Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

1 Introduction

In the last decade, there has been much interest in the Monte Carlo Tree Search (MCTS) algorithm. It started by the publication "Bandit Based Monte-Carlo Planning", when Kocsis and Szepesvári proposed a new, adaptive, randomized optimization algorithm [KS06]. In the same year, it was followed by Rémi Coulom in presenting "Efficient selectivity and backup operators in Monte-Carlo tree search" in Turin [Cou06]. After that, the time has arrived to collect the ideas in a framework for MCTS by Chaslot et al. [CWvdH+08b]. In fields as diverse as Artificial Intelligence, Combinatorial Optimization, and High Energy Physics (HEP), research has established that MCTS can find approximate answers without domain-dependent heuristics [KS06, KPVvdH13, Ver13]. The strength of the MCTS algorithm is that it provides answers for any given computational budget [GBC16]. The amount of error can typically be reduced by expanding the computational budget for more running time. Much effort has been put into the development of parallel algorithms for MCTS to reduce the running time. The efforts have as their target a broad spectrum of parallel systems, ranging from small shared-memory multi-core machines to large distributed-memory clusters. The emergence of the Xeon Phi co-processor with over 61 simple cores has extended this spectrum with shared-memory many-core processors. In this thesis, we will study the parallel MCTS algorithms for multi-core and many-core processors.

This chapter is structured as follows. Section 1.1 introduces the HEPGAME project. Section 1.2 explains briefly the MCTS algorithm. Section 1.3 discusses parallelism and parallelization. Section 1.4 explains the general obstacles to the parallelization of MCTS. Section 1.5 discusses performance and scalability. The scope and research goals are mentioned in Section 1.6. The problem statement and five research questions are given in Section 1.7. Section 1.8 discusses the research methodology. Section 1.9 gives the structure of the thesis. Section 1.10 provides a list of contributions.

1.1 HEPGAME

The work in the thesis is part of High Energy Physics Game (HEPGAME) project [Ver13]. The HEPGAME project intends to use techniques from game playing for solving large equations in particle physics (High Energy Physics (HEP)) calculations. One of these techniques is MCTS. Before the beginning of the project, it was clear that without parallelization any algorithm based on MCTS cannot be useful. The main prerequisite for the parallelization was that the algorithm should be executed in a reasonable time when trying to simplify large equations. Therefore, our focus in this research was on finding new methods to parallelize the MCTS algorithm. The multi-threaded version of the FORM program (i.e., TFORM) [TV10] can use our findings. FORM is open source software used for solving large High Energy Physics (HEP) equations. FORM has an optimization module which receives the main conclusions of our research.

1.2 Monte Carlo Tree Search

The MCTS algorithm iteratively repeats four steps to construct a search tree until a predefined computational budget (i.e., time or iteration constraint) is reached [Cou06, CWvdH+08b]. Figure 1.2 shows the main loop of the MCTS algorithm and Figure 1.1 shows an example of the search tree. At the beginning the search tree has only a root node. Each node in the search tree is a state of the domain, and directed edges to child nodes represent actions leading to the following states. Figure 1.3 illustrates one iteration of the MCTS algorithm on a search tree that already has nine nodes. Circles represent the non-terminal and internal nodes. Squares show the terminal nodes. The four steps are:

1. SELECT: A path of nodes inside the search tree is selected from the root node until a non-terminal leaf with unvisited children is reached (v_6). Each of the nodes inside the path is selected based on a predefined *selection policy*. This policy controls the balance between exploitation and exploration of searching inside the domain [KS06] (see Figure 1.3a).

2. EXPAND: One of the children (v_9) of the selected non-terminal leaf (v_6) is generated randomly and added to the tree and also the selected path (see Figure 1.3b).

3. PLAYOUT: From the given state of the newly added node, a sequence of randomly simulated actions is performed until a terminal state in the domain is

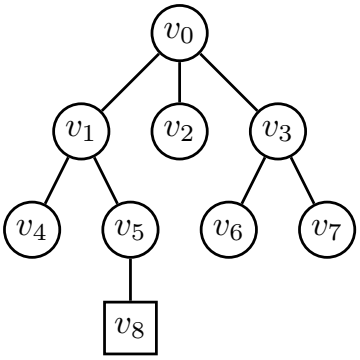


Figure 1.1: An example of the search tree.

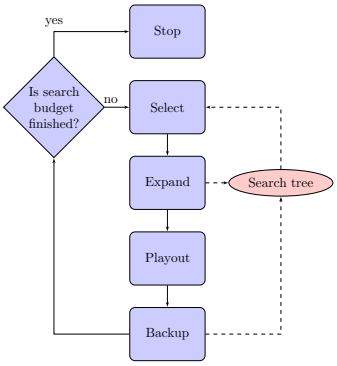


Figure 1.2: The main loop of MCTS.

reached, i.e., RANDOMSIMULATION. The terminal state is evaluated using a utility function to produce a reward value Δ, i.e., EVALUATION (see Figure 1.3c).

4. BACKUP: In the selected path, each node's visit count n is incremented by 1 and its reward value w updated according to Δ [BPW+12]. These values are required by the selection policy (see Figure 1.3d).

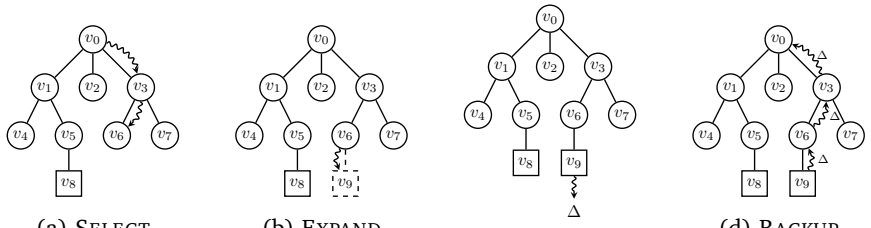
As soon as the computational budget is exhausted, the best child of the root node is returned (e.g., the one with the highest number of visits).

1.3 Parallelism and Parallelization

In this thesis, we aim at parallelism, and we use parallelization as the act towards parallelism. Doing more than one thing at the same time introduces *parallelism*. A programmer has to find opportunities for *parallelization* in an algorithm and use parallel programming methods to write a parallel program. A parallel program uses the parallel processing power of processors for faster execution.

Definition 1.1 (Parallelization) *Parallelization is the act of transforming code to enable simultaneous activities. The parallelization of a program allows execution of (at least parts of) the program in parallel.*

Below we describe two types of parallelism: thread-level parallelization in Subsection 1.3.1 and task-level parallelization in Subsection 1.3.2.



(a) SELECT (b) EXPAND (c) PLAYOUT (d) BACKUP

Figure 1.3: One iteration of MCTS.

1.3.1 Thread-level Parallelization

The first choice for doing parallel programming is to use software threads, such as POSIX threads, usually referred to as *pthreads*. It enables a program to control multiple different flows of work that overlap in time. Each flow of work is seen as a thread; creation and control over threads are achieved by making calls to the API (e.g., pThreads). Here we remark that the use of software threads in parallel programming is considered as equivalent to writing in assembly language [JR13]. A multi-core processor consists of multiple cores that execute at least one independent software thread per core through duplication of hardware. A multithreaded or hyperthreaded processor core will multiplex a single core to execute multiple software threads through interleaving of software threads via hardware mechanisms. A computation that employs multiple software threads in parallel is called *thread parallel* [MRR12]. This type of parallelization is what we call *thread-level parallelization*.

Definition 1.2 (Thread) *A thread is any software unit of parallel work with an independent flow of control.*

Definition 1.3 (Multi-core Processor) *A multi-core processor is a single chip that contains multiple core processing units, more commonly known as cores.*

1.3.2 Task-level Parallelization

To use task-level parallelization, a programmer should program in tasks, not threads [Lee06]. Threads are a mechanism for executing tasks in parallel, and tasks are units of work that merely provide the opportunity for parallel execution; tasks are not themselves a mechanism of parallel execution [MRR12]. For a proper definition, see below.

Definition 1.4 (Task) *A task is a logical unit of potential parallelism with a separate flow of control.*

Tasks are executed by scheduling them onto software threads, which in turn the operating system schedules onto hardware threads. Scheduling of software threads onto hardware threads is usually preemptive (i.e., it can happen at any time). In contrast, scheduling of tasks onto software threads is typically non-preemptive (i.e., a thread switches tasks only at predictable switch points). Non-preemptive scheduling enables significantly lower overhead and stronger reasoning about space and time requirements than preemptive scheduling [JR13]. A computation that employs tasks over threads is called *task parallel*. This type of parallelization is what we call *task-level parallelization*. It is the preferred method of parallelism, especially for many-core processors.

Definition 1.5 (Many-core Processor) *A many-core processor is a specialized multi-core processor designed for a high degree of parallel processing, containing a large number of simpler, independent processor cores.*

In the task-level parallelization, the programmer should expose parallelism and share the opportunities for parallelization as *tasks*, but the work to map tasks to threads should not be encoded into an application. Hence, do not mix the concept of exposing tasks with the effort to allocate tasks to threads. The later causes inflexibility in scaling on different and future hardware. Hence, we are creating tasks and give the job of mapping tasks onto hardware to a parallel programming library, such as Threading Building Blocks (TBB) [Rei07] and Cilk Plus [Suk15].

The task-level parallelization is also tightly coupled with *parallel patterns*. A pattern is a recurring combination of data and task management, separate from any particular algorithm [MRR12]. The parallel patterns are not necessarily tied to any particular hardware architecture or programming language or system. Parallel patterns are essential for efficient computations of tasks.

Definition 1.6 (Parallel Pattern) *A parallel pattern is a recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design [MRR12].*

Parallel patterns are often composed with, or generalized from, a set of serial patterns. The serial patterns are the foundation of *structured programming*. The pattern-based approach to parallel programming can be considered an extension of the idea of structured programming [MRR12].

1.4 General Obstacles for Parallelization of MCTS

Since its inception, MCTS was the subject of parallelization, and several parallelization methods have been developed for it [CWvdH08a, CJ08, EM10, SKW10, SHM+16].

This trend comes from the fact that MCTS usually needs a large number of iterations to converge, and every iteration can be executed in parallel. Therefore, MCTS has sufficient potential for parallelization in theory, and it even seems to be straightforward. However, a closer look reveals that there are four obstacles to achieve parallelism: (1) irregular parallelism, (2) shared data structure, (3) data dependencies, and (4) complex interactions among obstacles. If we are not able to overcome these obstacles, the consequence will be four types of overhead, respectively: (1) load balancing overhead, (2) synchronization overhead, (3) search overhead, and (4) deployment overhead. In the following, we will explain these obstacles and what type of overhead they cause. Each of the subsections below introduces the necessary techniques for dealing with these obstacles.

1.4.1 Irregular Parallelism Causes Load Balancing Overhead

The first obstacle is irregular parallelism. Parallel algorithms with irregular parallelism suffer from a lack of load balancing over processing cores. MCTS constructs asymmetric search trees because the selection policy in MCTS allows the algorithm to favor more promising nodes (exploitation), leading to a tree with unbalanced branches over time [BPW+12]. Parallel execution of the algorithm with such a search tree results in *irregular parallelism* because one thread traverses a shorter branch while the other one works on a longer one. Chapter 4 provides more details and tries to handle this obstacle.

Definition 1.7 (Irregular Parallelism) *In irregular parallelism, the units of possible parallel work in this type of parallelism are dissimilar in a way that creates unpredictable dependencies.*

Definition 1.8 (Load Balancing) *Load balancing is a method used to allocate workloads uniformly across multiple computing resources, such as computing cores, to improve the distribution of workloads.*

1.4.2 Shared Data Structure Causes Synchronization Overhead

The second obstacle for parallelizing MCTS is a shared search tree. A parallel algorithm with a shared data structure suffers from *synchronization overhead* when it utilizes locks for data protection. Locks are notoriously bad for parallel performance because other threads have to wait until the lock is released. Moreover, locks are often a bottleneck when many threads try to acquire the same lock. The MCTS algorithm uses a tree data structure for storing the states of the domain and guiding the search process. The basic premise of a search tree in MCTS is relatively simple: (A) nodes are

added to the tree in the order in which they were expanded. (B) nodes are updated in the tree along with the order in which they were selected. In parallel MCTS, parallel threads are manipulating a shared search tree concurrently, and locks are required for data protection. It seems that we should have synchronization without using locks to avoid synchronization overhead. In Chapter 5, we show how we deal with this obstacle.

Definition 1.9 (Shared Data Structure) *A shared data structure, also known as a concurrent data structure, is a particular way of storing and organizing data that can be accessed by multiple threads simultaneously on a shared-memory machine.*

Definition 1.10 (Synchronization) *Synchronization is the coordination of tasks or threads to obtain the desired runtime order [Wil12].*

1.4.3 Ignoring Data Dependencies Causes Search Overhead

The third obstacle that should be addressed is the data dependencies. We find two types of data dependencies in MCTS: (1) the data dependency that exists among iterations and (2) the data dependency that exists among operations. The first type of data dependency exists because each of the iterations in the main loop of the algorithm requires the updated data which should be provided by its previous iterations. This type of data dependency is also known as *loop carried data dependencies*. Ignoring this type of data dependency causes *search overhead*. The second type of data dependency exists because each of the four operations inside each iteration of the algorithm depends on the data that is provided by the previous operation. Ignoring this type of data dependency is not possible for obvious reasons. Chapter 6 provides more details and our solution for overcoming this obstacle.

Definition 1.11 (Loop Carried Data Dependency) *A loop carried data dependency exists when a statement in one iteration of a loop depends in some way on a statement in a different iteration of the same loop.*

Definition 1.12 (Loop Independent Data Dependency) *A loop independent data dependency exists when a statement in one iteration of a loop depends only on a statement in the same iteration of the loop.*

Definition 1.13 (Search Overhead) *Search overhead exists in the MCTS algorithm when the number of nodes searched by a parallel algorithm is more than that of the serial algorithm.*

1.4.4 Complex Interactions Leading to Deployment Overhead

The fourth obstacle is the complexity of addressing the three above mentioned obstacles together. Trying to address all of them at once is difficult, due to the interactions among them. The overhead caused by complex interactions is called *deployment overhead*. The level of complexity forced the researchers to make compromises when solving some of these obstacles to have a parallel implementation of MCTS. In this research, we aim to mitigate the deployment overhead through structured parallel programming.

Definition 1.14 (Complex Interactions) *Complex interactions refer to the relationships among the general obstacles for parallelization of MCTS.*

Definition 1.15 (Deployment Overhead) *Deployment overhead is the amount of time spent to deploy an algorithm in a hardware environment.*

1.5 Performance and Scalability Studies

MCTS works by selectively building a tree, expanding only branches it deems worthwhile to explore [Cou06, AHH10, vdHPKV13]. The algorithm can converge to an optimal solution using a large number of playouts. It means that the algorithm requires more computation and memory to converge as the number of playouts increases. It leads to two distinct goals. The first and ultimate goal of parallelization is improving the *performance* of the parallelized application. The performance could be measured differently depending on the context in which it is used. In the context of MCTS, we measure performance by two different terms: (A) in terms of runtime (i.e., playout speedup), and (B) in terms of search quality (i.e., playing strength).

Definition 1.16 (Performance Study) *A performance study for the parallel MCTS algorithm on shared-memory systems examines where the performance of the parallelization of MCTS is guided by a certain number of cores and a certain amount of memory for one specific performance metric such as the number of Playouts per Second (PPS) or the Percentage of Wins (PW).*

Definition 1.17 (Playout Speedup) *Playout speedup is the improvement in the speed of execution.*

Definition 1.18 (Playing Strength) *Playing strength is the achieved performance compared to a standard rating.*

Adding more computing power and memory makes the process faster only if a scalable parallelization of the algorithm exists to harness the additional resources. Therefore, the second goal of parallelization is *scalability*. It will let the MCTS algorithm converge faster to a solution. By scalability, we mean that when we increase the number of cores and memory bandwidth, it results in improved performance in a manner proportional to the resources added. Being scalable is the main idea behind many parallelization methods for the MCTS algorithm on shared-memory machines [CWvdH08a, EM10, SKW10, SHM+16].

Definition 1.19 (Scalability Study) *A scalability study for parallel MCTS on shared-memory systems refers to how the performance of parallelization of MCTS changes given the increase of the number of cores and the amount of memory.*

Definition 1.20 (Memory Bandwidth) *Memory bandwidth is the rate at which data can be read from or stored into memory by a processor. Memory bandwidth is usually expressed in units of bytes per second.*

1.6 Scope and Research Goals

Our research handles and investigates parallel systems. To understand later design and implementation decisions as well as evaluation results, it is necessary to explain the scope in which the research is conducted.

Concerning the scope, we see that two major types of parallel architectures are prevailing in the industry: (A) shared-memory architecture and (B) distributed-memory architecture. Among these two principal types, the shared-memory architecture is of our main concern. Therefore, we concentrate on developing algorithms and finding solutions for shared-memory machines only. In passing, we remark that studies with a focus on distributed-memory systems [SP14, YKK+11] may benefit from our examinations, since our findings might be indirectly useful for the distributed-memory community of research. The explanation is that shared-memory machines are building blocks for distributed-memory systems.

The shared-memory architecture also has two types: (A1) Uniform Memory Access (UMA) and (A2) Non Uniform Memory Access (NUMA). We are interested in both of these architectures. In the UMA shared-memory architecture, each processor must use the same shared bus to access memory. Here we note that the access time remains the same despite which shared-memory module contains the data to be retrieved. The Phi co-processor has an UMA-based many-core architecture called Many Integrated Core (MIC). In the NUMA architecture, each processor has direct access to its local memory module. At the same time, it can also access any remote memory module

belonging to another processor using a shared interconnect network. The outcome of having many memory modules is that memory access time varies with the location of the data to be accessed. Each processor in a NUMA machine is multi-core. In the thesis, our goal is to work with both NUMA-based multi-core systems and UMA-based many-core systems for both the design and the implementation of the algorithms.

Definition 1.21 (Uniform Memory Access) *A Uniform Memory Access refers to a memory system in which the memory access time is uniform across all processors.*

Definition 1.22 (Many Integrated Core) *A Many Integrated Core is an UMA-based many-core architecture designed for highly parallel workloads. The architecture emphasizes higher core counts on a single die, and simpler cores, than on a traditional CPU.*

Definition 1.23 (Non Uniform Memory Access) *A Non Uniform Memory Access is a system in which certain banks of memory take longer to access than others, even though all the memory uses a single address space.*

1.7 Problem Statement and Research Questions

The MCTS algorithm is a good candidate for parallelization. This has been known since the introduction of the algorithm in 2006 [Cou06, KS06, EM10]. However, until now, the research community has only used thread-level parallelization when parallelizing the algorithm. The current parallel programming approaches are unstructured and do not use modern parallel programming patterns, languages, and libraries. We aim to address the complications of designing parallel algorithms for MCTS using the modern techniques, tools, and machines which are discussed above. We focus on both NUMA and MIC architectures to evaluate our implementations. Therefore, the Problem Statement (PS) of the thesis is as follows.

- ***PS***: How do we design a structured pattern-based parallel programming approach for efficient parallelism of MCTS for both multi-core and many-core shared-memory machines?

We define five specific research questions (RQs) derived from the ***PS*** that we try to answer in the following chapters. We will describe the five research questions below.

Thread-level parallelization: Until now, the research community has only used thread-level parallelization when parallelizing MCTS. However, today, the NUMA-based multi-core and UMA-based many-core architectures are very important. These are the architectures that we will use in our experiments. We believe that thread-level parallelization is not anymore a suitable method for the many-core processors.

Therefore, it is important to know the performance of the thread-level parallelization on the new architectures. It leads us to the first research question.

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

Task-level parallelization: One of the essential developments in parallel programming methods is the use of task-level parallelization. In task-level parallelization, calculations are partitioned into tasks, rather than spread over software threads. The use of task-level parallelization has three benefits: (1) it is conceptually simpler, (2) it may make the development of parallel MCTS programs easier, and (3) it leads to more efficient scheduling of CPU time. These benefits lead us to the second research question.

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

A lock-free data structure: MCTS requires a tree data structure. For efficient parallelism, this tree data structure must be lock-free. The existing lock-free tree data structure is inconsistent; i.e., it suffers from loss of information during the search phase. We are interested in developing a lock-free tree data structure for use in parallelized MCTS, in such a way that it avoids loss of information and simultaneously improves the speed of MCTS execution. This leads us to the third research question.

- ***RQ3***: *How can we design a correct lock-free tree data structure for parallelizing MCTS?*

Patterns for task-level parallelization: Task-level parallelization requires specific patterns by which the tasks are processed. Modern parallel libraries and languages support these patterns, thereby allowing quick construction of parallel programs that have these patterns. It may be possible to apply one or more patterns in the parallelization of MCTS. We are interested in (1) finding these patterns, and (2) using them in the parallelization of MCTS. This leads us to the fourth research question.

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

Improving search quality of MCTS: It has been shown that the parallelization of MCTS leads to a decrease in the quality of search results. Various solutions have been developed that attempt to mitigate this decrease in quality. We are interested in

finding out to what extent the existing solutions apply to the parallelized MCTS that we will develop in this thesis. This leads us to the fifth research question.

- ***RQ5***: *To what extent do the existing solutions which improve search quality, apply to our version of parallelized MCTS?*

By the existing solutions, we mean two methods: (1) ensemble methods, and (2) virtual loss.

1.8 Research Methodology

For answering a research question, our research methodology consists of four phases:

- The first phase is characterized by collecting knowledge on existing methods and algorithms. It is performed by reading to some extent, the existing literature and becoming familiar with the existing tools.
- The second phase is investigating the performance of the existing methods, tools, and techniques for parallelizing MCTS.
- The third phase is designing new ideas and algorithms. Then, the implementation of these designs takes place in a new software framework.
- In the fourth phase, an experiment is executed, and the results are collected, interpreted, analyzed, and reported.

1.9 Structure of the thesis

The problem statement and the five research questions introduced in Section 1.7 are addressed in eight chapters. Below we provide a brief description of the contents of each chapter.

Chapter 1 introduces the Monte Carlo Tree Search algorithm and defines the concepts of parallelism and parallelization. Then, it gives four general obstacles for parallelization of MCTS: load balancing, synchronization overhead, search overhead, and deployment overhead. After that, the chapter gives the definitions for performance and scalability and provides the scope of research. Then, it formulates the problem statement, five research questions, and the research methodology. Finally, it lists our contributions.

Chapter 2 provides the necessary background for the rest of the thesis. It discusses the benchmark problems, the parallelization methods for MCTS, the performance metrics, and our Upper Confidence Bounds for Trees (UCT) parallelization software package.

Chapter 3 answers ***RQ1***. The chapter provides, to the best of our knowledge, the first performance and scalability study of non-trivial MCTS programs on the Intel Xeon Phi.

Chapter 4 answers ***RQ2***. The chapter investigates how to parallelize irregular and unbalanced tasks in MCTS efficiently on the Xeon Phi.

Chapter 5 answers ***RQ3***. The chapter proposes a new lock-free tree data structure for parallel MCTS.

Chapter 6 answers ***RQ4***. The chapter proposes a new algorithm based on a *Pipeline Pattern* for Parallel MCTS.

Chapter 7 answers the first part of ***RQ5***. The chapter shows that balancing between the exploitation-exploration parameter and the tree size can be useful in Ensemble UCT to improve its performance.

Chapter 8 answers the second part of ***RQ5***. The chapter evaluates the benefit of using the virtual loss in lock-free (instead of locked-based) Tree Parallelization. Hence, it addresses the trade-off between search overhead and efficiency.

Chapter 9 concludes the thesis with a summary of the answers to what has been achieved with regards to the research questions and the problem statement, formulates conclusions, describes limitations and shows possible directions for future work.

1.10 Contributions

Below we list six contributions of our research. There are three main contributions (1 to 3) and three technical contributions (4 to 6).

1. The use of many-core machines for studying the performance and scalability of MCTS (see Chapter 2 and 3).
2. The use of task-level parallelization for MCTS (see Chapter 4).

3. The design of a lock-free data structure for parallel MCTS (see Chapter 5)
4. The introduction of a pipeline pattern for parallel MCTS (see Chapter 6).
5. We established a balance for the trade-off between exploitation-exploration for Root Parallelization (see Chapter 7).
6. By using lock-free parallelization, a virtual loss does not bring any improvement in search quality for a Horner Scheme (see Chapter 8).