violating the iteration-level dependencies would be the search overhead in parallelized MCTS because a new selection in one thread may not have access to the updates from other threads. Therefore, the parallel algorithm conducts repeated or unnecessary searches. All the approaches for parallelizing MCTS should break this type of dependency, otherwise parallelization is not possible [KUV15]. The ideal scenario is to achieve parallelism while minimizing the search overhead. In the next subsection, we introduce our solution to reach this goal.

6.1.3 Why a Pipeline Pattern?

In the previous chapter, we have introduced the fork-join pattern for parallel MCTS. This parallel pattern provides structured parallelism for MCTS. However, we disrupt the decision making process in the MCTS algorithm by using the fork-join pattern. The key element of the MCTS algorithm is the UCT formula which controls the level of exploitation versus exploration to make the best decision in each iteration of the algorithm. The UCT formula requires updates from the previous iterations; however,

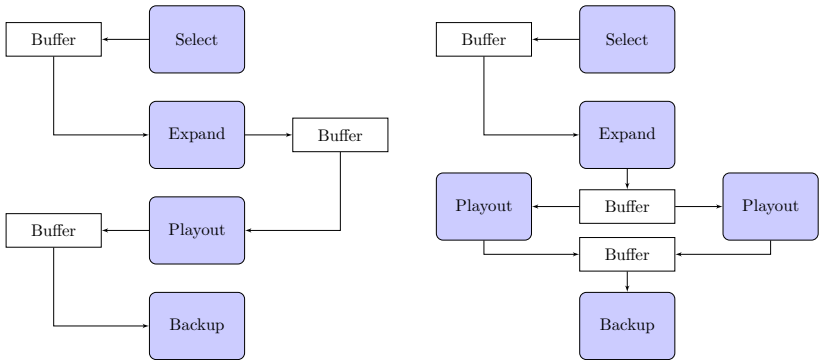


Figure 6.1: (6.1a) Flowchart of a pipeline with sequential stages for MCTS. (6.1b) Flowchart of a pipeline with parallel stages for MCTS.

parallelization based on the fork-join pattern cannot fulfill this requirement. The pipeline pattern is the only parallel pattern that allows us to handle the challenge of data dependencies and to avoid the problem of search overhead to some extent. In the next section, we provide the details of the proposed algorithm for parallelizing MCTS based on the pipeline pattern.

Definition 6.5 (Pipeline Pattern) *A pipeline pattern is a pattern of computation in which a set of processing elements is connected in series, generally so that the output of one element is the input of the next one. The elements of a pipeline are often executed concurrently.*

6.2 Design of 3PMCTS

In this section, we describe our proposed method for parallelizing MCTS. Section 6.2.1 describes *how* the pipeline pattern is applied in MCTS. Section 6.2.2 provides the 3PMCTS algorithm.

6.2.1 A Pipeline Pattern for MCTS

Below we describe *how* the pipeline pattern is used as a building block in the design of 3PMCTS. Figure 6.1 shows two types of pipelines for MCTS. The inter-stage buffers are used to pass information between the stages. When a stage of the pipeline completes its computation, it sends a path of nodes from the search to the next buffer.

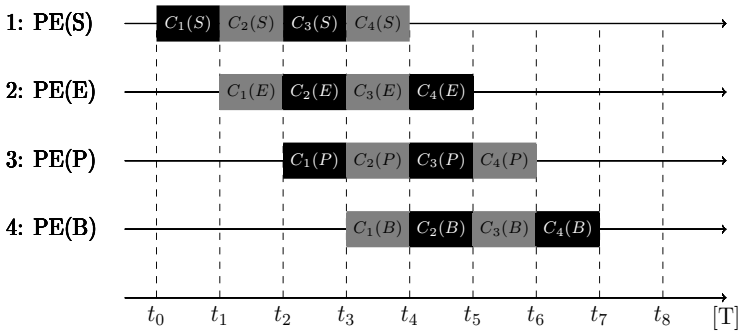


Figure 6.2: Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are equal.

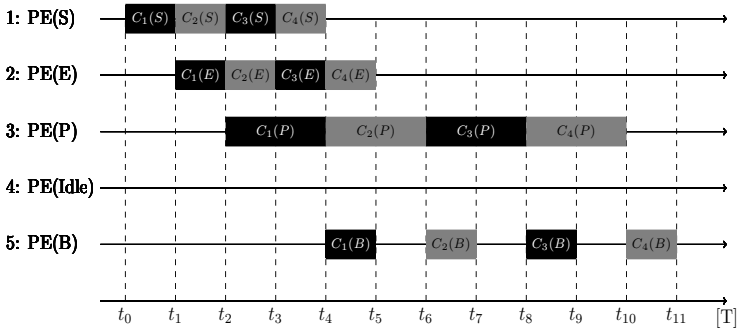


Figure 6.3: Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are not equal.

The subsequent stage picks a path from the buffer and starts its computation. Here we introduce two possible types of pipelines for MCTS.

1. *Pipeline with sequential stages*: Figure 6.1a shows a pipeline with sequential stages for MCTS. The idea is to map each MCTS operation to pipeline stages such that each stage of the pipeline computes one operation. Figure 6.2 illustrates how the pipeline executes the MCTS operations over time. Let C_i represent a multiple-step computation on path i. $C_i(j)$ is the jth step of the computation in MCTS (i.e., $j \in O = \{\text{S, E, P, B}\}$ and the elements of the set O are the first letters of the MCTS operations). Initially, the first stage of the pipeline performs $C_1(S)$. After the step has been completed, the second stage of the pipeline receives the first path and computes $C_1(E)$ while the first stage computes the first step of the second path, $C_2(S)$. Next, the third stage computes $C_1(P)$, while the second stage computes $C_2(E)$ and the first stage $C_3(S)$. Each

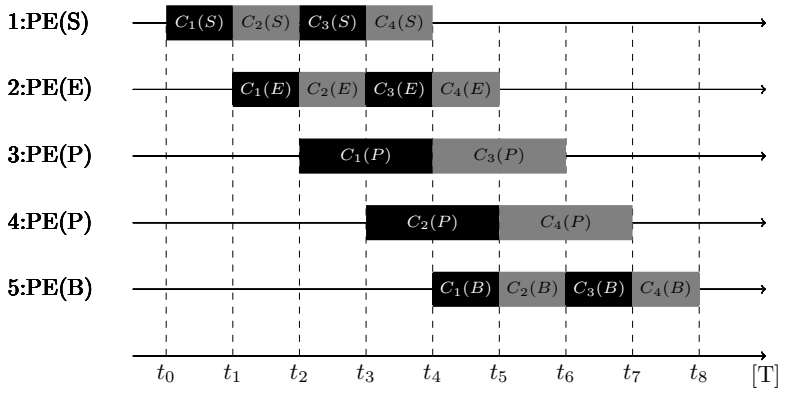


Figure 6.4: Scheduling diagram of a pipeline with parallel stages for MCTS. Using parallel stages create load balancing.

stage of the pipeline takes the same amount of time to do its work, say T. Figure 6.2 shows that the expected execution time for 4 paths in an MCTS pipeline with four stages is approximately $7 \times T$. In contrast, the sequential version takes approximately $16 \times T$ because each of the 4 paths must be processed one after another. The pipeline pattern works best if the operations performed by the various stages of the pipeline are all about equally computationally intensive. If the stages in the pipeline vary in computational effort, the slowest stage creates a bottleneck for the aggregate throughput. In other words, when there are a sufficient number of processors for each pipeline stage, the speed of a pipeline is approximately equal to the speed of its slowest stage. For example, Figure 6.3 shows the scheduling diagram that occurs when the PLAYOUT stage takes $2 \times T$ units of time while others take T units of time. Figure 6.3 shows that the expected execution time for 4 paths is approximately $11 \times T$.

2. *Pipeline with parallel stages*: Figure 6.1b shows a pipeline for MCTS with two parallel PLAYOUT stages. Using two PLAYOUT stages in the pipeline results in an overall speed of approximately T units of time per path as the number of paths grows. Figure 6.4 shows that the MCTS pipeline is perfectly balanced by using two PLAYOUT stages. The expected execution time for 4 paths is approximately $8 \times T$. Therefore, introducing parallel stages improves the scalability of the MCTS pipeline.

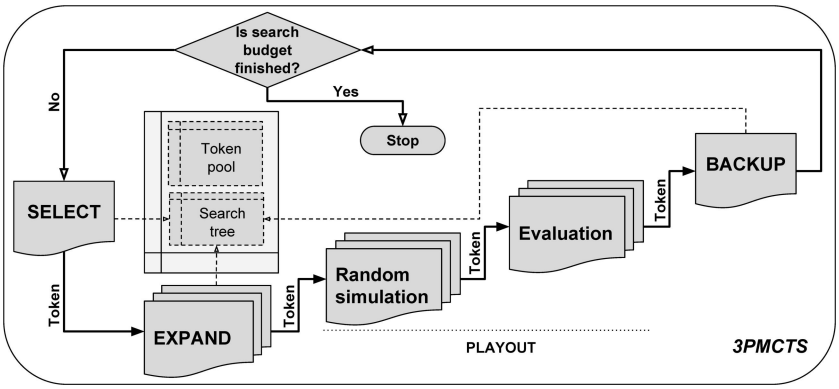


Figure 6.5: The 3PMCTS algorithm with a pipeline that has three parallel stages (i.e., EXPAND, RANDOMSIMULATION, and EVALUATION).

6.2.2 Pipeline Construction

The pseudocode of MCTS is shown in Algorithm 2.1. Each operation in MCTS constitutes a stage of the pipeline in 3PMCTS. In contrast to the existing methods, 3PMCTS is based on OLP for parallelizing MCTS. The pipeline pattern can satisfy the operation-level dependencies among the OLTs.

The potential concurrency is also exploited by assigning each stage of the pipeline to a separate processing element for execution on separate processors. If the pipeline has only sequential stages then the speedup is limited to the number of stages.[2] However, in MCTS, the operations are not equally computationally intensive, e.g., the PLAYOUT operation (random simulations plus evaluation of a terminal state) could be more computationally expensive than other operations. Therefore, 3PMCTS uses a pipeline with parallel stages. Introducing parallel stages makes 3PMCTS more scalable.

Figure 6.5 depicts one of the possible pipeline constructions for 3PMCTS. We split the PLAYOUT operation into two stages to achieve more parallelism (See Section 1.2). The five stages run the MCTS operations SELECT, EXPAND, RANDOMSIMULATION, EVALUATION, and BACKUP, in that order. The SELECT stage and BACKUP stage are serial. The three middle stages (EXPAND, RANDOMSIMULATION, and EVALUATION) are parallel and do the most time-consuming part of the search. A serial stage does process one token at a time. A parallel stage is able to process more than one token. Therefore, it needs more than one in-flight *token*. A token represents a path of nodes inside the search tree during the search.

[2]This holds when the operations performed by the various stages are all about equally computationally intensive.

The pipeline depicted in Figure 6.5 is one of the possible constructions for the 3PMCTS algorithm. Each of the five stages could be either serial or parallel. Therefore, 3PMCTS provides a great level of flexibility. For example, a pipeline could have a serial stage for the SELECT operation and a parallel stage for the BACKUP operation. In our experiments we use this construction (see Section 6.6).

6.3 Implementation Considerations

We have implemented the proposed 3PMCTS algorithm in the *ParallelUCT* package [MPvdHV15a]. The ParallelUCT package is an open source library for parallelization of the UCT algorithm (see Section 2.6). It uses *task-level parallelism* to implement different parallelization methods for MCTS. We have also used an algorithm called *grain-sized control parallel MCTS* (GSCPM) to measure the performance of ILP for MCTS. The GSCPM algorithm creates tasks based on the *fork-join pattern* [MRR12]. More details about this algorithm can be found in [MPvdHV15a]. Both 3PMCTS and GSCPM are implemented by the TBB parallel programming library [Rei07] and they are available online as part of the ParallelUCT package. In our implementation for the 3PMCTS algorithm, we can specify the number of in-flight tokens. This is equal to the number of tasks for the GSCPM algorithm. The details of the implementation are provided in Appendix B.

6.4 Experimental Setup

The performance of 3PMCTS is measured by using a High Energy Physics (HEP) expression simplification problem [KPVvdH13, RVPvdH14]. Our setup follows closely [KPVvdH13]. We discuss the case study in Subsection 6.4.1, the hardware in Subsection 6.4.3, and the performance metrics in Subsection 6.4.2.

6.4.1 Horner Scheme

Our case study is in the field of Horner's rule, which is an algorithm for polynomial computation that reduces the number of multiplications and results in a computationally efficient form. For a polynomial in one variable

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \tag{6.1}$$

the rule simply factors out powers of x. Thus, the polynomial can be written in the form

$$p(x) = ((a_n x + a_{n-1})x + \ldots)x + a_0. \tag{6.2}$$

This representation reduces the number of multiplications to n and has n additions. Therefore, the total evaluation cost of the polynomial is $2n$. Horner's rule can be generalized for multivariate polynomials. The order of choosing variables may be different, each order of the variables is called a *Horner scheme*, see Section 2.4.2.

We are using a polynomial from HEP domain, namely HEP(σ) expression with 15 variables to study the results of 3PMCTS [Ver13, KPVvdH13]. The MCTS is used to find an order of the variables that gives efficient Horner schemes [RVPvdH14]. The root node has n children, with n the number of variables. The children of other nodes represent the remaining unchosen variables in order. Starting at the root node, a path of nodes (variables) inside the search tree is selected. The incomplete order is completed with the remaining variables added randomly (i.e., RANDOMSIMULATION). The complete order is then used for Horners method followed by CSE to optimize the expression. The number of operations (i.e., Δ) in this optimized expression is counted (i.e., EVALUATION).

6.4.2 Performance Metrics

In our experiments, the performance is reported by (A) playout speedup or speedup (see Eq. 2.5) and (B) playing strength or the *number of operations* in the optimized expression (see Paragraph B2 of Subsection 2.5.2). A lower value is desirable for the second metric when we compare higher numbers of tasks. 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.

6.4.3 Hardware

Our experiments were performed on a dual socket Intel machine with 2 Intel Xeon E5-2596v2 CPUs running at 2.4 GHz. 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. We compiled the code using the Intel C++ compiler with a -$O3$ flag.

6.5 Experimental Design

In our experiments, the maximum number of playouts is 8192. Throughout the experiments, the number of threads is multiplied by a factor of two. Each data point represents the average of 21 runs.

Table 6.1: Sequential time in seconds when $C_p = 0.5$.

Processor	Num. Playouts	Time (s)
CPU	8192	215.72 ± 4.12

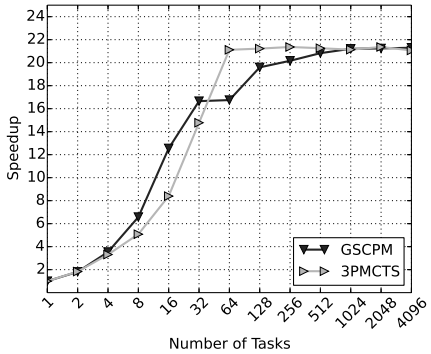


Figure 6.6: Playout-speedup as function of the number of tasks (tokens). Each data point is an average of 21 runs for a search budget of 8192 playouts. The constant C_p is 0.5. Here a higher value is better.

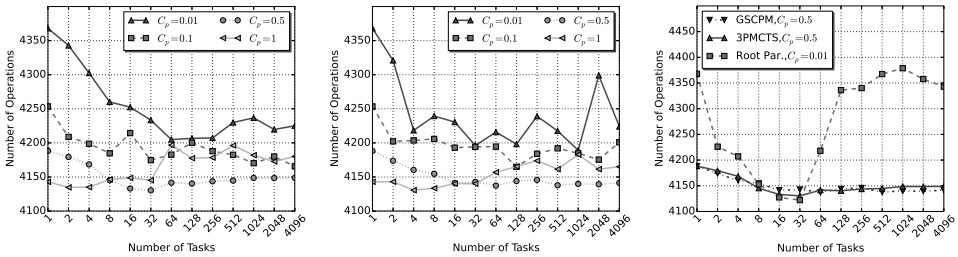
6.6 Experimental Results

In this section, we first provide the experimental results on the performance and the scalability of 3PMCTS in Subsection 6.6.1. In Subsection 6.6.2, the experimental results on the flexibility of task decomposition in 3PMCTS are shown and discussed.

6.6.1 Performance and Scalability of 3PMCTS

In this section, the performance of 3PMCTS is measured. Table 6.1 shows the sequential time to execute the specified number of playouts.

Figure 6.6 shows the playout-speedup for both 3PMCTS and GSCPM, as a function of the number of tasks (from 1 to 4096). The search budget for both algorithms is 8192 playouts. The 3PMCTS algorithm uses a pipeline with five stages for MCTS operations. Four stages are parallel; the SELECT stage is chosen to be serial (see the end of Section 6.2.2). A playout-speedup close to 21 on a 24-core machine is observed for both algorithms. From our results, we may provisionally conclude that 3PMCTS (a) for 4 to 32 parallel tasks, shows a speedup less than GSCPM and (b) for 64 to 512 parallel tasks, shows a better speedup than the GSCPM algorithm (see Figure 6.6). At the same time, 3PMCTS also allows flexible control of the parallel or serial



(a) 3PMCTS (b) GSCPM (c) Root Parallelization

Figure 6.7: Number of operations as function of the number of tasks (tokens). Each data point is an average of 21 runs for a search budget of 8192 playouts. Here a lower value is better.

execution of MCTS operations (e.g., the SELECT stage is sequential and the BACKUP stage is parallel in our case), something that GSCPM cannot provide.

Figure 6.7a and 6.7b show the results of the optimization in the number of operations in the final expression for both algorithms. These results show consistency with the findings in [KPVvdH13, RVPvdH14]. From our results, we may arrive at three conclusions. (1) When MCTS is sequential (i.e., the number of tasks is 1), for small values of C_p, such that MCTS behaves exploitively, the method gets trapped in local minima, and the number of operations is high. For larger values of C_p, such that MCTS behaves exploratively, lower values for the number of operations are found. (2) When MCTS is parallel, for small numbers of tasks (from 2 to 8), it turns out to be good to choose a high value for the constant C_p (e.g., 1) for both 3PMCTS and GSCPM. With higher numbers of tasks, a lower value for C_p in the range [0.5; 1) seems suitable for both algorithms. Figure 6.7 also shows that 3PMCTS can find a lower number of operations for 8, 16, and 32 tasks when $C_p = 0.5$. (3) When both algorithms find the same number of operations, the one with higher speedup is better. For instance, the 3PMCTS algorithm finds the same number of operations compared to GSCPM for 64 tasks, but it has higher speedup when $C_p = 0.5$. Note that these values hold for a particular polynomial and that different polynomials give different optimal values for C_p and number of tasks.

A comparison to Root Parallelization is illustrated in Figure 6.7c. Both 3PMCTS and GSCPM belong to the category of Tree Parallelization. For $C_p = 0.01$, Root Parallelization finds a lower number of operations for both 16 and 32 tasks compared to the two other methods. However, increasing the number of tasks causes Root Parallelization to provide a much higher number of operations. From these results, we may conclude that Root Parallelization could also be a feasible choice in this domain.

Table 6.2: Definition of layouts for 3PMCTS.

Layout Name	Num. Parallel Stage	Seq. Stage
3PMCTS(5-4-S)	4	SELECT
3PMCTS(5-4-B)	4	BACKUP

Table 6.3: Details of experiment to show the flexibility of 3PMCTS.

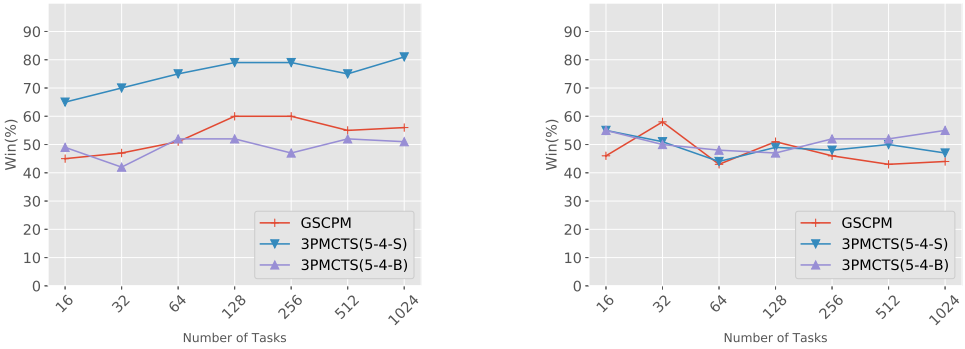
C_p	Player a	Player b
0.01	GSCPM 3PMCTS(5-4-S) 3PMCTS(5-4-B)	GSCPM with 8 tasks
1	GSCPM 3PMCTS(5-4-S) 3PMCTS(5-4-B)	GSCPM with 8 tasks

Kuipers et al. remarked that Tree Parallelization would give a result that is statistically a little bit inferior to a run with sequential MCTS with the same number of playouts due to the violation of iteration-level dependency that produces search overhead [KUV15]. It is clear from our results that the effectiveness of any parallelization method for MCTS depends heavily on the choice of three parameters: (1) the C_p constant, (2) the number of playouts, and (3) the number of tasks. If we select these parameters carefully, it is possible to overcome the search overhead to some extent. Furthermore, the 3PMCTS algorithm provides the flexibility of managing the execution (serial or parallel) of different MCTS operations that helps us even more to achieve this goal.

6.6.2 Flexibility of Task Decomposition in 3PMCTS

The most important feature of 3PMCTS is the flexibility in alternating each of its stages from being parallel to be serial and vice versa. Table 6.2 shows two of the possible layouts for 3PMCTS. In each layout, the first number inside the parentheses shows the total number of stages in the pipeline. The second number is the number of parallel stages, and the last letter identifies which one of the stages is serial. For example, one of the layouts for 3PMCTS is 3PMCTS(5-4-S). This layout has five stages, four of which are parallel and the serial stage is SELECT.

An Experiment is designed to present the effect of flexibility on the behavior of 3PMCTS. Table 6.3 gives the details of the experiment with the game Hex (11×11 board). Both players use the same C_p value. In all test cases, the opponent player is



(a) C_p=0.01 (b) C_p=1

Figure 6.8: Percentage of win as function of the number of tasks (tokens). Each data point is the outcome of 100 rounds of playing between the two opponent players. Each player has a search budget of $2^{20} = 1,048,576$ playouts in each round. Here a higher value is better.

GSCPM with eight tasks.

Figure 6.8 illustrates the results of the experiment. 3PMCTS(5-4-S) strongly defeats GSCPM for $C_p = 0.01$ while 3PMCTS(5-4-B) does not show such a behavior. From the experiment we may conclude that when the selection step is sequential, flexibility can solve search overhead to a large extent.

6.7 Answer to RQ4

This chapter proposes solutions for the following question.

- ***RQ4***: *What are the possible patterns for task-level parallelization in MCTS, and how do we use them?*

Our research in the previous chapter showed that the task-level parallelization method combined with lock-free data structure for the GSCPM algorithm achieved a very good performance and scalability on multi-core and many-core processors (see Section 5.7).

The GSCPM algorithm was design-based on the iteration-level parallelism. Hence, it relies on the iteration pattern (see Section 4.4) that violates the iteration-level data dependencies (see Subsection 6.1.2). The result of this violation is search overhead. Therefore, scalability is only one issue, although it is an important one.

The second issue is to handle the search overhead. Thus, we designed the 3PMCTS algorithm based on operation-level parallelism which relies on the pipeline pattern with the aim to avoid violating the iteration-level data dependencies (see Section 6.2). Hence, we managed to control the search overhead using the flexibility of task decomposition.

Based on our findings in this chapter we may conclude that different pipeline constructions are able to provide higher levels of flexibility that allow fine-grained managing of the execution of operations in MCTS (see Subsection 6.6.2). This is the answer to *RQ4*.