Structured Parallel Programming for Monte Carlo Tree Search

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

***Abstract*—In this paper, we present a new algorithm for parallel Monte Carlo tree search (MCTS). It is based on the pipeline pattern and allows flexible management of the control flow of the operations in parallel MCTS. The pipeline pattern provides for the first structured parallel programming approach to MCTS. Moreover, we propose a new lock-free tree data structure for parallel MCTS which removes synchronization overhead. The Pipeline Pattern for Parallel MCTS algorithm (called 3PMCTS), scales very well to higher numbers of cores when compared to the existing methods.**

I. Introduction

In recent years there has been much interest in the Monte Carlo tree search (MCTS) algorithm, at that time a new, adaptive, randomized optimization algorithm [1], [2]. 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 [3]. The strength of the MCTS algorithm is that it provides answers with a random amount of error for any fixed computational budget [4]. 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 multicore machines (CPU) to large distributed-memory clusters. The emergence of the Intel Xeon Phi (Phi) co-processor with a large number (over 60) of simple cores has extended this spectrum with shared-memory manycore processors. Indeed, there is a full array of different parallel MCTS algorithms [5]–[10]. However, there is still no *structured parallel programming* approach, based on computation patterns, for MCTS. In this paper, we propose a new algorithm based on the *Pipeline Pattern* for Parallel MCTS, called 3PMCTS.

The standard MCTS algorithm has four operations inside its main loop (Figure 1). In this loop, the computation associated with each iteration is assumed to be independent. Existing parallel methods use *iteration-level* (IL) parallelism. They assign each iteration to a separate processing element (thread) for execution on separate processors [5], [8], [9]. All three publications are facing a bottleneck in their implementation since they can not partition the iteration into constituent parts (operations) for parallel execution. Close analysis has learned us that the loop can actually be decomposed into separate operations for parallelization. Therefore, in our new design we introduce *operation-level* (OL) parallelism. The main idea is that the 3PMCTS algorithm assigns each operation to a separate processing element for execution by separate processors. This leads to flexibility in managing the control flow of operations in the MCTS algorithm.

Our approach of applying structured parallel programming focuses the attention on (1) a design issue and (2) an implementation issue. For the design issue, we describe how the pipeline pattern is used as a building block in the design of 3PMCTS. It consists of a precise arrangement of tasks and data dependencies in MCTS. For the implementation, it is important to measure the performance of 3PMCTS on real machines. We present the implementation of 3PMCT by using Threading Building Blocks (TBB) [11] and we measure its performance on CPU and Phi.[1] This paper has three contributions:

1) A new structured algorithm based on the pipeline pattern is introduced for parallel MCTS.
2) A new lock-free tree data structure for parallel MCTS is presented.
3) A new TBB implementation based on the concept of *token* is proposed. The experimental results show that our implementation scales well.

The remainder of the paper is organized as follows. In section II the required background information is briefly described. Section III provides necessary definitions and explanations for the design issue of structured parallel programming for 3PMCTS. Section IV gives the explanations for the implementation issue of the 3PMCTS algorithm. Section V provides the proposed lock-free tree data structure, Section VI the experimental setup of the 3PMCTS, and Section VII the experimental results for 3PMCTS. Section VIII discusses related work. Finally, in Section IX we conclude the paper.

II. Background

Below we discuss in II-A MCTS, in II-B tree parallelization, and in II-C TBB.

[1] We also discuss two elements related to the implementation of 3PMCTS, including the concept of *token* and a new lock-free tree data structure. A lock-free tree data structure plays a critical role in any parallel implementation for MCTS to be scalable.

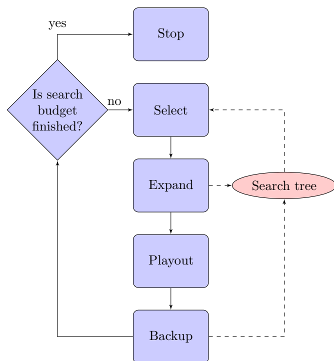


Fig. 1. Flowchart of the sequential MCTS.

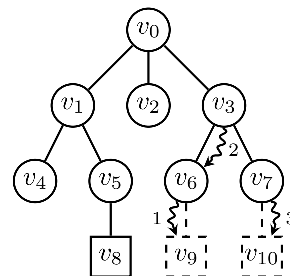


Fig. 2. Tree parallelization without locks. The curly arrows represent threads. The rectangles are terminal leaf nodes.

A. MCTS

Figure 1 shows a flowchart of MCTS [1]. 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.

1) SELECT: Starting at the root node, a path of nodes inside the search tree is selected until a non-terminal node with unvisited children is reached. Each of the nodes is selected based on a *selection policy*. Among the proposed selection policies, the Upper Confidence Bounds for Trees (UCT) is one of the most commonly used policies [2], [12]. A child node j is selected to maximize:

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

where $\overline{X}_j = \frac{w_j}{n_j}$ is the average reward from child j, w_j is the reward value of child j, n_j is the number of times child j has been visited, n is the number of times the current node has been visited, and $C_p \geq 0$ is a constant. The first term in the UCT equation is for *exploitation* of known parts of the tree and the second term is for *exploration* of unknown parts [12]. The level of exploration of the UCT algorithm can be adjusted by tuning the C_p constant.
2) EXPAND: One of the children of the selected non-terminal node is generated and added to the selected path.
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 state space is reached. The terminal state is evaluated to produce a reward value Δ.
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 [12].

As soon as the computational budget is exhausted, the best child of the root node is returned.

B. Tree Parallelization

In tree parallelization, one search tree is shared among several threads that are performing simultaneous searches [5]. The main challenge in this method is the prevention of data corruption. A lock-based method uses fine-grained locks to protect shared data. However, this approach suffers from synchronization overhead due to thread contentions and does not scale well [5]. A lock-free implementation of addresses the problem and scales better [13]. However, the method in [13] does not guarantee the computational consistency of the multi-threaded program with the single-threaded program. Figure 2 shows the tree parallelization without locks.

C. TBB

TBB is a C++ template library developed by Intel for writing software programs that take advantage of a multicore processor [14]. The TBB implementation of pipelines uses a technique that enables greedy scheduling, but the greed must be constrained to bound memory consumption. The user specifies the constraint as a maximum number of items allowed to flow simultaneously through the pipeline [14].

III. DESIGN OF 3PMCTS

In this section, we describe our structured parallel programming approach for MCTS. In section III-A we explain how to decompose MCTS into tasks. In section III-B we investigate what types of data dependencies exist among these tasks. Section III-C describes how the pipeline pattern is applied in MCTS. Finally, section III-D provides our design for the 3PMCTS algorithm.

A. Decomposition into Tasks

The first step towards designing our 3PMCTS algorithm is to find concurrent tasks in MCTS. There are two levels of **task decomposition** in MCTS.

1) *Iteration-level (IL) tasks*: In MCTS the computation associated with each SELECT-EXPAND-PLAYOUT-BACKUP-iteration is independent. Therefore, *these are candidates* to guide a task decomposition by mapping each iteration onto a task.

2) *Operation-level (OL) tasks*: The task decomposition for MCTS occurs inside each iteration. Each of the four MCTS steps can be treated as a separate task.

B. Data Dependencies

In 3PMCTS, a search tree is shared among multiple parallel tasks. Therefore, there are two levels of **data dependency**.

1) *Iteration-level (IL) dependencies*: Strictly speaking, in MCTS, iteration j has a *soft dependency* to its predecessor iteration $j-1$. Obviously, to select an optimal path, it requires updates on the search tree from the previous iteration.[2] A parallel MCTS can ignore IL dependencies and simply suffers from the *search overhead*.[3]
2) *Operation-level (OL) dependencies*: Each of the four operations in MCTS has a *hard dependency* to its predecessor.[4] For example, the expansion of a path cannot start until the selection operation has been completed.

C. Pipeline Pattern

In this section, we focus on the pipeline pattern in MCTS using OL tasks. The pipeline pattern is the most straightforward way to enforce the required ordering among the OL tasks. Below we explain two possible types of pipelines for MCTS.

1) *Linear pipeline*: Figure 3a shows a linear MCTS pipeline with the selected paths inside the search tree; from one stage to the next stage buffers are given a path as operations are completed.
2) *Non-linear pipeline*: Figure 3b shows a non-linear MCTS pipeline with two parallel PLAYOUT stages. Both of the PLAYOUT stages take paths produced by the EXPAND stage of the pipeline.

D. Parallelism of a Pipeline

The existing parallel methods such as tree parallelization use IL tasks. There are only IL dependencies when performing IL parallelism. The potential concurrency is exploited by assigning each of the IL tasks to a separate processing element and having them work on separate processors. So far IL parallelism is investigated quite extensively [5], [8]–[10], [13].

In contrast to the existing methods, our 3PMCTS algorithm uses OL tasks which have both IL and OL dependencies. The OL tasks are assigned to the stages of a pipeline. The pipeline pattern can satisfy the OL dependencies among the OL tasks. 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 is *linear* then the scalability is limited to the number of stages.[5] However in MCTS, the operations are not equally computationally intensive, e.g., generally the PLAYOUT operation

[2]i.e., a violation of IL dependency does not impact the correctness of the algorithm.

[3]Occurs when a parallel implementation in a search algorithm searches more nodes of the search space than the sequential version; for example, since the information to guide the search is not yet available.

[4]i.e., a violation of OL dependency yields an incorrect algorithm.

[5]When the operations performed by the various stages are all about equally computationally intensive.

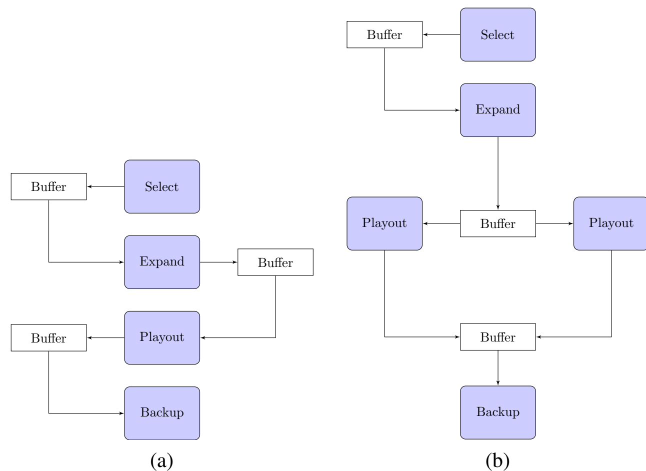


Fig. 3. (3a) Flowchart of a linear pipeline for MCTS. (3b) Flowchart of a nonlinear pipeline for MCTS.

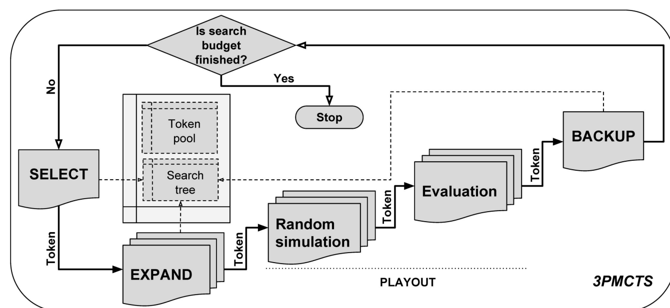


Fig. 4. The 3PMCTS algorithm with a five-stage non-linear pipeline.

(random simulations plus evaluation of a terminal state) is more computationally expensive than other operations. Therefore, our 3PMCTS algorithm uses a *non-linear* pipeline with parallel stages. Introducing parallel stages makes 3PMCTS more scalable. The 3PMCTS algorithm, depicted in Figure 4, has three parallel stages (i.e., EXPAND, *Random Simulation*, and *Evaluation*). It will be usable in almost any sufficiently powerful parallel programming model (e.g., TBB [14] or Cilk [15]).

IV. IMPLEMENTATION OF 3PMCTS

In this section, we introduce the implementation of our 3PMCTS algorithm. In section IV-A we present the concept of *token* (when used as type name, we write *Token*). Section IV-B describes the implementation of 3PMCTS using TBB.

A. Token

A token represents a path inside the search tree during the search. Algorithm 1 presents definition for the type *Token*. It has four fields. (1) id represents a unique identifier for a token, (2) v represents the current node in the tree, (3) s represents the search state of the current node, and (4) Δ represents the reward value of the state. In our implementation for 3PMCTS, each stage (task) performs its operation on a token. We can also specify the number of in-flight tokens.

Algorithm 1: Type definition for token.

```
1  type
2      type id : int;
3      type v : Node*;
4      type s : State*;
5      type Δ : int;
6  Token;
```

Algorithm 2: Serial implementation of MCTS, with stages SELECT, EXPAND, PLAYOUT, and BACKUP.

```
1  Function UCTSEARCH(s0)
2      v0 = create root node with state s0;
3      t0.s = s0;
4      t0.v = v0;
5      while within search budget do
6          tl = SELECT(t0);
7          tl = EXPAND(tl);
8          tl = PLAYOUT(tl);
9          BACKUP(tl);
10     end
```

B. TBB Implementation

The pseudocode of MCTS is shown in Algorithm 2. A data structure of type *State* describes the search state of the current node in the tree and a data structure of type *Node* shows the current node being searched inside the tree. The functions of the MCTS algorithm are defined in Algorithm 3. Each function constitutes a stage of the non-linear pipeline in 3PMCTS. There are two approaches for parallel implementation of a non-linear pipeline [16]:

- *Bind-to-stage*: A processing element (e.g., thread) is bound to a stage and processes tokens as they arrive. If the stage is parallel, it may have multiple processing elements bound to it.
- *Bind-to-item*: A processing element is bound to a token and carries the token through the pipeline. When the processing element completes the last stage, it goes to the first stage to select another token.

Our implementation for 3PMCTS algorithm is based on a bind-to-item approach. Figure 4 depicts a five-stage pipeline for 3PMCTS that can be implemented using TBB *tbb::parallel_pipeline* template [14]. The five stages run the functions SELECT, EXPAND, *RandomSimulation*, *Evaluation*, and BACKUP, in that order. The first (SELECT) and last stage (BACKUP) are serial in-order; They process one token at a time. The three middle stages (EXPAND, *RandomSimulation*, and *Evaluation*) are parallel and do the most time-consuming part of the search. The *Evaluation* and *RandomSimulation* functions are extracted out of the PLAYOUT function to achieve more parallelism. The serial version uses a single token. The 3PMCTS algorithm aims to search multiple paths in parallel. Therefore, it needs more than one in-flight *token*.

Algorithm 3: The functions of the MCTS algorithm.

```
1  Function SELECT(Token t) : <Token>
2      while t.v →IsFullyExpanded() do
3          t.v :=   argmax        v'.UCT(Cp);
                 v'∈childrenofv
4          t.s →SetMove(t.v → move);
5      end
6      return t;
7  Function EXPAND(Token t) : <Token>
8      if !(t.s →IsTerminal()) then
9          moves := t.s →UntriedMoves();
10         shuffle moves uniformly at random;
11         t.v →Init(moves);
12         v' := t.v →AddChild();
13         if t.v ≠ v' then
14             t.v :=v';
15             t.s →SetMove(v' → move);
16         end
17     end
18     return t;
19 Function RandomSimulation(Token t)
20     moves := t.s →UntriedMoves();
21     shuffle moves uniformly at random;
22     while !(t.s →IsTerminal()) do
23         choose new move ∈ moves;
24         t.s →SetMove(move);
25     end
26     return t
27 Function Evaluation(Token t)
28     t.Δ := t.s → Evalute();
29     return t
30 Function BACKUP(Token t) : void
31     while t.v ≠ null do
32         t.v → Update(t.Δ);
33         t.v := t.v → parent;
34     end
```

Figure 5 shows the key parts of the TBB code with the syntactic details for the 3PMCTS algorithm.

V. Lock-free Search Tree

In this section, we provide our new lock-free tree search. A potential bottleneck in a parallel implementation is the *race condition*. A race condition occurs when concurrent threads perform operations on the same memory location without proper synchronization, and one of the memory operations is a write [16]. Consider the example search tree in Figure 6a. Three parallel threads attempt to perform MCTS operations on the shared search tree. There are three race condition scenarios.

1) Shared Expansion (SE): Figure 6b shows two threads (1 and 2) concurrently performing EXPAND(v_6). In this SE scenario, synchronization is required. Obviously, a

```
void 3PMCTS(tokenlimit){
...
/* The routine tbb::parallel_pipeline takes two parameters.
(1) A token limit. It is an upper bound on the number of tokens that are processed simultaneously.
(2) A pipeline. Each stage is created by function tbb::make_filter. The template arguments to
make_filter indicate the type of input and output items for the filter. The first ordinary argument
specifies whether the stage is parallel or not and the second ordinary argument specifies a function
that maps the input item to the output item.
*/
    tbb::parallel_pipeline(tokenlimit,
       /* The SELECT stage is serial and mapping a special object of type tbb::flow_control, used
       to signal the end of the search, to an output token. */
       tbb::make_filter<void, Token*>(tbb::filter::serial_in_order,[&](tbb::flow_control & fc)->Token*
    {
            /* A circular buffer is used to minimize the overhead of allocating and freeing tokens
            passed between pipeline stages (it reduces the communication overhead). */
            Token* t = tokenpool[index];
            index = (index+1) % tokenlimit;
            if (within the search budget) {
                /* Invocation of the method stop() tells the tbb::parallel_pipeline that no more
                paths will be selected and that the value returned from the function should be ignored.
     */
                fc.stop();
                return NULL;
            } else {
                t = SELECT(t);
                return t
            }
        }
        ) &
        // The EXPAND stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
             return EXPAND(t);
        }) &
        // The RandomSimulation stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
             return RandomSimulation(t);
        }) &
        // The Evaluation stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
             retun Evaluation(t);
        }) &
        /* The BACKUP stage has an output type of void since it is only consuming tokens,
        not mapping them.
        */
        tbb::make_filter<Token*, void>(tbb::filter::serial_in_order,[&](Token * t){
             return BACKUP(t);
        })
    );
...
}
```

.

Fig. 5. An implementation of the 3PMCTS algorithm in TBB.

race condition exists if two parallel threads intend to initialize the list of children in a node simultaneously. In such an SE race, the list of children for a selected node should be created only once. Enzenberger et al. assign to each thread an own memory array for creating a list of new children [17]. Only after the children are fully created and initialized, they are linked to the parent node. Of course, this causes memory overhead. What usually happens is the following. If several threads expand the same node, only the children created by the last thread will be used in future simulations. It can also happen that some of the children that are lost already received some updates; these updates will also be lost.

2) Shared Backup(SB): Figure 6c shows two threads (1 and 3) concurrently performing BACKUP(v_3). In the SB scenario, synchronization is required because it is a race condition that parallel threads update the value of w and n in a node simultaneously. Enzenberger et. al ignore these race conditions. They accept the possible faulty updates and the inconsistency of parallel computation.
3) Shared Backup and Selection (SBS): Figure 6d shows thread 2 performing BACKUP(v_3) and thread 3 performing SELECT(v_3). In the SBS scenario, synchronization is required because it is a race condition in which a

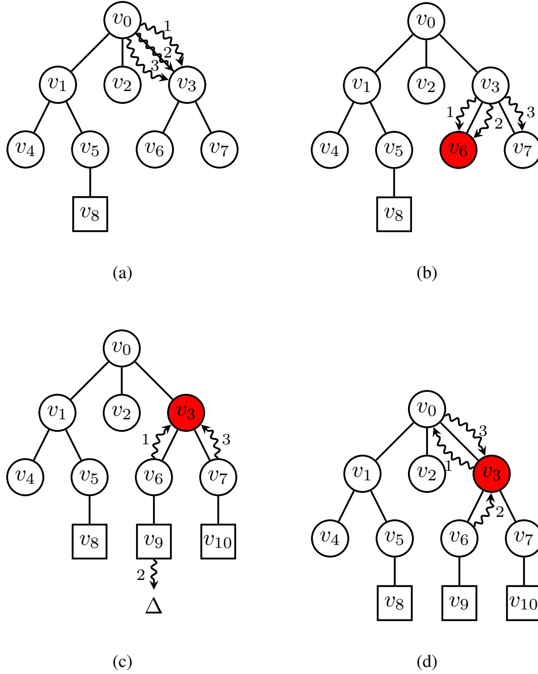


Fig. 6. (6a) The initial search tree. The internal and non-terminal leaf nodes are circles. The terminal leaf nodes are squares. (6b) Thread 1 and 2 are expanding node v_6. (6c) Thread 1 and 2 are updating node v_3. (6d) Thread 1 is selecting node v_3 while thread 2 is updating this node.

thread reads the value of w, and before reading the value of n, another thread updates the value of w and n. In this case, the first thread reads inconsistent values. Enzenberger et al. ignore these race conditions. They accept the possible faulty updates and the inconsistency of parallel computation.

Algorithm 4 shows our new lock-free tree data structure of type *Node* for MCTS. Our lock-free tree data structure uses the new multithreading-aware memory model of the C++11 Standard. In order to avoid the race conditions, the ordering between the memory accesses in the threads has to be enforced [18]. In our lock-free implementation, we use the synchronization properties of *atomic* operations to enforce an ordering between the accesses. We have used the atomic variants of the built-in types (i.e., *atomic_int* and *atomic_bool*); they are lock-free on most popular platforms. The standard atomic types have different member functions such as *load()*, *store()*, *exchange()*, *fetch_add()*, and *fetch_sub()*. The member function *store()* is a store operation, whereas the *load()* is a load operation. The *exchange()* member function replaces the stored value in the atomic variable with a new value and automatically retrieves the original value. We use two memory models for the memory-ordering option for all operations on atomic types: *sequentially consistent* ordering (*memory_order_seq_cst*) and *acquire_release* ordering (*memory_order_acquire* and *memory_order_release*). The *sequentially consistent* ordering implies that the behavior of a multithreaded program is consistent with a single threaded program. In the *acquire_release* ordering model, *load()* is an *acquire* operation, *store()* is a release operation, *exchange()* or *fetch_add()* or *fetch_sub()* are either *acquire*, *release* or both (*memory_order_acq_rel*). We have solved all the three above cases of race conditions (SE,SB, and SBS) using these two memory models and the atomic operations.

1) A node has an $isparent$ flag member. This flag indicates whether the list of children is created or not. The *isparent* flag is initially set to $false$. To change the state of the node to be a parent, we set its $isparent$ to $true$. Before an EXPAND adds a child to a node, it must create the list of children for the node and set the $isparent$ to $true$. After a node successfully becomes a parent, one of the unexpanded children in this list can be added to the node. It prevents the problem in EE that the list of children is created by two threads at the same time. Thus, the key steps in the EXPAND operation are as follows: (A, see Algorithm 4) change $v_6.isparent$ to $true$ (i.e., no other thread can enter), (B) create the list of children for v_6, (C) set the value of $v_6.untriedmoves$, (D) set the value of $v_6.isexpandable$ to $true$ (D1) and load the value of $v_6.isexpandable$ (D2), and (E) $untriedmoves$ is used as a $count$ of the number of items in the list of $children$.
2) In the SB and SBS race conditions, we use atomic types for variables w and n. The thread accesses to these variables (reads (F1,F2) and writes (G1,G2)) are *sequentially consistent*. This memory model preserve the order of operations in all threads. Therefore we have no faulty updates and guarantee consistency of computation.

VI. EXPERIMENTAL SETUP

The performance of 3PMCTS is measured by using a High Energy Physics (HEP) expression simplification problem [3]. Our setup follows closely [3]. We discuses in VI-A the case study, in VI-B the hardware, and in VI-C the performance metrics.

A. Case Study

Horner's rule 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

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0, \tag{2}$$

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

$$p(x) = ((a_n x + a_{n-1})x + ...)x + a_0. \tag{3}$$

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. Here, Eq. 3 applies on a polynomial for each variable, treating the other variables as constants. The order of choosing

Algorithm 4: Type definition for a lock-free tree data structure.

```
1  type
2    type move : int;
3    type w : atomic_int;
4    type n : atomic_int;
5    type isparent := false : atomic_bool;
6    type untriedmoves := −1 : atomic_int;
7    type isexpandable := false : atomic_bool;
8    type isfullyexpanded := false : atomic_bool;
9    type parent : Node*;
10   type children : Node*[];
11   Function Init(moves) : <void>
12     int nomoves = moves.size();
13     if !(isparent.exchange(true));                                  ▷ [see A]
14      then
15       initialize list of children using moves;                      ▷ [see B]
16       untriedmoves.store(nomoves);                                  ▷ [see C]
17       isexpandable.store(true,memory_order_release);                ▷ [see D1]
18     end
19   Function AddChild() : <Node*>
20     int index;
21     if isexpandable.load(memory_order_acquire);                     ▷ [see D2]
22      then
23       if (index := undriedmoves.fetch_sub(1)) = 0;                  ▷ [see E]
24        then
25         isfullyexpanded.store(true);
26       end
27       if index < 0 then
28         return current node;
29       else
30         return children[index];
31       end
32     else
33       return current node;
34     end
35   Function IsFullyExpanded() : <bool>
36     return isfullyexpanded.load();
37   Function UCT(C_p) : <float>
38     w' := w.load(memory_order_seq_cst);                             ▷ [see F1]
39     n' := n.load(memory_order_seq_cst);                             ▷ [see F2]
40     n" := parent → n.load(memory_order_seq_cst);
41     return w'/n' + C_p * sqrt(ln(n")/n')
42   Function Update(Δ) : <void>
43     w.fetch_add(Δ,memory_order_seq_cst);                            ▷ [see G1]
44     n.fetch_add(1,memory_order_seq_cst);                            ▷ [see G2]
45 Node;
```

variables may be different, each order of the variables is called a *Horner scheme*.

The number of operations can be reduced even more by performing common subexpression elimination (CSE) after transforming a polynomial with Horner's rule. CSE creates new symbols for each subexpression that appears twice or more and replaces them inside the polynomial. Then, the subexpression has to be computed only once.

We are using the HEP(σ) expression with 15 variables to study the results of 3PMCTS. The MCTS is used to find an order of the variables that gives efficient Horner schemes [3]. The root node has n children, with n the number of

variables. The children of other nodes represent the remaining unchosen variables in the 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 (*RandomSimulation*). This complete order is then used for Horners method followed by CSE (*Evaluation*). The number of operations in this optimized expression is counted (Δ).

B. 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.

C. Performance Metrics

The primary goal of parallelization is performance. There are two important metrics related to performance and parallelism for MCTS.

1) Playout Speedup (PS): If we have a fixed number of playouts seen as the search budget, then

$$PS = \frac{\text{time in sequential}}{\text{time in parallel}}. \quad (4)$$

2) Search Overhead (SO): If we have to find a desired optimal point in the search space, then

$$SO = \frac{\text{required \# of playouts in parallel}}{\text{required \# of playouts in sequential}} - 1. \quad (5)$$

If the parallel MCTS algorithm expands more nodes (i.e., do more playouts) than the equivalent serial algorithm to solve a problem, then there is SO.

In this paper, we use playout-speedup to report the performance.

VII. Experimental Results

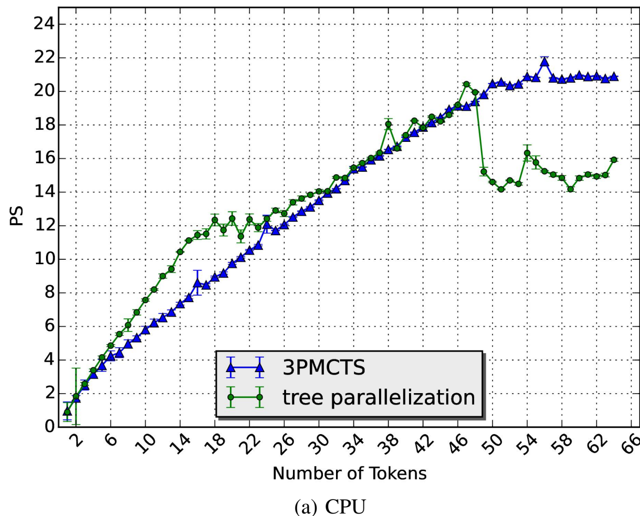
In this section, the performance of 3PMCTS is measured.

1) Playout-speedup for CPU: The graph in Figure 7a shows the playout-speedup for both 3PMCTS and tree parallelization, as a function of the number of tokens on CPU. Both 3PMCTS and tree parallelization are doing 1024 playouts. We see a playout-speedup for 3PMCTS close to 22 for 56 tokens. A playout-speedup close to 21 is observed for tree parallelization for 47 tasks. After 48 tasks the playout-speedup for tree parallelization drops (it is run on a machine with 48 hyperthreads) while the performance of 3PMCTS continues to grow until it becomes stable.
2) Playout-speedup for Phi: The graph in Figure 7b shows the playout-speedup for both 3PMCTS and tree parallelization, as a function of the number of tokens on Phi. Both 3PMCTS and tree parallelization are doing 1024 playouts. We see a playout-speedup for 3PMCTS close to 42 for 128 tokens. A playout-speedup close to 36 is observed for tree parallelization for 128 tasks.

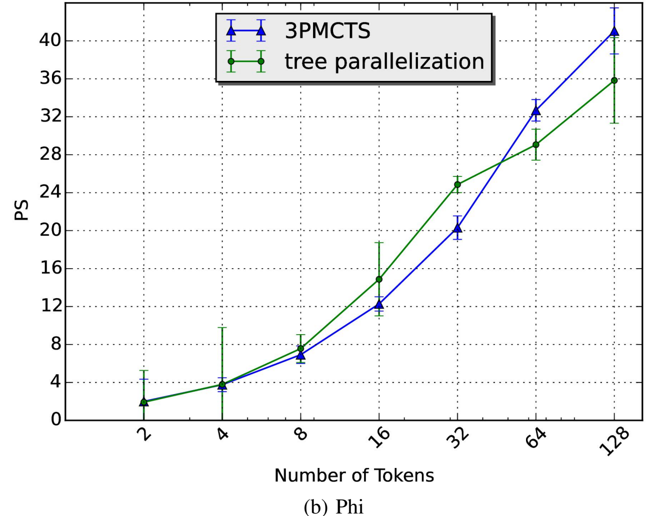
From these results, we may provisionally conclude that the 3PMCTS algorithm shows a playout-speedup as good as tree parallelization, for a well-studied problem. It allows fine-grained managing of the control flow of operations in MCTS in contrast to tree parallelization.

VIII. Related Work

Below we review related work on MCTS parallelizations. The two major parallelization methods for MCTS are root parallelization and tree parallelization [5]. There exist also less



(a) CPU



(b) Phi

Fig. 7. Playout-speedup as function of the number of tokens. Each data point is an average of 10 runs. The constant C_p is 0.1. The search budget is 1024 playouts.

frequently encountered techniques, such as leaf parallelization [5] and approaches based on transposition table driven work scheduling (TDS) [6], [19].

1) Tree parallelization: For shared memory machines, tree parallelization is a suitable method. It is used in FUEGO, a well-known open source Go program [17]. In tree parallelization one MCTS tree is shared among several threads that are performing simultaneous searches [5]. It is shown in [5] that the playout-speedup of tree parallelization with *virtual loss* does not scale perfectly up to 16 threads. The main challenge is the use of locks to prevent data corruption.
2) Root parallelization: Chaslot et al. [5] report that root parallelization shows perfect playout-speedup up to 16 threads. Soejima et al. [20] analyzed the performance of root parallelization in detail. They revealed that a Go program that uses lock-free tree parallelization with 4 to 8 threads outperformed the same program with root parallelization that utilized 64 distributed CPU cores. This result suggests the superiority of tree parallelization over root parallelization in shared memory machines. Fern and Lewis [7] thoroughly investigated an Ensemble UCT approach in which multiple instances of UCT were run independently. Their root statistics were combined to yield the final result, showing that Ensembles can significantly improve performance per unit time in a parallel model. This is also shown in [10].

IX. CONCLUSION AND FUTURE WORK

Monte Carlo Tree Search (MCTS) is a randomized algorithm that is successful in a wide range of optimization problems. The main loop in MCTS consists of individual iterations, suggesting that the algorithm is well suited for parallelization. The existing parallelization methods, e.g., tree parallelization, simply fans out the iterations over available cores.

In this paper, a structured parallel programming approach is used to develop a new parallel algorithm based on the pipeline pattern for MCTS. The idea is to break-up the iterations themselves, splitting them into individual operations, which are then parallelized in a pipeline. Experiments with an application from High Energy Physics show that our implementation of 3PMCTS scales well. Scalability is only one issue, although it is an important one. The second issue is flexibility of task decomposition in parallelism. These flexibilities allow fine-grained managing of the control flow of operations in MCTS. We consider the flexibility an even more important characteristic of 3PMCTS.

We may conclude the following. Our new method is highly suitable for heterogeneous computing because it is possible that some of the MCTS operations might not be suitable for running on a target processor, whereas others are. Our 3PMCTS algorithm gives us full flexibility for offloading a variety of different operations of MCTS to a target processor. For future work, we will study the implementation of 3PMCTS on a heterogeneous machine.

ACKNOWLEDGMENT

This work is supported in part by the ERC Advanced Grant no. 320651, HEPGAME.

REFERENCES

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proceedings of the 5th International Conference on Computers and Games*, ser. CG'06, vol. 4630. Springer-Verlag, may 2006, pp. 72–83.
[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning Levente," in *ECML'06 Proceedings of the 17th European conference on Machine Learning*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer Berlin Heidelberg, sep 2006, pp. 282–293.
[3] J. Kuipers, A. Plaat, J. Vermaseren, and J. van den Herik, "Improving Multivariate Horner Schemes with Monte Carlo Tree Search," *Computer Physics Communications*, vol. 184, no. 11, pp. 2391–2395, nov 2013.
[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning Series. MIT Press, 2016. [Online]. Available: https://books.google.nl/books?id=Np9SDQAAQBAJ
[5] G. Chaslot, M. Winands, and J. van den Herik, "Parallel Monte-Carlo Tree Search," in *the 6th Internatioal Conference on Computers and Games*, vol. 5131. Springer Berlin Heidelberg, 2008, pp. 60–71.
[6] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa, "Scalable Distributed Monte-Carlo Tree Search," in *Fourth Annual Symposium on Combinatorial Search*, may 2011, pp. 180–187.
[7] A. Fern and P. Lewis, "Ensemble Monte-Carlo Planning: An Empirical Study." in *ICAPS*, 2011, pp. 58–65.
[8] L. Schaefers and M. Platzner, "Distributed Monte-Carlo Tree Search : A Novel Technique and its Application to Computer Go," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 3, pp. 1–15, 2014.
[9] S. A. Mirsoleimani, A. Plaat, J. van den Herik, and J. Vermaseren, "Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors," in *ISPA 2015 : The 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Helsinki, 2015, pp. 77–83.
[10] ——, "Scaling Monte Carlo Tree Search on Intel Xeon Phi," in *Parallel and Distributed Systems (ICPADS), 2015 20th IEEE International Conference on*, 2015, pp. 666–673.
[11] "Intel threading building blocks TBB." [Online]. Available: https://www.threadingbuildingblocks.org
[12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.
[13] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," *Advances in Computer Games*, vol. 6048, pp. 14–20, 2010.
[14] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.
[15] C. E. Leiserson and A. Plaat, "Programming Parallel Applications in Cilk," *SINEWS: SIAM News*, vol. 31, no. 4, pp. 6–7, 1998.
[16] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.
[17] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "FuegoAn Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 259–270, dec 2010.
[18] A. Williams, *C++ Concurrency in Action: Practical Multithreading*, ser. Manning Pubs Co Series. Manning, 2012. [Online]. Available: https://books.google.nl/books?id=EttPPgAACAAJ
[19] J. Romein, A. Plaat, H. E. Bal, and J. Schaeffer, "Transposition Table Driven Work Scheduling in Distributed Search," in *In 16th National Conference on Artificial Intelligence (AAAI'99)*, 1999, pp. 725–731.
[20] Y. Soejima, A. Kishimoto, and O. Watanabe, "Evaluating Root Parallelization in Go," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 278–287, dec 2010.