Universiteit
Leiden
The Netherlands

# Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

**Citation**

Mirsoleimani, S. A. (2020, June 17). *Structured parallel programming for Monte Carlo Tree Search*. *SIKS Dissertation Series*. Retrieved from https://hdl.handle.net/1887/119358

| | |
|---|---|
| Version: | Publisher's Version |
| License: | Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden |
| Downloaded from: | https://hdl.handle.net/1887/119358 |

**Note:** To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

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

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

# Structured Parallel Programming

FOR

# Monte Carlo Tree Search

S. Ali Mirsoleimani

# Structured Parallel Programming

**FOR**

# Monte Carlo Tree Search

## Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof. mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 17 juni 2020
klokke 11.15 uur

door

**Sayyed Ali Mirsoleimani**
geboren te Abadeh, Iran
in 1986

**Promotoren**:
Prof. dr. H. J. van den Herik
Prof. dr. A. Plaat
**Copromotor**:
Dr. J. A. M. Vermaseren Nikhef

**Promotiecommissie**:
Prof. dr. P. J. G. Mulders Vrije Universiteit Amsterdam
Prof. dr. F. J. Verbeek
Prof. dr. H. A. G. Wijshoff
Dr. F. Khunjush Shiraz University, Shiraz, Iran
Dr. W. A. Kosters
Dr. ir. A. L. Varbanescu Universiteit van Amsterdam

HEPGAME, ERC Advanced Grant No. 320651
The research reported in this thesis has been additionally funded by Nikhef, the Nationaal instituut voor subatomaire fysica.

In the first year, the research reported in this thesis has been performed at Tilburg center for Cognition and Communication (TiCC) at Tilburg University, the Netherlands.

The research reported in this thesis has been completed at Leiden Centre of Data Science (LCDS) hosted by Leiden Institute of Advanced Computer Science (LIACS) at the Faculty of Science, Universiteit Leiden, the Netherlands.

SIKS Dissertation Series No. 2020-08
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.



Copyright © 2020 by S.A. Mirsoleimani

An electronic version of this dissertation is available at
`http://openaccess.leidenuniv.nl/`.

I would like to dedicate this thesis to my wife Elahe and to my parents,
for all of their love and support.

In loving memory of my grandfathers,
Bahram and Abolghasem

# Preface

The thesis is part of a bigger project, the HEPGAME (High Energy Physics Game). The project started in 2011 when Jos Vermaseren developed the first ideas on improving FORM at Nikhef, Amsterdam. In 2012 he submitted an ERC advanced research grant together with Tilburg University. It was accepted on 12/12/2012. Half a year later in July 2013, the program started. The main objective for HEPGAME was the utilization of AI solutions, particularly by using MCTS for simplification of HEP calculations. One of the issues is solving mathematical expressions of interest with millions of terms. Up to 2011, these calculations were executed with the FORM program, which is software for symbolic manipulation. These calculations are computationally intensive and take a large amount of time. Hence, the FORM program was parallelized to solve large equations in a reasonable amount of time. Therefore, any new algorithm, for instance, the ones based on MCTS, should also be parallelized. Here our research comes in. It is dedicated to parallelization of MCTS on multi-core and many-core processors. The research was ambitious and challenging. Therefore, we divided the research area into three main parts: (1) the evaluation of current methods for parallelization of MCTS, (2) addressing the shortcomings in these methods, and (3) providing new ways of parallelization for MCTS. In the first part, we investigated the current methods and evaluated them in terms of performance and scalability on both multi-core and manycore processors. In the second part, we examined how we can solve the actual shortcomings in the existing parallelization methods for MCTS. The third part was dedicated to finding new ideas, methods, and ways beyond the existing ones to parallelize MCTS.

*Sayyed Ali Mirsoleimani, Leiden, July 2019*

# Contents

**Preface** **vii**

**Contents** **ix**

**List of Definitions** **xv**

**List of Figures** **xvii**

**List of Tables** **xxi**

**List of Listings** **xxiii**

**List of Abbreviations** **xxv**

**1 Introduction** **1**
1.1 HEPGAME . . . . . 2
1.2 Monte Carlo Tree Search . . . . . 2
1.3 Parallelism and Parallelization . . . . . 3
1.3.1 Thread-level Parallelization . . . . . 4
1.3.2 Task-level Parallelization . . . . . 4
1.4 General Obstacles for Parallelization of MCTS . . . . . 5
1.4.1 Irregular Parallelism Causes Load Balancing Overhead . . . . . 6
1.4.2 Shared Data Structure Causes Synchronization Overhead . . . . 6
1.4.3 Ignoring Data Dependencies Causes Search Overhead . . . . . 7
1.4.4 Complex Interactions Leading to Deployment Overhead . . . . 8

1.5 Performance and Scalability Studies . . . . . 8
1.6 Scope and Research Goals . . . . . 9
1.7 Problem Statement and Research Questions . . . . . 10
1.8 Research Methodology . . . . . 12
1.9 Structure of the thesis . . . . . 12
1.10 Contributions . . . . . 13

**2 Background** **15**
2.1 Upper Confidence Bound (UCB) . . . . . 16
2.2 Upper Confidence Bounds for Trees (UCT) . . . . . 16
2.2.1 UCT Formula . . . . . 17
2.2.2 UCT Algorithm . . . . . 17
2.3 Parallelization Methods for MCTS . . . . . 17
2.3.1 Parallel Methods with a Shared Data Structure . . . . . 17
2.3.2 Parallel Methods with More than one Data Structure . . . . . 18
2.4 Case Studies . . . . . 19
2.4.1 Case 1: The Game of Hex . . . . . 19
2.4.2 Case 2: Horner Schemes . . . . . 20
2.5 Performance Metrics . . . . . 20
2.5.1 Playout Speedup . . . . . 21
2.5.2 Playing Strength . . . . . 21
2.6 Our ParallelUCT Package . . . . . 22
2.6.1 Framework of multiple benchmark problems . . . . . 22
2.6.2 Framework of multiple parallelization methods . . . . . 23
2.6.3 Framework of multiple programming models . . . . . 23

**3 Thread-level Parallelization for MCTS** **25**
3.1 Micro-benchmark Code Performance . . . . . 27
3.1.1 Xeon Phi Micro-architecture . . . . . 27
3.1.2 Experimental Setup . . . . . 28
3.1.3 Experimental Design . . . . . 30
3.1.4 Experimental Results . . . . . 30
3.1.5 Section Conclusion . . . . . 33
3.2 FUEGO Performance and Scalability . . . . . 33
3.2.1 Experimental Setup . . . . . 34
3.2.2 Experimental Design . . . . . 34
3.2.3 Experimental Results . . . . . 35
3.2.4 Section Conclusion . . . . . 39
3.2.5 Answer to RQ1a for FUEGO . . . . . 39
3.3 ParallelUCT Performance and Scalability . . . . . 40

3.3.1 Experimental Setup . . . 40
3.3.2 Experimental Design . . . 41
3.3.3 Experimental Results . . . 41
3.3.4 Section Conclusions . . . 46
3.3.5 Answer to RQ1a for ParallelUCT . . . 47
3.4 Related Work . . . 48
3.5 Answer to RQ1 . . . 48

**4 Task-level Parallelization for MCTS 51**
4.1 Irregular Parallelism Challenge . . . 52
4.2 Achieving Task-level Parallelization . . . 52
4.2.1 Decomposition of Iterations into Tasks . . . 53
4.2.2 Ignoring Data Dependencies among Iterations . . . 53
4.3 Threading Libraries . . . 53
4.3.1 Cilk Plus . . . 54
4.3.2 Threading Building Blocks . . . 54
4.4 Grain Size Controlled Parallel MCTS . . . 54
4.5 Implementation Considerations . . . 56
4.5.1 Shared Search Tree Using Locks . . . 56
4.5.2 Random Number Generator . . . 57
4.6 Performance and Scalability Study . . . 57
4.7 Experimental Setup . . . 58
4.8 Experimental Design . . . 58
4.9 Experimental Results . . . 59
4.10 Discussion and Analysis . . . 60
4.11 Related Work . . . 64
4.12 Answer to RQ2 . . . 64

**5 A Lock-free Algorithm for Parallel MCTS 67**
5.1 Shared Data Structure Challenge . . . 68
5.1.1 Parallelization with a Single Shared Tree . . . 69
5.1.2 The Race Conditions . . . 69
5.1.3 Protecting Shared Data Structure . . . 70
5.2 Related Work . . . 71
5.2.1 Lock-based Methods . . . 71
5.2.2 Lock-free Methods . . . 72
5.3 A New Lock-free Tree Data Structure and Algorithm . . . 73
5.4 Implementation Considerations . . . 77
5.5 Experimental Setup . . . 77
5.5.1 The Game of Hex . . . 78

5.5.2 Performance Metrics . . . 78
5.5.3 Hardware . . . 78
5.6 Experimental Design . . . 78
5.7 Experimental Results . . . 79
5.7.1 Scalability and $C_p$ parameters . . . 79
5.7.2 GSCPM vs. Root Parallelization . . . 82
5.8 Answer to RQ3 . . . 83

**6 Pipeline Pattern for Parallel MCTS 85**
6.1 Data Dependencies Challenges . . . 86
6.1.1 Loop Independent Data Dependency . . . 86
6.1.2 Loop Carried Data Dependency . . . 87
6.1.3 Why a Pipeline Pattern? . . . 87
6.2 Design of 3PMCTS . . . 88
6.2.1 A Pipeline Pattern for MCTS . . . 88
6.2.2 Pipeline Construction . . . 91
6.3 Implementation Considerations . . . 92
6.4 Experimental Setup . . . 92
6.4.1 Horner Scheme . . . 92
6.4.2 Performance Metrics . . . 93
6.4.3 Hardware . . . 93
6.5 Experimental Design . . . 93
6.6 Experimental Results . . . 94
6.6.1 Performance and Scalability of 3PMCTS . . . 94
6.6.2 Flexibility of Task Decomposition in 3PMCTS . . . 96
6.7 Answer to RQ4 . . . 97

**7 Ensemble UCT Needs High Exploitation 99**
7.1 Ensemble UCT . . . 100
7.2 Related Work . . . 101
7.3 Experimental Setup . . . 102
7.3.1 The Game of Hex . . . 102
7.3.2 Hardware . . . 102
7.4 Experimental Design . . . 103
7.5 Experimental Results . . . 103
7.6 Answer to the First Part of RQ5 . . . 106

**8 An Analysis of Virtual Loss in Parallel MCTS** **109**
8.1 Virtual Loss . . . 110
8.2 Related Work . . . 112
8.3 Experimental Setup . . . 112
8.4 Experimental Design . . . 112
8.5 Experimental Results . . . 113
8.6 Answer to the Second Part of RQ5 . . . 115
8.7 A Complete answer to RQ5 . . . 115

**9 Conclusions and Future Research** **117**
9.1 Answers to the RQs . . . 117
9.1.1 Answer to RQ1 . . . 117
9.1.2 Answer to RQ2 . . . 118
9.1.3 Answer to RQ3 . . . 118
9.1.4 Answer to RQ4 . . . 119
9.1.5 Answer to RQ5 . . . 119
9.2 Answer to the PS . . . 120
9.3 Limitations . . . 120
9.3.1 Maximizing Hardware Usage . . . 120
9.3.2 Using More Case Studies . . . 121
9.4 Future Research . . . 121

**Bibliography** **123**

**Appendices** **131**

**A Micro-benchmark Programs** **133**

**B Statistical Analysis of Self-play Experiments** **135**

**C Implementation of GSCPM** **137**
C.1 TBB . . . 137
C.2 Cilk Plus . . . 137
C.3 TPFIFO . . . 138

**D Implementation of 3PMCTS** **139**
D.1 Definition of Token Data Type (TDT) . . . 139
D.2 TBB Implementation Using TDD . . . 141

**Summary** **143**

**Samenvatting** **145**

**Acknowledgment** **147**

**Curriculum Vitae** **149**

**Publications** **151**

**SIKS Dissertation Series** **153**

# List of Definitions


1.1 Parallelization . . . 3
1.2 Thread . . . 4
1.3 Multi-core Processor . . . 4
1.4 Task . . . 4
1.5 Many-core Processor . . . 5
1.6 Parallel Pattern . . . 5
1.7 Irregular Parallelism . . . 6
1.8 Load Balancing . . . 6
1.9 Shared Data Structure . . . 7
1.10 Synchronization . . . 7
1.11 Loop Carried Data Dependency . . . 7
1.12 Loop Independent Data Dependency . . . 7
1.13 Search Overhead . . . 7
1.14 Complex Interactions . . . 8
1.15 Deployment Overhead . . . 8
1.16 Performance Study . . . 8
1.17 Playout Speedup . . . 8
1.18 Playing Strength . . . 8
1.19 Scalability Study . . . 9
1.20 Memory Bandwidth . . . 9
1.21 Uniform Memory Access . . . 10
1.22 Many Integrated Core . . . 10
1.23 Non Uniform Memory Access . . . 10

2.1 Exploitation . . . . . 16
2.2 Exploration . . . . . 16
2.3 Tree Parallelization . . . . . 18
2.4 Root Parallelization . . . . . 19
2.5 Strong Scalability . . . . . 21

3.1 Thread Affinity Policy . . . . . 28
3.2 Double-Precision Floating-Point Format . . . . . 29
3.3 Integer Format . . . . . 30

4.1 Iteration Pattern . . . . . 53
4.2 Fork-join Pattern . . . . . 53
4.3 Iteration-level Task . . . . . 55
4.4 Iteration-level Parallelism . . . . . 55
4.5 Fork-join Parallelism . . . . . 55

5.1 Race Condition . . . . . 69
5.2 Lock-based . . . . . 70
5.3 Lock-free . . . . . 71

6.1 Operation-Level Task . . . . . 86
6.2 Operation-Level Parallelism . . . . . 86
6.3 Sequence Pattern . . . . . 87
6.4 Iteration Pattern . . . . . 87
6.5 Pipeline Pattern . . . . . 88

8.1 Virtual Loss . . . . . 110

# List of Figures


1.1 An example of the search tree. . . . . 3
1.2 The main loop of MCTS. . . . . 3
1.3 One iteration of MCTS. . . . . 4

2.1 A sample board for the game of Hex . . . . . 19

3.1 Intel Xeon Phi Architecture. . . . . 29
3.2 Performance and scalability of double-precision operations for different numbers of iterations. . . . . 31
3.3 Memory bandwidth of double-precision operations on the Xeon Phi for increasing numbers of threads. Each interval contains 27 points. . . . . 32
3.4 Performance and scalability of integer operations of the Xeon Phi for different numbers of threads. . . . . 33
3.5 Performance and scalability of FUEGO in terms of PPS when it makes the second move. Average of 100 games for each data point. The board size is $9 \times 9$. . . . . 35
3.6 Scalability of FUEGO in terms of PW with $N$ threads against FUEGO with $N/2$ threads. The board size is $9 \times 9$. . . . . 38
3.7 Performance and scalability of ParallelUCT in terms of PPS for both Tree and Root Parallelization. . . . . 43
3.8 Scalability of ParallelUCT in terms of PW for Tree Parallelization. . . . 44
3.9 Scalability of ParallelUCT in terms of PW for Root Parallelization. . . . 46

4.1 The scalability profile produced by Cilkview for the GSCPM algorithm. The number of tasks is shown. Higher is more fine-grained. . . . . 57
4.2 Speedup for task-level parallelization utilizing five methods for parallel implementation from four threading libraries. Higher is better. Left: coarse-grained parallelism. Right: fine-grained parallelism. . . . . 61
4.3 Comparing Cilkview analysis with TPFIFO speedup on the Xeon Phi. The dots show the number of tasks used for TPFIFO. The lines show the number of tasks used for Cilkview. . . . . 63

5.1 (5.1a) The initial search tree. The internal and non-terminal leaf nodes are circles. The terminal leaf nodes are squares. The curly arrows represent threads. (5.1b) Thread 1 and 2 are expanding node $v_6$. (5.1c) Thread 1 and 2 are updating node $v_3$. (5.1d) Thread 1 is selecting node $v_3$ while thread 2 is updating this node. . . . . 69
5.2 Tree parallelization with coarse-grained lock. . . . . 72
5.3 Tree parallelization with fine-grained lock. . . . . 72
5.4 The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$. (5.4a) Coarse-grained lock. (5.4b) Lock-free. . . . . 80
5.5 The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$ on the Xeon Phi. (5.5a) Coarse-grained lock. (5.5b) Lock-free. . . . . 80
5.6 (5.6a) The scalability of the algorithm for different $C_p$ values. (5.6b) Changes in the depth of tree when the number of tasks are increasing. 81
5.7 The playing results for lock-free Tree Parallelization versus Root Parallelization. The first value for $C_p$ is used for Tree Parallelization and the second value is used for Root Parallelization. . . . . 82

6.1 (6.1a) Flowchart of a pipeline with sequential stages for MCTS. (6.1b) Flowchart of a pipeline with parallel stages for MCTS. . . . . 88
6.2 Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are equal. . . . . 89
6.3 Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are not equal. . . . . 89
6.4 Scheduling diagram of a pipeline with parallel stages for MCTS. Using parallel stages create load balancing. . . . . 90
6.5 The 3PMCTS algorithm with a pipeline that has three parallel stages (i.e., EXPAND, RANDOMSIMULATION, and EVALUATION). . . . . 91

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. . . . . 94
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. . . . . 95
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. . . . . 97

7.1 The number of visits for root's children in Ensemble UCT and plain UCT. Each child represents an available move on the empty Hex board with size $11 \times 11$. Both Ensemble UCT and plain UCT have 80,000 playouts and $C_p = 0$. In Ensemble UCT, the size of the ensemble is 8. . 104
7.2 The percentage of wins for ensemble UCT is reported. The value of $C_p$ for plain UCT is always 1.0 when playing against Ensemble UCT. To the left few large UCT trees, to the right many small UCT trees. . . . . 105

8.1 Search overhead ($SO$) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles, and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that the higher $SO$ of Tree Parallelization with virtual loss means lower performance. . . . . 113
8.2 Efficiency (*Eff*) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that Tree Parallelization with virtual loss has a lower efficiency meaning lower performance. . . . . 114

# List of Tables


3.1 Thread affinity policies . . . . . 29
3.2 Performance of FUEGO on the Xeon CPU. Each column shows data for $N$ threads. The board size is $9 \times 9$. . . . . 36
3.3 Performance of FUEGO on the Xeon Phi. Each column shows data for $N$ threads. The board size is $9 \times 9$. . . . . 37

4.1 The conceptual effect of grain size. . . . . 56
4.2 Sequential baseline for GSCPM algorithm. Time in seconds. . . . . 59

5.1 Sequential execution time in seconds. . . . . 81

6.1 Sequential time in seconds when $C_p = 0.5$. . . . . 94
6.2 Definition of layouts for 3PMCTS. . . . . 96
6.3 Details of experiment to show the flexibility of 3PMCTS. . . . . 96

7.1 Different possible configurations for Ensemble UCT. Ensemble size is $n$. 101
7.2 The performance of Ensemble UCT vs. plain UCT based on win rate. . 103

# List of Listings

A.1 Micro-benchmark code for measuring performance of Xeon Phi. . . . . 133
A.2 Micro-benchmark code for measuring memory bandwidth of Xeon Phi. 134
C.1 Task parallelism for GSCPM using TBB (*task_group*). . . . . 137
C.2 Task parallelism for GSCPM using Cilk Plus (*cilk_spawn*). . . . . 138
C.3 Task parallelism for GSCPM using Cilk Plus (*cilk_for*). . . . . 138
C.4 Task parallelism for GSCPM, based on TPFIFO. . . . . 138
D.1 An implementation of the 3PMCTS algorithm in TBB. . . . . 141

# List of Abbreviations

| | |
|---|---|
| **3PMCTS** | Pipeline Pattern for Parallel MCTS. |
| **FIFO** | First In, First Out. |
| **FMA** | Fused Multiply Add. |
| **GFLOPS** | Giga Floating Point Operations per Second. |
| **GIPS** | Giga Integers per Second. |
| **GSCPM** | Grain Size Controlled Parallel MCTS. |
| **HEP** | High Energy Physics. |
| **HEPGAME** | High Energy Physics Game. |
| **ILD** | Iteration-Level Dependency. |
| **ILP** | Iteration-Level Parallelism. |
| **ILT** | Iteration-Level Task. |
| **ISA** | Instruction Set Architecture. |
| **MC** | Memory Controller. |
| **MCTS** | Monte Carlo Tree Search. |
| **MIC** | Many Integrated Core. |
| **NUMA** | Non Uniform Memory Access. |

**OLD** Operation-Level Dependency.
**OLP** Operation-Level Parallelism.
**OLT** Operation-Level Task.

**PPS** Playouts per Second.
**PS** Problem Statement.
**PW** Percentage of Wins.

**RNG** Random Number Generation.
**RQ** Research Question.

**SMT** Simultaneous Multithreading.

**TBB** Threading Building Blocks.
**TD** Tag Directories.
**TPFIFO** Thread Pool with FIFO scheduling.

**UCB** Upper Confidence Bound.
**UCT** Upper Confidence Bounds for Trees.
**UMA** Uniform Memory Access.

**VPUs** Vector Processing Units.

# 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

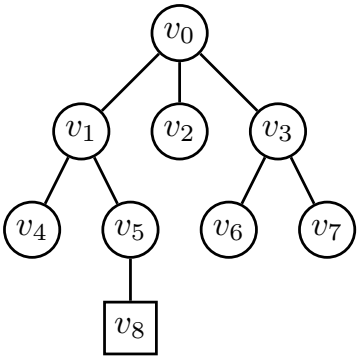



Figure 1.1: An example of the search tree.

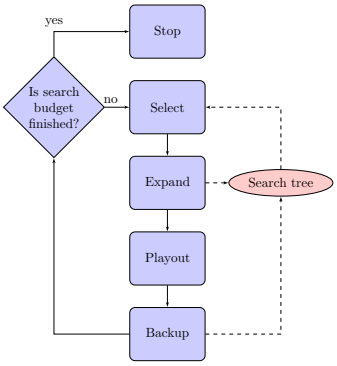



Figure 1.2: The main loop of MCTS.

reached, i.e., RANDOMSIMULATION. The terminal state is evaluated using a utility function to produce a reward value $\Delta$, 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 $\Delta$ [BPW+12]. These values are required by the selection policy (see Figure 1.3d).

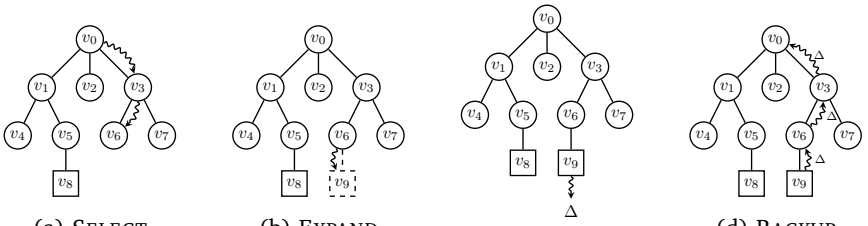
As soon as the computational budget is exhausted, the best child of the root node is returned (e.g., the one with the highest number of visits).

## 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).

# 2 Background

The MCTS algorithm iteratively repeats four steps or operations to construct a search tree until a predefined computational budget (i.e., time or iteration constraint) is reached [CWvdH+08b]. Algorithm 2.1 shows the general MCTS algorithm (see Section 1.2 and Algorithm 2.2).

**Algorithm 2.1:** The general MCTS algorithm.

```
1 Function MCTS(s_0)
2     v_0 := create root node with state s_0;
3     while within search budget do
4         ⟨v_l, s_l⟩ := SELECT(v_0, s_0);
5         ⟨v_l, s_l⟩ := EXPAND(v_l, s_l);
6         Δ := PLAYOUT(v_l, s_l);
7         BACKUP(v_l,Δ);
```

The purpose of MCTS is to approximate the domain-dependent theoretic value of the actions that may be selected from the current state by iteratively creating a partial search tree [BPW+12]. How the search tree is built depends on how nodes in the tree are selected (i.e., tree selection policy). For instance, nodes in the tree are selected according to the estimated probability that they are better than the current best action. It is essential to reduce the estimation error of the nodes' values as quickly as possible. The current chapter provides a more detailed overview of MCTS. Section 2.1 describes the UCB selection policy. In Section 2.2, we provide the UCT formula and the UCT algorithm. Section 2.3 discusses the parallelization methods for MCTS. Section 2.4 presents the benchmarks for experimental studies. Section 2.5 explains the performance metrics. Finally, Section 2.6 briefly describes our software tool.

## 2.1 Upper Confidence Bound (UCB)

The tree selection policy in the MCTS algorithm is based on two fundamentally different concepts, viz. exploitation and exploration. Hence, the selection is a search process and the aim of the search is to reduce the error as soon as possible [KS06].

**Definition 2.1 (Exploitation)** *Exploitation looks in areas which appear to be promising [BPW+12].*

**Definition 2.2 (Exploration)** *Exploration looks in areas that so far have not been sampled well [BPW+12].*

Kocsis and Szepesvári [KS06] aimed to design a Monte Carlo search algorithm that had a small error probability if stopped prematurely and that converged to the domain-dependent theoretic optimum given sufficient time [KS06]. They proposed the use of the simplest Upper Confidence Bound (UCB) policy (i.e., UCB1) as a tree selection policy for MCTS. UCB1 is an obvious choice for node selection given its application in multi-armed bandit problems for balancing between exploitation and exploration of actions. Bandit problems are a well-known class of sequential decision problems, in which one needs to choose among *K* actions (e.g., the *K* arms of a multi-armed bandit slot machine) to maximize the cumulative reward by consistently taking the optimal action [BPW+12, ACBF02].

Auer et al. [ACBF02] proposed UCB1 for *bandit problems*. The UCB1 policy selects the arm $j$ that maximizes:

$$UCB1(j) = \overline{X}_j + \sqrt{\frac{2\ln(n)}{n_j}} \tag{2.1}$$

where $\overline{X}_j$ is the average reward from arm $j$; $n_j$ is the number of times arm $j$ was played, and $n$ is the overall number of plays so far. The first term at the right-hand side $\overline{X}_j$ encourages the exploitation of higher-reward arms, while the second term at the right-hand side $\sqrt{\frac{2\ln(n)}{n_j}}$ promotes the exploration of less played arms.

## 2.2 Upper Confidence Bounds for Trees (UCT)

This section explains the most common algorithm in the MCTS family, the Upper Confidence Bounds for Trees (UCT) algorithm. The formulas are given in Subsection 2.2.1 and the algorithm in Subsection 2.2.2

### 2.2.1 UCT Formula

The UCT algorithm addresses the exploitation-exploration dilemma in the selection step of the MCTS algorithm using the UCB1 policy [KS06]. A child node $j$ is selected to maximize:

$$UCT(j) = \overline{X}_j + 2C_p\sqrt{\frac{2\ln(N(v))}{N(v_j)}} \tag{2.2}$$

where $\overline{X}_j = \frac{Q(v_j)}{N(v_j)}$ is an approximation of the node $j$ domain-dependent theoretic value. $Q(v_j)$ is the total reward of all playouts that passed through node $j$, $N(v_j)$ is the number of times node $j$ has been visited, $N(v)$ is the number of times the parent of node $j$ has been visited, and $C_p \geq 0$ is a constant. The first term at the right-hand side is for exploitation and the second term is for exploration [KS06]. The decrease or increase in the amount of exploration can be adjusted by $C_p$ in the exploration term.

### 2.2.2 UCT Algorithm

The UCT algorithm is given in Algorithm 2.2. Each node $v$ stores four pieces of data: the action to be taken $a(v)$, $p(v)$ the current player at node $v$, the total simulation reward $Q(v)$ (a real number), and the visit count $N(v)$ (a non-negative integer). Each node $v$ is also associated with a state $s$. The state $s$ is recalculated as the SELECT and EXPAND steps descends the tree. The term $\Delta\langle p(v)\rangle$ denotes the reward after simulation for each player.

## 2.3 Parallelization Methods for MCTS

In this section, two categories for parallelization of MCTS are presented. Traditionally, parallelization methods for MCTS are classified based on the parallelism technique. Currently, we believe that we should classify them into two categories solely based on the way that the search tree is used. We introduce parallel methods with a shared tree in Subsection 2.3.1 and with an ensemble of search trees in Subsection 2.3.2.

### 2.3.1 Parallel Methods with a Shared Data Structure

The first category is for the parallel methods with a shared search tree. The tree is shared among parallel threads or processes which means data is accessible globally. The methods that belong to this category can be implemented on both shared-memory and distributed-memory systems. In both environments, a synchronization method should create constraints threads from accessing the tree simultaneously. The most well-known method in this category is Tree Parallelization.

**Algorithm 2.2:** The UCT algorithm.

**Function** UCTSEARCH($s_0$)
  $v_0$ := create root node with state $s_0$;
  **while** *within search budget* **do**
    $\langle v_l, s_l \rangle$ := SELECT($v_0, s_0$);
    $\langle v_l, s_l \rangle$ := EXPAND($v_l, s_l$);
    $\Delta$ := PLAYOUT($v_l, s_l$);
    BACKUP($v_l$,$\Delta$);
  **return** $a(best\ child\ of\ v_0)$

**Function** SELECT(*Node v,State s) : <Node,State>*
  **while** $v$ *is fully expanded* **do**
    $v_l := \underset{v_j \in children\ of\ v}{\arg\max} \frac{Q(v_j)}{N(v_j)} + 2C_p\sqrt{\frac{2\ln(N(v))}{N(v_j)}}$;
    $s_l := p(v)$ takes action $a(v_l)$ from state $s$;
    $v := v_l$;
    $s := s_l$;
  **return** $\langle v, s \rangle$;

**Function** EXPAND(*Node v,State s) : <Node,State>*
  **if** $s$ *is non-terminal* **then**
    choose $a \in$ set of untried actions from state $s$;
    add a new child $v'$ with $a$ as its action to $v$;
    $s' := p(v)$ takes action $a$ from state $s$;
  **return** $\langle v', s' \rangle$;

**Function** PLAYOUT(*Node v,State s)*
  **while** $s$ *is non-terminal* **do**
    choose $a \in$ set of untried actions from state $s$ uniformly at random;
    $s := p(v)$ takes action $a$ from state $s$;
  $\Delta\langle p(v) \rangle$ := reward for state $s$ for each player $p$;
  **return** $\Delta$

**Function** BACKUP(*Node v,*$\Delta$*) : void*
  **while** $v$ *is not* **null do**
    $N(v) := N(v) + 1$;
    $Q(v) := Q(v) + \Delta\langle p(v) \rangle$;
    $v :=$ parent of $v$;

**Definition 2.3 (Tree Parallelization)** *In Tree Parallelization, the tree is shared among parallel threads, tasks, or processes which means data is accessible globally.*

### 2.3.2 Parallel Methods with More than one Data Structure

The second category is for the parallel methods where several search trees or an ensemble of search trees are used. Each parallel thread has its own search tree which means the information is local to that thread. The methods that belong to this category can also be implemented on both shared-memory and distributed-memory environments. The most well-known method in this category is Root Parallelization.

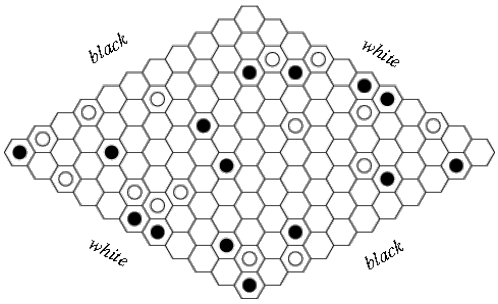



Figure 2.1: A sample board for the game of Hex

**Definition 2.4 (Root Parallelization)** *In Root Parallelization, each parallel thread, task, or process has its own search tree which means the information is local to that thread.*

## 2.4 Case Studies

In this section, we present two case studies for MCTS. In Subsection 2.4.1 we present the game of Hex, a strategy board game for two players. In Subsection 2.4.2 we describe the method for approximating the roots of a polynomial called Horner scheme.

### 2.4.1 Case 1: The Game of Hex

Hex is a board game with a diamond-shaped board of hexagonal cells [AHH10, HT19]. The game is usually played on a board of size 11 on a side, for a total of 121 hexagons, as illustrated in Figure 2.1 [Wei17]. Each player is represented by a color (Black or White). Players take turns placing a stone of their color on a cell on the board. The goal for each player is to create a connected chain of stones between the opposing sides of the board marked by their colors. The first player to complete this path wins the game. The game cannot end in a draw since no path can be completely blocked except by a complete path of the opposite color. Since the first player to move in Hex has a distinct advantage, the swap rule is generally implemented for fairness. This rule allows the second player to choose whether to switch positions with the first player after the first player has made a move.

#### Evaluation Function

In our implementation of Hex, a disjoint-set data structure is used to determine the connected stones. A disjoint-set data structure maintains a collection of disjoint (non-overlapping) subsets of a set of elements $S = \{S_1, S_2, \ldots, S_k\}$. A union-find algorithm

performs two operations on such a data structure: First, the *Find* operation determines in which subset a particular element is located. This can be used for determining whether two elements are in the same subset. Second, the *Union* operation joins two subsets into a single subset. Each set is identified by a representative, which usually is a member in the set. Using this data structure and algorithm, the evaluation of the board position to find the player who won the game becomes very efficient [GI91].

### 2.4.2 Case 2: Horner Schemes

Horner's rule is an algorithm for polynomial computation that reduces the number of multiplications and results in a computationally efficient form [OS12]. For a polynomial in one variable

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \tag{2.3}$$

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

$$p(x) = ((a_n x + a_{n-1})x + \ldots)x + a_0. \tag{2.4}$$

This representation reduces the number of multiplications to $n$ and has $n$ additions. Therefore, the total evaluation cost of the polynomial is $2n$. Here it is assumed that the cost of addition and multiplication are equal.

Horner's rule can be generalized for multivariate polynomials. Here, Eq. 2.4 applies to a polynomial for each variable, treating the other variables as constants. The order of choosing 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 [ALSU07]. 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.

## 2.5 Performance Metrics

In our experiments, the performance is reported by two metrics: (A) playout speedup (Subsection 2.5.1) and (B) playing strength (Subsection 2.5.2). Below, we define both metrics.

### 2.5.1 Playout Speedup

The most important metric related to performance and parallelism is *speedup*. In the literature, this form of speedup is called *playout speedup* [CWvdH08a]. We use playout speedup to show the effect on a program's performance in terms of speed of execution after any resource enhancement (e.g., increasing the number of threads or cores). The speedup can be defined for two different types of quantities: (A1) latency and (A2) throughput.

#### A1: Playout speedup in latency

We measure the speedup in time, which is a latency measure. Speedup compares the time for solving the identical computational problem on one worker versus that on $P$ workers

$$PlayoutSpeedup_{latency} = \frac{T_1}{T_P}. \tag{2.5}$$

where $T_1$ is the time of the program with one worker and $T_P$ is the time of the program with $P$ workers. In our results we report the scalability of our parallelization as *strong scalability* which means that the problem size remains fixed as $P$ varies. The problem size is the number of playouts (i.e., the search budget) and $P$ is the number of threads or tasks.

**Definition 2.5 (Strong Scalability)** *Strong scalability means that the problem size remains fixed as the number of resources varies.*

#### A2: Playout speedup in throughput

We measure the speedup in Playouts per Second (PPS), which is a throughput measure. First, we execute the program with one thread, which yields a PPS of $n$. Next, we execute the program with P threads, which yields a PPS of $m$. Using the speedup formula gives

$$PlayoutSpeedup_{throughput} = \frac{Q_{\text{P}}}{Q_1} = \frac{m \text{ PPS}}{n \text{ PPS}} \tag{2.6}$$

### 2.5.2 Playing Strength

The second most important metric related to the performance of parallel MCTS is *playing strength*. We use playing strength to show the effect on a program's performance in terms of quality of search after any resource enhancement (e.g., increasing the number of threads or cores). Playing strength can be defined for two different types of problems: (B1) two-player game and (B2) optimization problem.

**B1: Playing Strength in a two player game**

We measure the strength of player $a$ in Percentage of Wins (PW) per tournament versus player $b$ for the two-player game, such as Hex or Go, which is a win-rate

$$PlayingStrength(a)_{PW} = \frac{W_a}{W_a + W_b} * 100. \tag{2.7}$$

where $W_a$ is the number of wins for player $a$ and $W_b$ is the number of wins for player $b$. If there is a draw, it will be counted as a win for both players.

**B2: Playing strength in an optimization problem**

We measure the strength of an MCTS player $a$ in the *number of operations* in the optimized expression, which is a solution for the Horner scheme optimization problem. A lower value is desirable when we increase the numbers of threads or tasks.

## 2.6 Our ParallelUCT Package

To be able to investigate the research questions of this thesis a new software framework for parallel MCTS has been developed. The tool has been designed from scratch and is implemented in C++. Our tool is named *ParallelUCT*. The tool is open source, and its source codes are accessible [1].

The ParallelUCT framework has many features that enable us to answer the research questions mentioned in Section 1.7. Below we describe the three most important elements of this package, viz. in Subsection 2.6.1 we present multiple benchmark problems that are provided by the ParallelUCT, in Subsection 2.6.2 we describe multiple parallelization methods in ParallelUCT, and in Subsection 2.6.3 we provide the list of parallel programming models that are used in ParallelUCT.

### 2.6.1 Framework of multiple benchmark problems

In our research we focus on two benchmark problems. They are our case studies (Hex and Horner schemes). Both are implemented in the ParallelUCT framework. This software framework is extensible. It means that new problems such as other games or optimization problems can be added to it quickly. A developer should solely implement the original problem and provide it to the framework. The only requirement is to follow the standard of implementation which is provided by the software framework. The standard is available in the documentation of the ParallelUCT package [MPvdHV15a].

[1] `http://github.com/mirsoleimani/paralleluct`

### 2.6.2 Framework of multiple parallelization methods

We focus on two parallelization methods. They are methods with a shared data structure and with more than one data structure which are implemented in the framework. Examples of such methods are Tree Parallelization and Root Parallelization. A user can run the ParallelUCT executable program using one of these methods via command line options. The complete list of command line options is accessible via the help option of the program (see `http://github.com/mirsoleimani/paralleluct`).

### 2.6.3 Framework of multiple programming models

Finally, we focus on programming models. The parallelization methods are implemented in the framework using modern threading libraries such as Cilk [LP98], TBB [Rei07], and C++11. A user can run the ParallelUCT executable program with one of these threading libraries also via command line options. The complete list of command line options is accessible via the help option of the ParallelUCT executable program (see `http://github.com/mirsoleimani/paralleluct`).

# 3

# Thread-level Parallelization for MCTS

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

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

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

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

[1] Based on:

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

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

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

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

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

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

important question called *RQ1a*.

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

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

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

## 3.1 Micro-benchmark Code Performance

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

### 3.1.1 Xeon Phi Micro-architecture

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

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

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

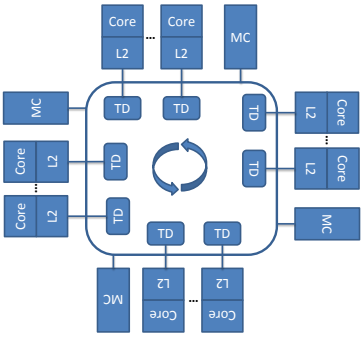
**Thread affinity policies:** On Xeon Phi, there are three predefined thread affinity policies for assigning threads to a core for obtaining improved or predictable performance [RVW+13]. These three policies are given in Table 3.1. A user can select one of the three policies for assigning threads or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

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

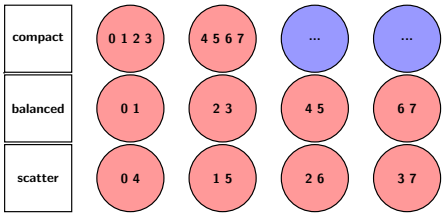
### 3.1.2 Experimental Setup

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

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




(a) Abstract microarchitecture




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

Figure 3.1: Intel Xeon Phi Architecture.

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

Table 3.1: Thread affinity policies

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

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

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

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

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

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

### 3.1.3 Experimental Design

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

### 3.1.4 Experimental Results

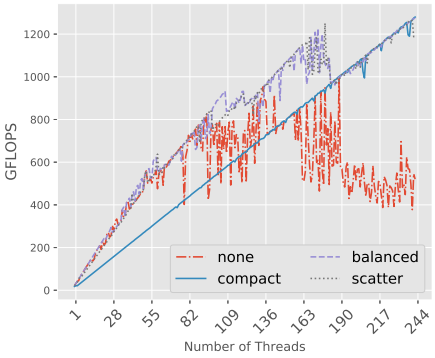
Figure 3.2, 3.3, and 3.4 show the results of our experiment. We will discuss the results for (A) double-precision operations and (B) integer operations.

#### A: Double-precision operations

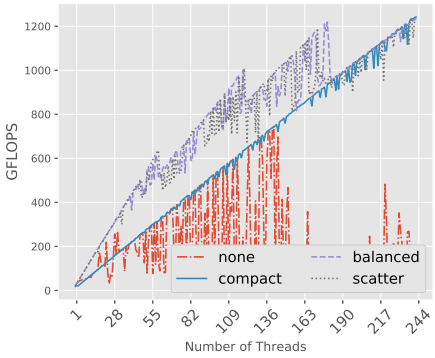
Below we discuss three issues: performance, scalability, and bandwidth.

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

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




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




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

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

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

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

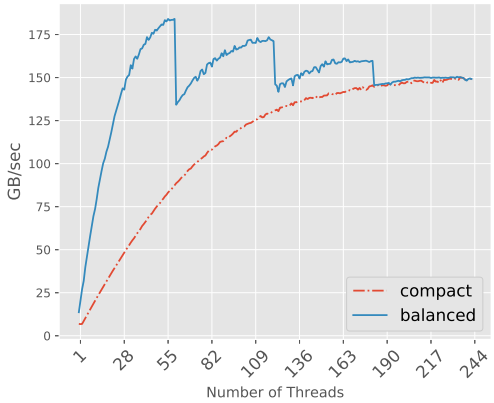



Figure 3.3: Memory bandwidth of double-precision operations on the Xeon Phi for increasing numbers of threads. Each interval contains 27 points.

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

**B: Integer Operations**

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

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

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

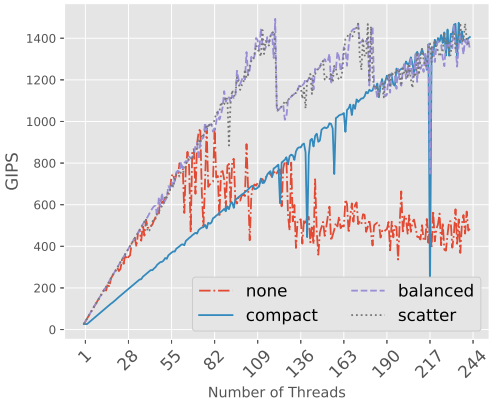



Figure 3.4: Performance and scalability of integer operations of the Xeon Phi for different numbers of threads.

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

### 3.1.5 Section Conclusion

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

## 3.2 FUEGO Performance and Scalability

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

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

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

### 3.2.1 Experimental Setup

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

#### Performance Metrics

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

### 3.2.2 Experimental Design

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

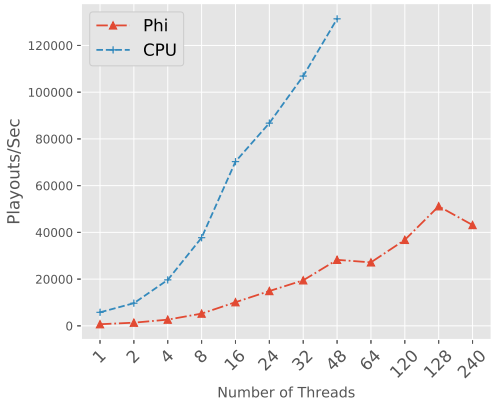



Figure 3.5: Performance and scalability of FUEGO in terms of PPS when it makes the second move. Average of 100 games for each data point. The board size is $9 \times 9$.

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

### 3.2.3 Experimental Results

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

#### A: Playout Speedup

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

#### A1: Experiment on multi-core

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

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

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

**A2: Experiment on many-core**

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

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

**A3: Conclusion**

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

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

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

**B: Playing Strength**

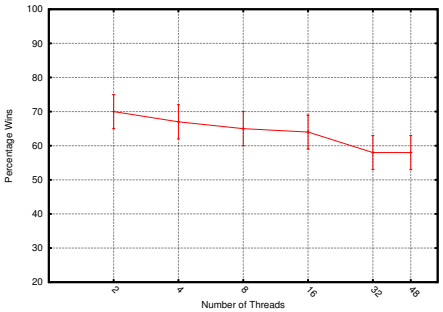
Figure 3.6 shows the scalability of FUEGO on both the Xeon CPU and the Xeon Phi in terms of the PW versus the number of threads. The graph shows the win-rate of the program with $N$ threads as the first player. A straight line means that the program is scalable in terms of PW. In the following, the results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

**B1: Experiment on multi-core**

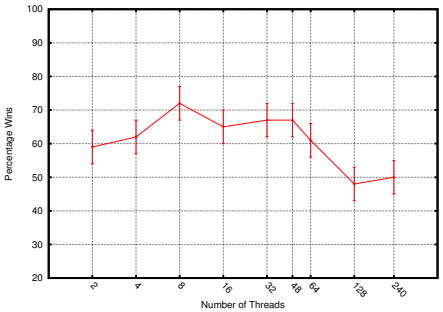
Figure 3.6a shows the results of the self-play experiments for FUEGO on the Xeon CPU. For the $9 \times 9$ board, the win-rate of the program with double the number of threads is better than the base program, starting at 70%, decreasing to 58% at 32 threads and then becomes flat. These results are entirely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

**B2: Experiment on many-core**

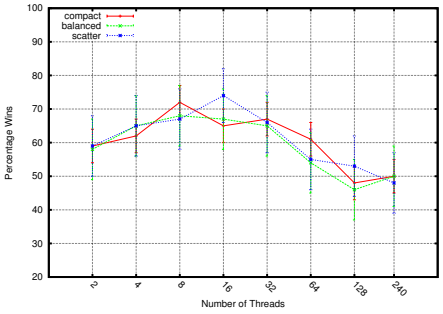
Figure 3.6b shows the performance and scalability of FUEGO in terms of PW on the many-core Xeon Phi. The scalability for the playing strength of FUEGO on the Xeon Phi differs notably from the Xeon CPU in Figure 3.6a. The Xeon CPU shows a smooth, slightly decreasing line. The Xeon Phi shows a more ragged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



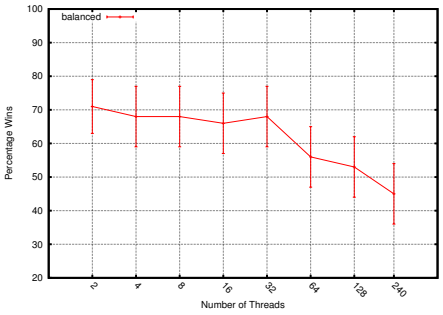

(a) Xeon CPU processor, with 200 games for each data point.




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




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




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

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

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

### B2a: Different thread affinity policies

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

#### B2b: Increasing time limit

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

#### B3: Conclusion

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

### 3.2.4 Section Conclusion

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

### 3.2.5 Answer to RQ1a for FUEGO

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

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

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

## 3.3 ParallelUCT Performance and Scalability

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

### 3.3.1 Experimental Setup

In order to generate statistically significant results for the game of Hex (board size $11 \times 11$) in a reasonable amount of time, both players do playouts of 1 second for choosing a move. To calculate the playing strength for the first player, we perform matches of two players against each other. Each match consists of 200 games, 100 with White and 100 with Black for each player. A statistical method based on [Hei01] and similar to [MKK14] is used to calculate 95%-level confidence lower and upper bounds on the real winning rate of a player, indicated by error bars in the graphs. The parameter $C_p$ is set at $1$ in all our experiments. To calculate the playout speedup

for the first player when considering the second move of the game, the average of the number of PPS over 200 games is measured. Taking the average removes: (1) the randomized feature of MCTS in game playing and (2) the so-called warm-up phase on the Xeon Phi [RJM+15].

The results were measured on a dual socket Intel machine with 2 Intel Xeon E5-2596v2 CPUs running at 2.40GHz. Each CPU has 12 cores, 24 hyperthreads, and 30 MB L3 cache. Each physical core has 256KB L2 cache. The peak TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.1* compiler is used to compile the program. The machine is equipped with an Intel Xeon Phi 7120P 1.238GHz which has 61 cores and 244 hardware threads. Each core has 512KB L2 cache. The co-processor has 16GB GDDR5 memory on board with an aggregate theoretical bandwidth of 352 GB/s. The peak turbo frequency is 1.33GHz. The theoretical performance of the 7120P is 2.416 TFLOPS or TIPS and 1.208 TFLOPS for single-precision or integer and double-precision floating-point arithmetic operations, respectively [Int13]. Intel's *icc 14.1* compiler is used to compile the program in *native mode*. A *native application* runs directly on the Xeon Phi and its embedded Linux operating system.

#### Performance Metrics

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

### 3.3.2 Experimental Design

In all of our experiments, we perform self-play Hex games in a tournament to measure performance and scalability. Each tournament consists of 200 head-to-head matches between the first player with $N$ threads and the second player with $N/2$ threads. Both players are given 1 second to make a move.

### 3.3.3 Experimental Results

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

**A: Playout Speedup**

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

**A1: Tree Parallelization**

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

**A1a: Experiment on multi-core**

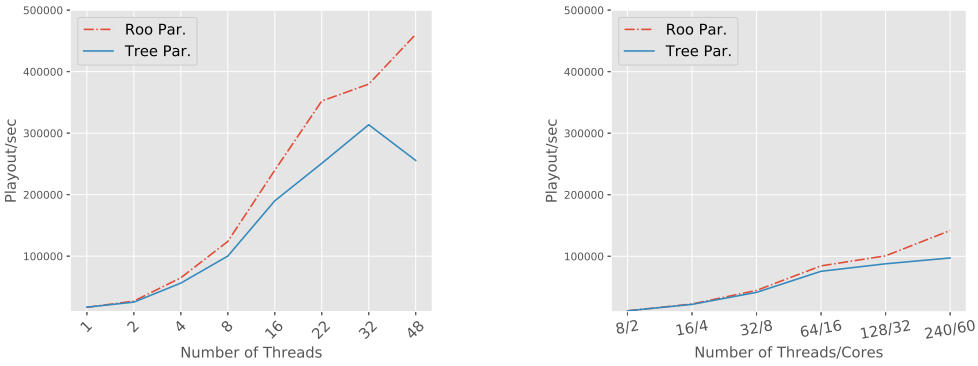
Figure 3.7a shows playout speedup on the Xeon CPU. We see a perfect playout speedup up to 4 threads and a near perfect speedup up to 16 threads. The increase in the number of playouts continues up to 32 threads, although the increase is no longer perfect. There is a sharp decrease in the number of playouts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

**A1b: Experiment on many-core**

In Figure 3.7b the playout speedup on the Xeon Phi is shown. A perfect playout speedup is observed up to 64 threads. We see that using a lock does not affect the performance of the algorithm up to this point. After 64 threads the performance drops, although the number of PPS still increases up to 240 threads. It should be noted that even with playout speedup increasing up to 240 threads, we see that at 240 threads on the Xeon Phi still, the number of PPS is less than on eight threads on the Xeon CPU. Our provisional conclusion here is that the performance for Tree Parallelization on the Xeon Phi is almost 30% of the peak performance on the Xeon CPU.

**A2: Root Parallelization**

Next, we will discuss the Root Parallelization, where threads are running independently and where no locking mechanism exists. Root Parallelization is well suited to




(a) Xeon CPU

(b) Xeon Phi

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

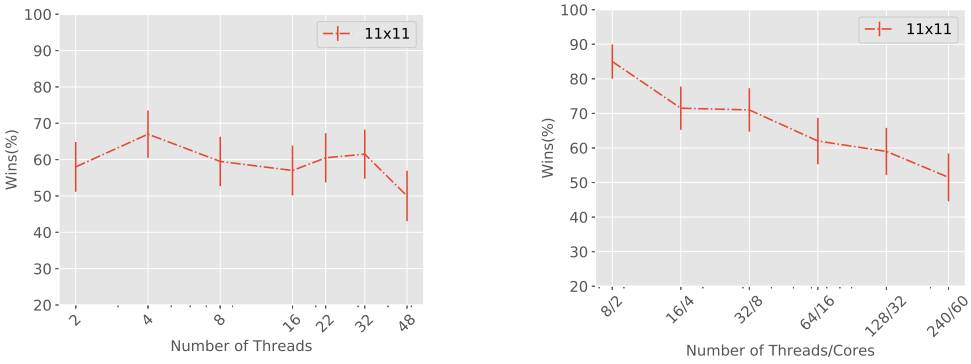
see whether the decrease in playout speedup in Tree Parallelization is due to locks or not. In Figure 3.7 the scalability of Root Parallelization on the Xeon CPU and the Xeon Phi are compared. In the following the results for the experiments on (A2a) the multi-core Xeon CPU and (A2b) the many-core Xeon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

**A2a: Experiment on multi-core**

As is shown in Figure 3.7a for the Xeon CPU, the playout speedup is perfect for up to 16 threads (while in Tree Parallelization it is for up to 4 threads). The second difference between these two algorithms is revealed at 48 threads where Root Parallelization still shows improvement in playout speedup. We may conclude that removing the lock in the expansion phase of Tree Parallelization improves performance for a high number of threads on the Xeon CPU.

**A2b: Experiment on many-core**

The performance of Root Parallelization on the Xeon Phi is shown in Figure 3.7b. Here, we require at least eight threads on the Xeon Phi to reach almost the same number of PPS as one thread on the Xeon CPU. On the Xeon Phi, with Root Parallelization, perfect playout speedup is achieved for up to 64 threads, which implies that the drops on 64 threads in Tree Parallelization performance are likely not due to locking. However, for 240 threads the number of playouts increases by a higher rate compared to Tree Parallelization. Overall, the peak performance for Root Parallelization on the Xeon Phi is almost 30% of its counterpart on the Xeon CPU. To understand the reason




(a) Xeon CPU

(b) Xeon Phi

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

for this low performance we did a detailed timing analysis to find out where most of the time of the algorithm has been spent in the selection, expansion, playout, or backup phase. For the Hex board size of $11 \times 11$, MCTS spends most of its time in the playout phase. This phase of the algorithm is problem dependent, for example, it is different for Go ($9 \times 9$) and Hex ($9 \times 9$) because they have different rules; the difference is even different for distinct board sizes. In our program, around 80% of the total execution time for performing a move is spent in the playout phase.

**A3: Conclusion**

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

**B: Playing Strength**

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

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

**B1: Tree Parallelization**

As already mentioned, it is also essential to evaluate the playing strength of the MCTS player for a game such as Hex. The goal is to see how the increase in the number of PPS reflects in a more dominant player. In the following the results for the experiments on (B1a) the multi-core Xeon CPU and (B1b) the many-core Xeon Phi are discussed.

**B1a: Experiment on multi-core**

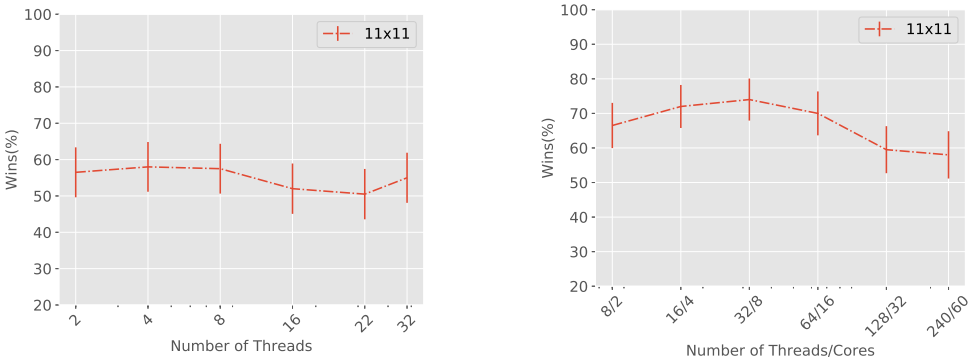
In Figure 3.8a playing strength for Tree Parallelization on the Xeon CPU is shown. Note that, since we compare the performance of $N$ threads against $N/2$ threads, an ideal perfect playing strength would give a straight, horizontal line of, say, 60% win rate for the player with more threads. We see good playing strength up to 32 threads. The win rate drops to 50 percent for 48 threads. This decrease in win rate is consistent with the drop in the number of PPS for 48 threads in Figure 3.7a. Our provisional conclusion here is that on the Xeon CPU, the playing strength follows playout speedup closely.

**B1b: Experiment on many-core**

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

**B2: Root Parallelization**

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




(a) Xeon CPU

(b) Xeon Phi

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

**B2a: Experiment on multi-core**

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

**B2b: Experiment on many-core**

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

**B3: Conclusion**

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

### 3.3.4 Section Conclusions

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

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

To understand the reason for this low performance we performed a detailed timing analysis to find out where the most time of the algorithm has been spent in the selection step, expansion step, playout step, or update step. For the Hex board size of $11 \times 11$, MCTS spends most of its time in the playout phase. This phase of the algorithm is problem dependent, for example, it is different for Go and Hex; the difference is even different for distinct board sizes. In our program, around 80% of the total execution time for performing a move is spent in the playout phase.

Since the playout phase dominates execution time of each thread, the Xeon CPU outperforms the Xeon Phi significantly because of more powerful cores. No method for vectorization has been devised for the playout phase. Therefore, for the current ratio of Xeon CPU cores versus Xeon Phi cores (24 versus 61), it is not possible to reach the same performance on the Xeon Phi because each core of the Xeon CPU is more powerful than each core of the Xeon Phi for sequential execution. From these results, we may conclude that for the current ratio of Xeon CPU cores versus Xeon Phi cores, the parallel MCTS algorithms for games such as Hex or Go on the Xeon Phi have a limitation. Therefore, it is interesting to investigate the limitation problem in the other domains in which MCTS has been successful such as those mentioned in [vdHPKV13].

### 3.3.5 Answer to RQ1a for ParallelUCT

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

## 3.4 Related Work

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

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

- Root Parallelization: Chaslot et al. [CWvdH08a] reported results that Root Parallelization shows perfect playout speedup for up to 16 threads. Soejima et al. [SKW10] analyzed the performance of Root Parallelization in detail. They showed that a Go player that uses lock-free Tree Parallelization with 4 to 8 threads outperformed the same program with Root Parallelization which utilizes 64 distributed CPU cores. This result suggests the superiority of Tree Parallelization over Root Parallelization in shared-memory machines.

## 3.5 Answer to RQ1

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

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

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

# 4

# Task-level Parallelization for MCTS

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

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

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

Three main contributions of this chapter are as follows.

[1]Based on:

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

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

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

## 4.1 Irregular Parallelism Challenge

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

## 4.2 Achieving Task-level Parallelization

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

### 4.2.1 Decomposition of Iterations into Tasks

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

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

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

### 4.2.2 Ignoring Data Dependencies among Iterations

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

## 4.3 Threading Libraries

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

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

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

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

### 4.3.1 Cilk Plus

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

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

### 4.3.2 Threading Building Blocks

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

## 4.4 Grain Size Controlled Parallel MCTS

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

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

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

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

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

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

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

Table 4.1: The conceptual effect of grain size.

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

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

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

## 4.5 Implementation Considerations

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

### 4.5.1 Shared Search Tree Using Locks

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

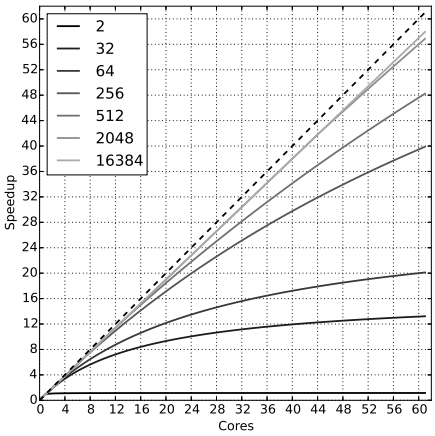



Figure 4.1: The scalability profile produced by Cilkview for the GSCPM algorithm. The number of tasks is shown. Higher is more fine-grained.

### 4.5.2 Random Number Generator

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

## 4.6 Performance and Scalability Study

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

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

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

## 4.7 Experimental Setup

The performance evaluation of GSCPM is carried out on a dual-socket Intel machine with 2 Intel Xeon E5-2596v2 CPUs running at 2.40GHz. Each CPU has 12 cores, 24 hyperthreads, and 30 MB L3 cache. Each physical core has 256KB L2 cache. The peak TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. The machine is equipped with an Intel Xeon Phi 7120P 1.238GHz which has 61 cores and 244 hardware threads. Each core has 512KB L2 cache. The Xeon Phi has 16GB GDDR5 memory on board with an aggregate theoretical bandwidth of 352 GB/s.

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

## 4.8 Experimental Design

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

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

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

| Processor | Board Size | Sequential Time (s) |
|---|---|---|
| Xeon CPU | $11 \times 11$ | $21.47 \pm 0.07$ |
| Xeon Phi | $11 \times 11$ | $185.37 \pm 0.53$ |

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

## 4.9 Experimental Results

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

### A: Sequential Performance

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

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

### B: Performance and Scalability on Xeon CPU

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

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

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

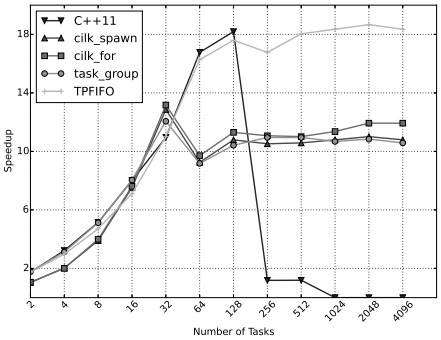
### C: Performance and Scalability on Xeon Phi

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

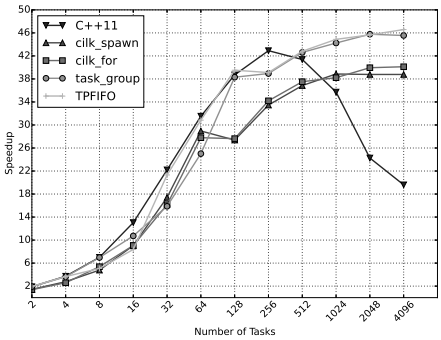
Creating a number of threads in C++11 that is equal to the number of tasks is the first approach that comes to mind. The best performance for the C++11 method is around 43 times speedup for 256 threads/tasks. However, the limitation of this approach is that creating larger numbers of threads has a large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

## 4.10 Discussion and Analysis

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




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




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

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

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

## A: Scaling behavior

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

## B: Performance

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

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

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

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

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

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

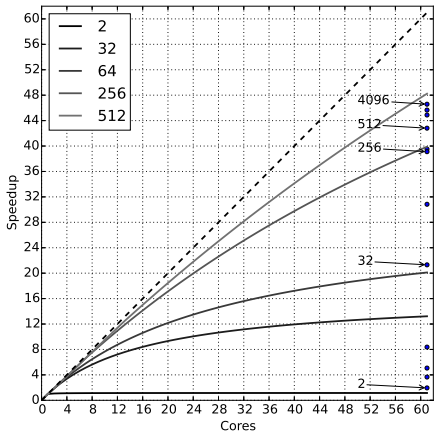



Figure 4.3: Comparing Cilkview analysis with TPFIFO speedup on the Xeon Phi. The dots show the number of tasks used for TPFIFO. The lines show the number of tasks used for Cilkview.

## C: Range of tasks

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

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

## 4.11 Related Work

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

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

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

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

## 4.12 Answer to RQ2

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

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

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

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

# 5

# A Lock-free Algorithm for Parallel MCTS

This chapter[1] addresses *RQ3* which is mentioned in Section 1.7.

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

One of the approaches for parallelizing MCTS for shared-memory systems is Tree Parallelization. The method is called so because a search tree is shared among multiple parallel threads. Each iteration of the MCTS has four operations (SELECT, EXPAND, PLAYOUT, and BACKUP). They are executed on the shared tree simultaneously. The MCTS algorithm uses the tree for storing the states of the domain and guiding the search process. The basic premise of the tree in MCTS is relatively straightforward: (a) nodes are added to the tree in the same order as they were expanded and (b) nodes are updated in the tree in the same order as they were selected. Therefore the following holds, if two parallel threads are performing the task of adding (EXPAND) or updating (BACKUP) the same node, there are potentially *race conditions*. Thus, one of the main challenges in Tree Parallelization is the prevention of *race conditions*.

In a parallel program a race condition shows a non-deterministic behavior that is generally considered to be a programming error [Wil12]. This behavior occurs when

[1] Based on:

- S. A. Mirsoleimani, H. J. van den Herik, A. Plaat and J. Vermaseren, A Lock-free Algorithm for Parallel MCTS, in Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2, 2018, pp. 589--598.

parallel threads perform operations on the same memory location without proper *synchronization* and one of the memory operations is a write. A program with a race condition may operate correctly sometimes and fail other times. Therefore, proper synchronization helps to coordinate threads to obtain the desired runtime order and avoid a race condition.

There are two lock-based methods to create synchronization in Tree Parallelization: (1) a coarse-grained lock, (2) a fine-grained lock [CWvdH08a].

Both methods are straightforward to design and to implement. However, locks are notoriously bad for parallel performance, because other threads have to wait until the lock is released. This is called *synchronization overhead*. The fine-grained lock has less synchronization overhead than the coarse-grained lock [CWvdH08a]. Yet, even fine-grained locks are often a bottleneck when many threads try to acquire the same lock. Hence, a *lock-free* tree data structure for parallelized MCTS is desirable and has the potential for maximal concurrency. A tree data structure is lock-free when more than one thread must be able to access its nodes concurrently. Here, the problem is that the development of a lock-free tree for parallelized MCTS is shown to be non-trivial. The difficulty of designing an adequate data structure stimulated the researchers in the community to come up with a spectrum of ideas [EM10, BG11]. As a case in point, Enzenberger et al. compromised over the correctness of computation. They accepted faulty results to have a lock-free search tree [EM10]. Below, we propose a new lock-free tree data structure without compromises together with a corresponding algorithm that uses the tree for parallel MCTS.

The remainder of this chapter is organized as follows. Section 5.1 describes the shared data structure challenge. Section 5.2 discusses related work. Section 5.3 gives the proposed lock-free algorithm. Section 5.4 shows implementation considerations. Section 5.5 presents the experimental setup, Section 5.6 describes experimental design, Section 5.7 provides the experimental results, and Section 5.8 provides an answer to RQ3.

## 5.1 Shared Data Structure Challenge

One of the difficulties for parallelizing MCTS is protecting a shared search tree without using locks to avoid synchronization overhead. The difficulty of this process caused the researchers in the MCTS community to even compromise over correctness of computation to have a lock-free search tree [EM10]. Below we discuss parallelization with a single shared tree in Subsection 5.1.1 and race conditions in Subsection 5.1.2. Subsection 5.1.3 provides the data protection methods for a shared tree.

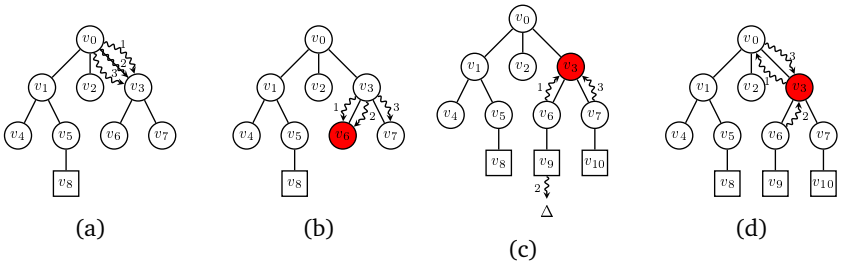



Figure 5.1: (5.1a) The initial search tree. The internal and non-terminal leaf nodes are circles. The terminal leaf nodes are squares. The curly arrows represent threads. (5.1b) Thread 1 and 2 are expanding node $v_6$. (5.1c) Thread 1 and 2 are updating node $v_3$. (5.1d) Thread 1 is selecting node $v_3$ while thread 2 is updating this node.

### 5.1.1 Parallelization with a Single Shared Tree

There are three parallelization methods for MCTS (i.e., *Root Parallelization*, *Leaf Parallelization*, and *Tree Parallelization*) that belong to two main categories: (A) parallelization with an ensemble of trees, and (B) parallelization with a single shared tree. The parallelization methods that belong to the former category (i.e., Root and Leaf Parallelization) do not need a shared search tree. But the methods that belong to the latter category use a shared search tree such as Tree Parallelization. In Tree Parallelization, parallel threads are potentially able to perform different MCTS operations on a same node of the shared tree [CWvdH08a]. These shared accesses are the source of the potential *race conditions*.

### 5.1.2 The Race Conditions

In parallel MCTS, parallel threads are manipulating a shared search tree concurrently. If two threads are performing the task of adding or updating the same node, there is a *race condition*.

**Definition 5.1 (Race Condition)** *A race condition occurs when concurrent tasks perform operations on the same memory location without proper synchronization and one of the memory operations is a write [MRR12].*

Consider the example search tree in Figure 5.1. Three parallel threads (1, 2 and 3 from $v_0$ to $v_3$) attempt to perform MCTS operations on the shared search tree. There are three race condition scenarios.

- Shared Expansion (SE): Figure 5.1b shows two threads (1 and 2) concurrently performing EXPAND($v_6$). In this SE scenario, synchronization is required. Obvi-

ously, a race condition exists if two parallel threads intend to add node $v_9$ to $v_6$ simultaneously. In such an SE race, the child node should be created and added to its parent only once.

- Shared Backup (SB): Figure 5.1c shows two threads ($1$ and $3$) concurrently performing BACKUP($v_3$). In the SB scenario, synchronization is required because there are two data race conditions when parallel threads update the value of $Q(v_3)$ and $N(v_3)$ simultaneously. There are two dangers: (a) the value of either $Q(v_3)$ or $N(v_3)$ could be corrupted due to concurrently writing them, and (b) the variable $Q(v_3)$ and $N(v_3)$ could be in an inconsistent state when the writing of their values does not happen together at the same time (i.e., the state of one variable is ahead of the other one).
- Shared Backup and Selection (SBS): Figure 5.1d shows thread $2$ performing BACKUP($v_3$) and thread $3$ performing SELECT($v_3$). In the SBS scenario, synchronization is required. Otherwise, a race condition may occur between (i) thread 3 reading the value of $Q(v_3)$, and (ii) before thread 3 can read the value of $N(v_3)$, thread 2 updates the value of $Q(v_3)$ and $N(v_3)$. Thus what happens is that when thread 3 reads the value of $N(v_3)$, the variables $Q(v_3)$ and $N(v_3)$ are not in the same state anymore and therefore thread 3 reads an inconsistent set of values ($Q(v_3)$ and $N(v_3)$).

Code with race conditions may operate correctly sometimes and fail other times. We have tp protect the shared data to avoid uncertainty in the execution.

### 5.1.3 Protecting Shared Data Structure

There are two groups of methods to protect a shared data structure, lock-based methods and lock-free methods.

**Lock-based Methods** use mutexes and locks to create synchronization and protect the shared data. The first obvious design used one mutex to protect the entire search tree, but later ones used more than one mutex to protect smaller parts of the search tree and allow a greater level of concurrency in accesses to the search tree [CWvdH08a]. Locks are notoriously bad for parallel performance, because other threads have to wait until the lock is released, and locks are often a bottleneck when many threads try to acquire the same lock. If we can write a search tree data structure that is safe for concurrent accesses without locks, there is the potential for maximum concurrency.

**Definition 5.2 (Lock-based)** *A data structure is lock-based when it uses mutexes and locks to create synchronization to protect the shared data.*

**Lock-free Methods** use a lock-free data structure. Such a data structure often uses the compare/exchange operation to make progress in an algorithm, rather than protecting a part that makes progress. For example, when modifying a shared variable, an approach using locks would first acquire the lock, then modify the variable, and finally release the lock. A lock-free approach would use compare/exchange to modify the variable directly. This requires only one memory operation rather than three, but designing a lock-free data structure is hard and needs extreme care.

**Definition 5.3 (Lock-free)** *A data structure is lock-free when more than one thread must be able to access it concurrently.*

## 5.2 Related Work

In this section, we present the related work for two categories of synchronization methods for Tree Parallelization: (1) lock-based methods and (2) lock-free methods.

### 5.2.1 Lock-based Methods

As already mentioned, one of the main challenges in Tree Parallelization is to prevent date race conditions using synchronization. Figure 5.2 shows the Tree Parallelization where two threads (1 and 2) simultaneously perform the EXPAND operation on a node ($v_6$) of the tree. There are two methods to create synchronization in this case for Tree Parallelization: (1) coarse-grained lock [CWvdH08a], (2) fine-grained lock [CWvdH08a]:

1. The coarse-grained lock method uses one lock to protect the entire search tree [CWvdH08a]. For example, in Figure 5.2a, both thread 1 and 2 want to expand node $v_6$, then thread 1 first acquires a lock; subsequently, it performs the EXPAND operation and finally releases the lock. During this process thread 2 also wanting to perform the EXPAND operation on node $v_6$ should wait for the release of the lock (see Figure 5.2b). This method is called coarse-grained because the access to the tree for performing the EXPAND operation will be given to one and only one thread, even if multiple threads want to expand different nodes inside the tree. For example, in Figure 5.2a, thread 3 also wants to perform the EXPAND operation, but on node $v_7$. However, the lock is already acquired by thread 1. Therefore, thread 3 should wait until the lock is released (see Figure 5.2b).

2. The fine-grained lock method uses one lock for each node of the tree to protect a smaller part of the search tree and to allow a greater level of concurrency in accesses to the search tree [CWvdH08a]. For example, in Figure 5.3a, thread 3

(a) (b)

Figure 5.2: Tree parallelization with coarse-grained lock.

(a) (b)

Figure 5.3: Tree parallelization with fine-grained lock.

also wants to perform the EXPAND operation, but on node $v_7$. It can acquire the lock in $v_7$ and should not wait (see Figure 5.2b).

Both lock-based methods use locks to protect shared data. However, these approaches suffer from synchronization overhead due to thread contentions and do not scale well [CWvdH08a]. A lock-free method can remove these problems.

### 5.2.2 Lock-free Methods

A lock-free implementation exists in the FUEGO package [EM10]. However, the method in [EM10] does not guarantee the computational consistency of the multithreaded program with the single-threaded program. To address the SE race condition, Enzenberger et al. assign to each thread an own memory array for creating nodes [EMAS10]. 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 in this way already received some updates; these updates will also be lost. It means that Enzenberger et al. ignore the SB and SBS race conditions. They accept the possible faulty updates and the inconsistency of parallel computation.

In the PACHI package [BG11], the method in [EM10] is used for performing lock-

free tree updates. Again, it means that both SB and SBS race conditions are neglected. However, to allocate children of a given node, PACHI does not use a per-thread memory pool as FUEGO does, but uses instead a pre-allocated global node pool and a single atomic increment instruction updating the pointer to the next free node. This addresses the memory overhead problem in FUEGO. However, there are still two other issues with this method: (1) the number of required nodes should be known in advance, and (2) the children of a node may not be assigned in consecutive memory locations which results in poor *spatial locality* (i.e., if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future). The spatial locality is specifically important for the SELECT operation.

## 5.3 A New Lock-free Tree Data Structure and Algorithm

We show our new lock-free tree data structure in Algorithm 5.1. The type name is *Node*. The UCT algorithm that uses the proposed data structure is given in Algorithm 5.2 (for the difference, see the end of this section).

Algorithm 5.1 uses the new multithreading-aware memory model of the C++11 Standard [Wil12]. To avoid the race conditions, the ordering of memory accesses by the threads has to be enforced [Wil12]. In our lock-free approach, we use the synchronization properties of the *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 the most popular platforms. The standard atomic types have different member functions such as *load()*, *store()*, *exchange()*, *fetch_add()*, and *fetch_sub()*. The differences are subtle. The member function *load()* is a load operation, whereas the *store()* is a store operation. The *exchange()* member function is special. It replaces the stored value in the atomic variable by a new value and automatically retrieves the original value. Therefore, we use two memory models for the memory-ordering option for all operations on atomic types: (1) *sequentially consistent* ordering (*memory_order_seq_cst*) and (2) *acquire_release* ordering (*memory_order_acquire* and *memory_order_release*). The default behavior of all atomic operations provides for *sequentially consistent* ordering. This 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*).

In Algorithm 5.1 each node $v$ stores nine different pieces of data: (1) $a$ the action to be taken, (2) $p$, the current player at node $v$, (3) $w_n$ (a 64-bit atomic integer)

**Algorithm 5.1:** The new lock-free tree data structure.

```
1  type
2      type a : int;
3      type p : int;
4      type w_n : atomic_int_64;
5      type children : Node*[];
6      type is_parent := false : atomic_bool;
7      type n_nonexpanded_children := -1 : atomic_int;
8      type is_expandable := false : atomic_bool;
9      type is_fully_expanded := false : atomic_bool;
10     type parent : Node*;
11     Function CreateChildren(actions) : <void>
12         if is_parent.exchange(true) is false then
13             j := 0;
14             while actions is not empty do
15                 choose a' ∈ actions;
16                 add a new child v' with a' as its action and p' as its player to the list of children;
17                 j := j+1;
18             n_nonexpanded_children.store(j);
19             is_expandable.store(
20             true,memory_order_release);
21     Function AddChild() : <Node*>
22         index := -1;
23         if is_expandable.load(memory_order_acquire) is true then
24             if (index := n_nonexpanded_children.fetch_sub(1)) is 0 then
25                 is_fully_expanded.store(true);
26             if index < 0 then
27                 return current node;
28             else
29                 return children[index];
30         else
31             return current node;
32     Function IsFullyExpanded() : <bool>
33         return is_fully_expanded.load();
34     Function Get() : <int,int>
35         w_n' := w_n.load();
36         w := high 32 bits of w_n';
37         n := low 32 bits of w_n';
38         return ⟨w, n⟩;
39     Function Set(int Δ)
40         w_n' := 0;;
41         high 32 bits of w_n' := Δ;
42         low 32 bits of w_n' := 1;
43         w_n.fetch_add(w_n');
44     Function UCT(int n) : <float>
45         ⟨w', n'⟩ := Get();
46         return w'/n' + 2C_p √(2 ln(n)/n')
47 Node;
```

**Algorithm 5.2:** The Lock-free UCT algorithm.

```
1  Function UCTSEARCH(Node* v_0, State s_0, budget)
2      while within search budget do
3          ⟨v_l, s_l⟩ := SELECT(v_0, s_0);
4          ⟨v_l, s_l⟩ := EXPAND(v_l, s_l);
5          Δ := PLAYOUT(v_l, s_l);
6          BACKUP(v_l,Δ);
7  Function SELECT(Node* v,State s) : <Node*,State>
8      while v.ISFULLYEXPANDED() do
9          ⟨w, n⟩ := v.GET();
10         v_l := argmax_{v_j ∈ children of v} v_j.UCT(n);
11         s := v.p takes action v_l.a from state s;
12         v := v_l;
13     return ⟨v, s⟩;
14 Function EXPAND(Node* v,State s) : <Node*,State>
15     if s is non-terminal then
16         actions := set of untried actions from state s;
17         v.CREATECHILDREN(actions);
18         v' := v.ADDCHILD();
19         if v' is not v then
20             v := v';
21             s := v.p takes action v.a from state s;
22     return ⟨v, s⟩;
23 Function PLAYOUT(Node* v,State s)
24     while s is non-terminal do
25         choose a ∈ set of untried actions from state s uniformly at random;
26         s := the current player p takes action a from state s;
27     Δ⟨v.p⟩ := reward for state s for each player p;
28     return Δ
29 Function BACKUP(Node* v,Δ) : void
30     while v is not null do
31         v.SET(Δ⟨v.p⟩);
32         v := v.parent;
```

that stores both the total simulation reward $Q(v)$ and the visit count $N(v)$, (4) the list of children, (5) the $is_parent$ flag (an atomic boolean) that shows whether the list of children is already created, (6) $n_nonexpanded_children$ the number of children that are not expanded yet, (7) the $is_expandable$ flag (an atomic Boolean) that shows whether $v$ is ready to be expanded, (8) the $is_fully_expanded$ flag (an atomic Boolean) that shows whether all children of $v$ are already expanded and (9) $parent$ that points to the parent of $v$. By using (a) the atomic variables, (b) the atomic operations, and (c) the associated memory models, we can solve all the three above cases of race conditions (SE, SB, and SBS).

- SE: To solve the SE race condition, the EXPAND operation in Algorithm 5.2 consists of two separate sub-operations: (A) the CREATECHILDREN operation and (B) the ADDCHILD operation. The first operation has four key steps (A-1, A-2, A-3, A-4) which are given in Algorithm 5.1. (A-1): Exchanging the value of $is_parent$ from $false$ to $true$ prevents the other threads to create the list of children (Line 12). Thus, the problem that the list of children is created by two threads at the same time is solved. (A-2): Creating the list of children (Line 14--18). (A-3): Set the value of $n_nonexpanded_children$ to counter $j$ (Line 19), (A-4): Set the value of $is_expandable$ to $true$ (Line 20). After a node successfully has become a parent, one of the non-expanded children in its list of children can be added using the ADDCHILD operation. The ADDCHILD operation in Algorithm 5.1 has three key steps (B-1, B-2, B-3). (B-1): Read the value of $is_expandable$ (Line 24), if it is $true$, try to expand a new child (Line 25--32). Otherwise, return the current node (Line 34). (B-2): The value of $index$ is calculated (Line 25), if it is zero, then node $v$ is fully expanded (Line 26). (B-3): $index$ shows the next child to be expanded (Line 31), if $index$ becomes negative, the current node is returned (Line 29).

- SB: To solve the SB race condition, Algorithm 5.1 uses a single 64-bit atomic integer $w_n$ for storing both variables $Q(v)$ and $N(v)$. The value of $Q(v)$ is stored in the high 32 bits of $w_n$, while the value of $N(v)$ is stored in the low 32 bits. This compression technique preserves the correct state of the variables $Q(v)$ and $N(v)$ in all threads because they should always be written together using a SET operation. Therefore, we have no faulty updates and guarantee consistency of computation.

- SBS: To solve the SBS race condition, Algorithm 5.2 always reads variable $w_n$ by a GET operation in the SELECT operation. The GET operation always reads the value of $Q(v)$ and $N(v)$ together. If a BACKUP operation wants to update the variable $w_n$ at the same time, it happens through a SET operation which

**Algorithm 5.3:** The pseudo-code of the GSCPM algorithm.

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

writes the value of $Q(v)$ and $N(v)$ together. Therefore, the values of $Q(v)$ and $N(v)$ are always correct, in the same state, and consistency of computation is guaranteed.

In Algorithm 5.2, each node $v$ is also associated with a state $s$. The state $s$ is recalculated as the SELECT and EXPAND steps descend the tree. The term $\Delta\langle p(v)\rangle$ denotes the reward after simulation for each player.

## 5.4 Implementation Considerations

We have implemented the proposed lock-free data structure and algorithm in the *ParallelUCT* package [MPvdHV15a]. The implementation is available online as part of the package. The ParallelUCT package is an open source tool for parallelization of the UCT algorithm (see Section 2.6). It uses *task-level parallelization* to implement different parallelization methods for MCTS. We have used an algorithm called *grain-sized control parallel MCTS* (GSCPM) to implement and measure the performance of the proposed lock-free UCT algorithm. The pseudo-code for GSCPM is given in Algorithm 5.3. The GSCPM algorithm is implemented by multiple methods from different parallel programming libraries such as C++11 STL, thread pool (*TPFIFO*), TBB (*task_group*) [Rei07], and Cilk Plus (*cilk_for* and *cilk_spwan*) [Rob13] in the ParallelUCT package. More details about each of these methods can be found in [MPvdHV15a].

## 5.5 Experimental Setup

Section 5.5.1 discusses our case study, Section 5.5.2 explains the performance metrics, and Section 5.5.3 provides the details of hardware.

### 5.5.1 The Game of Hex

The performance of the lock-free algorithm is measured by using the game of Hex. The game of Hex is described in Subsection 2.4.1. Below follows complementary information needed for this chapter. Hex is a board game with a diamond-shaped board of hexagonal cells [AHH10]. In our experiments, the game is played on a board of size 11 on a side, for a total of 121 hexagons [Wei17].

In our implementation of Hex, a disjoint-set data structure is used to determine the connected stones. Using this data structure the evaluation of the board position to find the player who won the game becomes very efficient [GI91].

### 5.5.2 Performance Metrics

In our experiments, the performance is reported by (A) playout speedup (or speedup) and (B) playing strength (or percentage of win). 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) are increasing.

### 5.5.3 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.

## 5.6 Experimental Design

The goal of this experiment is to measure the performance and scalability of a lock-free algorithm for parallel MCTS on both multi-core and many-core shared-memory machines. We do so using the ParallelUCT packages. The package implements, highly optimized, Hex playing program, in order to generate realistic real-world search spaces.

To generate statistically significant results in a reasonable amount of time, $2^{20}$ playouts are executed to choose a move. The board size is $11 \times 11$. The UCT constant $C_p$ is either 0, 0.1, or 1 in all of our experiments. To calculate the playout speedup the average of time over ten games is measured for making the first move of the game when the board is empty. The empty board is used because it has the biggest playout

time; it is the most time-consuming position (since the whole board should be filled randomly). The results are within less than 3% standard deviation.

## 5.7 Experimental Results

In Subsection 5.7.1 we discuss two topics. (A) the scalability is studied and the achieved playout speedup is reported, and (B) the effect of differences in values of $C_p$ parameters on the speedup of the parallel algorithm is measured. The performance of the proposed lock-free algorithm for Tree Parallelization when playing against Root Parallelization is reported in Subsection 5.7.2.

### 5.7.1 Scalability and $C_p$ parameters

As mentioned before, we are interested in strong scalability. Therefore, the search budget is fixed to $2^{20} = 1,048,576$ playouts as the number of tasks is increasing. Figure 5.4 shows the scalability of the algorithm for different parallel programming libraries on a CPU when the first move on the empty board is made. Each data point is the average of 21 games. Figure 5.4a illustrates the scalability when a coarse-grained lock is used (the graph is taken from [MPvdHV15a]) and Figure 5.4b demonstrates the scalability when the proposed lock-free method is used.

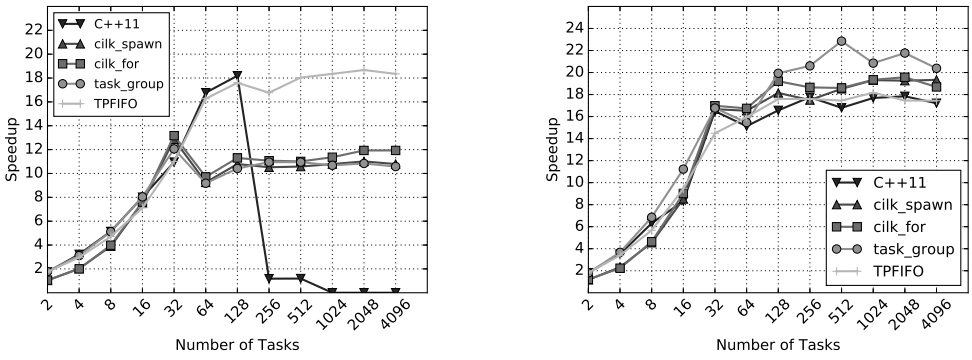
#### A: Playout Speedup

There are three main improvements when the lock-free tree is used (see A1 to A3).

- A1: the maximum speedup increases from 18 to 23.
- A2: the scalability of all methods is improved (it shows the notoriously bad effect of locks on the scalability for Cilk Plus, TBB, and C++11).
- A3: 32 tasks are sufficient to reach near 17 times speedup, while for the lock-based method at least 64 tasks are required.

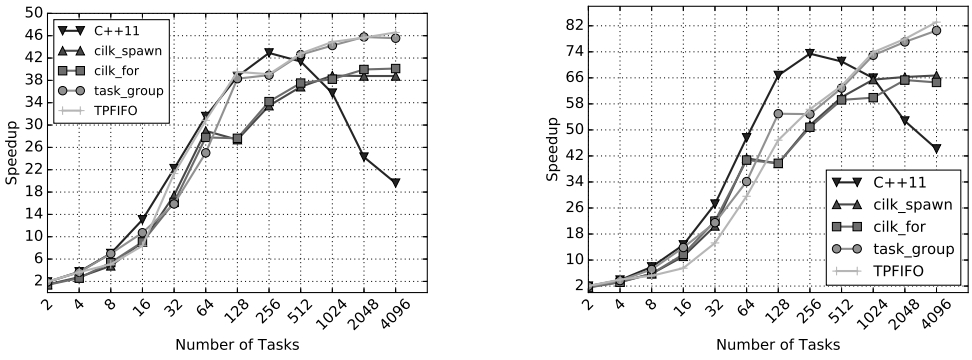
Figure 5.5 shows the scalability on the Xeon Phi. There are three main improvements when the lock-free tree is used (see A4 to A6).

- A4: the maximum speedup increases from 47 to 83.
- A5: the scalability of all methods is improved (it shows the notoriously bad effect of locks on the scalability for Cilk Plus, TBB, and C++11).
- A6: 64 tasks are sufficient to reach near 46 times speedup, while for the lock-based method at least 2048 tasks are required.




(a) (b)

Figure 5.4: The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$. (5.4a) Coarse-grained lock. (5.4b) Lock-free.




(a) (b)

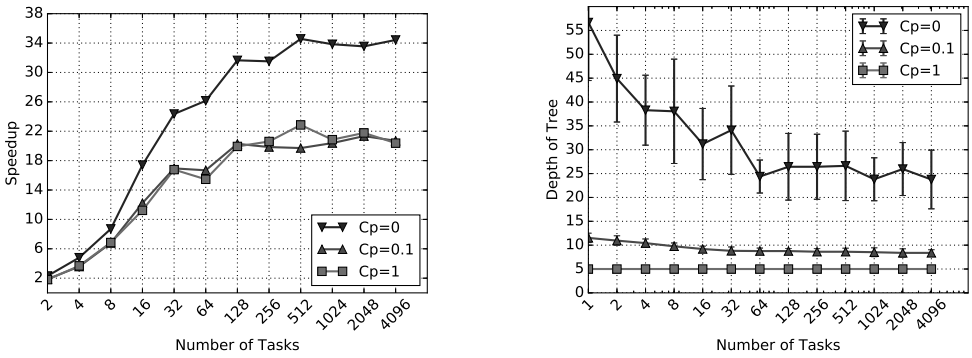
Figure 5.5: The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$ on the Xeon Phi. (5.5a) Coarse-grained lock. (5.5b) Lock-free.

**B: The Effect of $C_p$ on Playout Speedup**

Table 5.1 shows the execution time of the sequential UCT algorithm for three different $C_p$ values. It is observed that the execution time is decreasing as the value of $C_p$ is increasing. There is an obvious explanation for this behavior. When the algorithm uses high exploitation (i.e., low value for $C_p$), it constructs a search tree that is deeper and more asymmetric. In Figure 5.6b, the depth of the tree is 56 when the number of tasks is 1 and $C_p = 0$. When the shape of the tree is more asymmetric, each iteration of

Table 5.1: Sequential execution time in seconds.

| $C_p$ | Time (s) | Depth of Tree (Avg.) |
|---|---|---|
| 0 | $59.97 \pm 10.93$ | $56.66 \pm 12.16$ |
| 0.1 | $26.66 \pm 0.81$ | $11.52 \pm 0.98$ |
| 1 | $20.7 \pm 0.3$ | 5 |




(a) (b)

Figure 5.6: (5.6a) The scalability of the algorithm for different $C_p$ values. (5.6b) Changes in the depth of tree when the number of tasks are increasing.

the algorithm must traverse a deeper path of nodes inside the tree using the SELECT operation until it can perform a PLAYOUT operation. The SELECT operation consists of a *while loop* which for a tree with a depth of 56 has to perform 56 iterations in the worst case (see Algorithm 5.2). The BACKUP operation also consists of a while loop which for a deeper tree has more iterations. These two operations are also memory intensive ones (i.e., accessing the nodes of the tree which reside in memory). The results are that the execution time of the sequential algorithm becomes higher for high exploitation. Increasing the value of $C_p$ means more exploration and thus a more symmetric tree with a lower depth. In Figure 5.6b, the depth of the tree is 5 when the number of tasks is 1 and $C_p = 1$. In this case, the while loop in the SELECT operation has to perform only 5 iterations in the worst case.

We have measured the scalability of the proposed lock-free algorithm for different $C_p$ values (see Figure 5.6a). The sequential time for each $C_p$ in Table 5.1 is used as the baseline. The maximum speedup for $C_p = 0$ is around 34. It is much higher than 23 times, the speedup when $C_p = 1$. There is a possible explanation for the

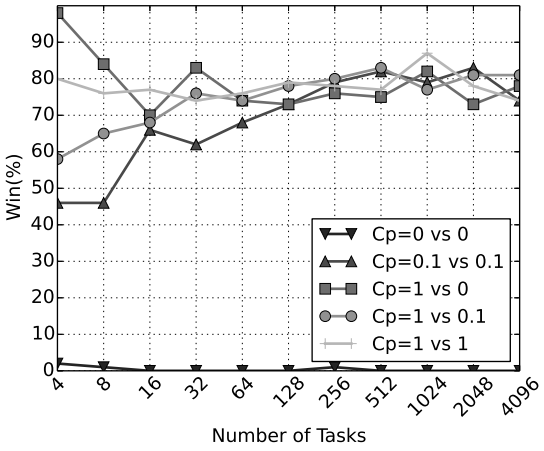



Figure 5.7: The playing results for lock-free Tree Parallelization versus Root Parallelization. The first value for $C_p$ is used for Tree Parallelization and the second value is used for Root Parallelization.

higher speedup. The parallel algorithm may be more efficient than the equivalent serial algorithm, since the parallel algorithm may be able to avoid work that in every serialization would be forced to be performed [MRR12]. For example, Figure 5.6b shows the changes in the depth of the constructed tree with regards to the number of tasks for three different values for $C_p$. Increasing the number of tasks reduces the depth of the tree from 56, when the serial execution is exploitative (i.e., $C_p = 0$), to around 25. It means that, in parallel execution (a) threads explore different branches of the tree and (b) the tree is more symmetric compared to the serial execution. Hence, the number of iterations in both SELECT and BACKUP operations reduces in parallel execution and therefore causes a higher speedup. When the serial execution has high exploration (i.e., $C_p = 1$), increasing the number of tasks does not change the depth of the tree.

### 5.7.2 GSCPM vs. Root Parallelization

Figure 5.7 presents the result of playing Hex between the proposed lock-free Tree Parallelization against Root Parallelization. Root parallelization is also a parallelization method that does not use locks because it uses an ensemble of independent search trees. Therefore, it is interesting to see the performance of the proposed lock-free algorithm versus Root Parallelization. Figure 5.7 reports the percentage of wins for lock-free Tree Parallelization for five different combinations of $C_p$. Both methods use

the same number of tasks. For each data point, 100 games are played.

When $C_p = 0$ for both algorithms, Tree Parallelization cannot win against Root Parallelization. It shows that the high speedup for $C_p = 0$ (see Figure 5.6a) is not useful. However, when the value of $C_p$ is selected to be more exploratory, the lock-free Tree Parallelization is superior to Root Parallelization, specifically for a higher number of tasks.

## 5.8 Answer to RQ3

In this chapter we presented the lock-free tree data structure for parallelization of MCTS. As such, this chapter proposes solutions for the following question.

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

To answer *RQ3* we have found our way step by step. We did so in three steps.

First, we remark that the existing Tree Parallelization algorithm for MCTS uses a shared search tree to run the iterations in parallel (see Subsection 5.1.1). Here we observe that the shared search tree has potential race conditions (see Subsection 5.1.2).

Our second step is to overcome this obstacle (see Section 5.3). In this section, we have shown that having a correct lock-free data structure is possible. To achieve this goal we have used methods from modern memory models and atomic operations (see Section 5.3). Using these methods allows removing of synchronization overhead. Hence, we have implemented the new lock-free algorithm that has no race conditions (see Section 5.4).

The third step was to evaluate the lock-free algorithm. Therefore we performed an extensive experiment in a small area (Hex on a $11 \times 11$ board), see Sections 5.5 and 5.6.

To conclude, the experiment showed that the lock-free algorithm had a better performance and scalability when compared to other synchronization methods (see Section 5.7). The performance of task-level parallelization to implement the lock-free GSCPM algorithm on a multi-core machine with 24 cores was very good. It reached a speedup of 23 and showed very good scalability for up to 4096 tasks. The performance on a many-core co-processor was also very good; a speedup of 83 on the 61 cores of the Xeon Phi was reached. In summary, the Xeon Phi showed very good scalability for up to 4096 tasks.

# 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,

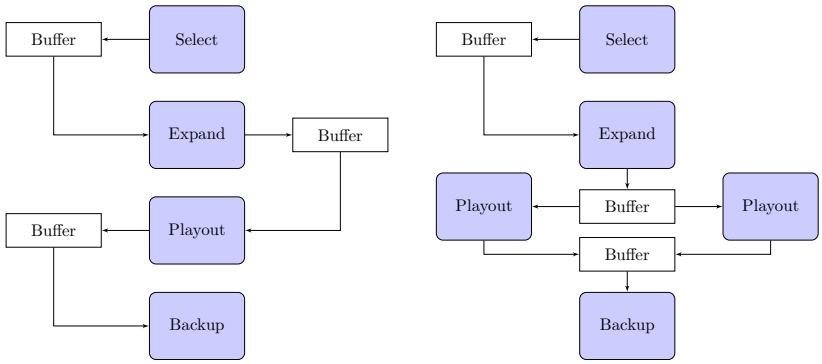



Figure 6.1: (6.1a) Flowchart of a pipeline with sequential stages for MCTS. (6.1b) Flowchart of a pipeline with parallel stages for MCTS.

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.

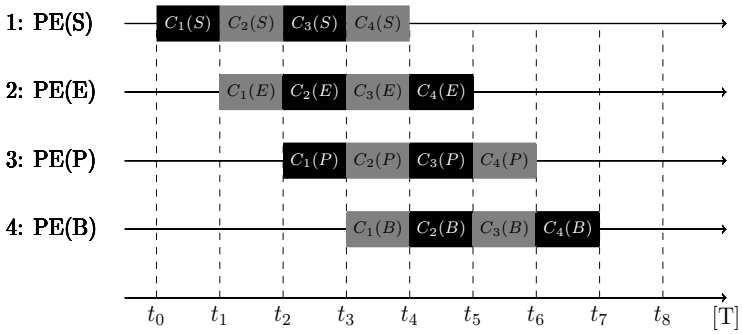



Figure 6.2: Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are equal.

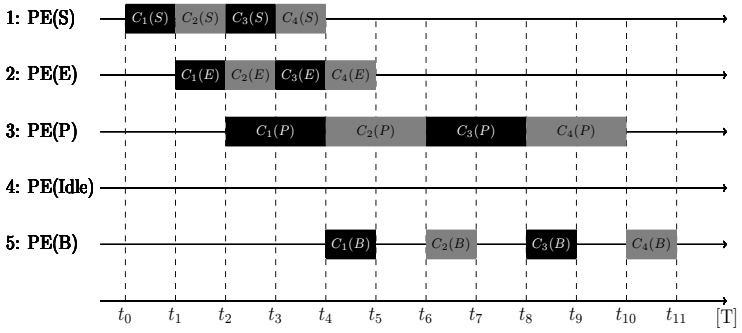



Figure 6.3: Scheduling diagram of a pipeline with sequential stages for MCTS. The computations for stages are not equal.

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 $j$th 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

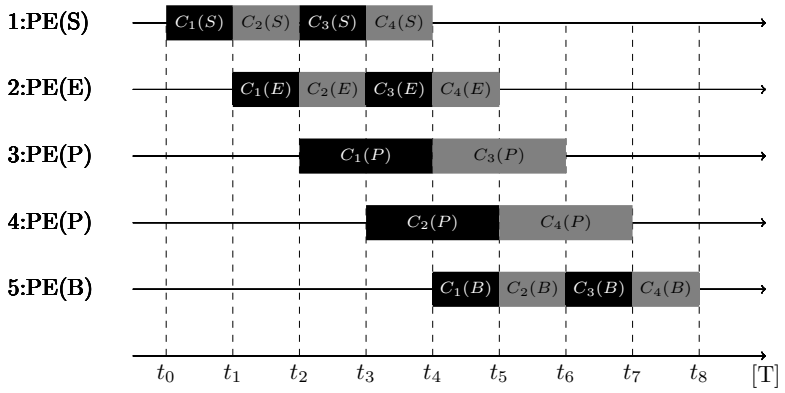



Figure 6.4: Scheduling diagram of a pipeline with parallel stages for MCTS. Using parallel stages create load balancing.

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.

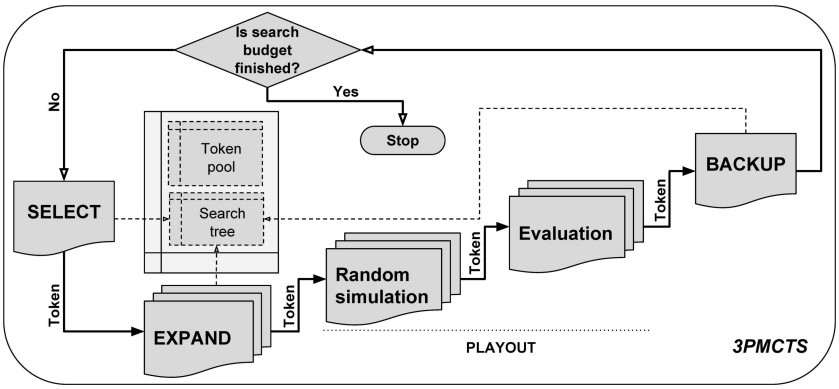



Figure 6.5: The 3PMCTS algorithm with a pipeline that has three parallel stages (i.e., EXPAND, RANDOMSIMULATION, and EVALUATION).

### 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($\sigma$) 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., $\Delta$) 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 | $215.72 \pm 4.12$ |

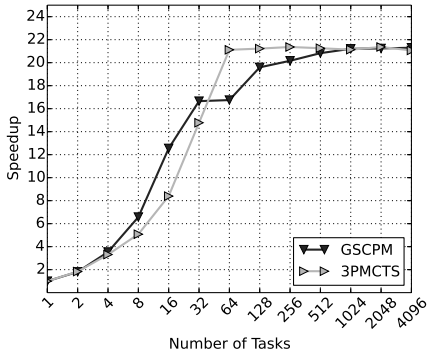



Figure 6.6: Playout-speedup as function of the number of tasks (tokens). Each data point is an average of 21 runs for a search budget of 8192 playouts. The constant $C_p$ is 0.5. Here a higher value is better.

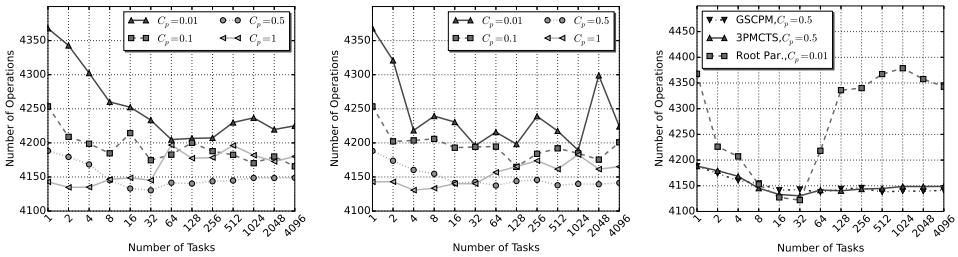
## 6.6 Experimental Results

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.

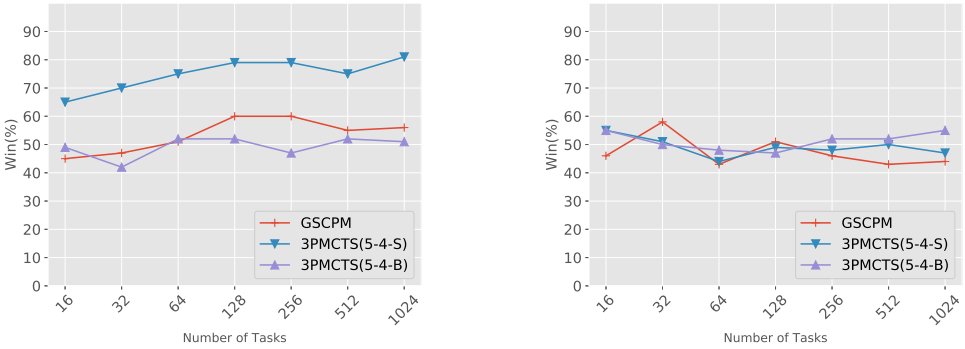
| $C_p$ | Player a | Player b |
|---|---|---|
| 0.01 | GSCPM<br>3PMCTS(5-4-S)<br>3PMCTS(5-4-B) | GSCPM with 8 tasks |
| 1 | GSCPM<br>3PMCTS(5-4-S)<br>3PMCTS(5-4-B) | GSCPM with 8 tasks |

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 ($11 \times 11$ 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*.

# 7

# Ensemble UCT Needs High Exploitation

The last research question is *RQ5* (see our research design in Section 1.7).

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

This research question is derived from the fact that the quality of search in MCTS depends on the balance between exploitation (look in areas which appear to be promising) and exploration (look in areas that have not been well sampled yet). The most popular algorithm in the MCTS family which addresses this dilemma is UCT [KS06] (see Section 2.2). Parallelization of MCTS intends to decrease the execution time of the algorithm, but it also affects the exploitation-exploration balance. A set of solutions has been developed to control the exploitation-exploration balance when parallelizing MCTS to improve the quality of search [BPW+12, KPVvdH13]. We partition the set of solutions into two parts, (1) adjusting the exploitation-exploration balance with respect to the tree size, and (2) adjusting the exploitation-exploration balance by an artificial increase in exploration, called *virtual loss*. We provide an answer for *RQ5* with respect to these two parts. This chapter [1] investigates the application of the first solution to Root Parallelization. In Chapter 8, we analyze the use of the second solution on the lock-free Tree Parallelization algorithm.

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, Ensemble UCT Needs High Exploitation, in Proceedings of the 8th International Conference on Agents and Artificial Intelligence, 2016, pp. 370--376.

Hence, we start investigating the adjustment of the exploitation-exploration balance with respect to the tree size. As with most sampling algorithms, one way to improve the quality of the result is to increase the number of samples and thus enlarge the size of the MCTS tree. However, constructing a single large search tree with $t$ samples or playouts is a time-consuming process (see Subsection 2.3.2). A solution for this problem is to create a group of $n$ smaller trees that each has $t/n$ playouts and search these in parallel. This approach is used in Root Parallelization [CWvdH08a] and in Ensemble UCT [FL11] (from now on we use these two names interchangeably). In both Root Parallelization and Ensemble UCT, multiple independent UCT instances are constructed. At the end of the search process, the statistics of all trees are combined to yield the final result [BPW$^+$12]. However, there is contradictory evidence on the success of Ensemble UCT [BPW$^+$12]. On the one hand, Chaslot et al. found that, for Go, Ensemble UCT (with $n$ trees of $t/n$ playouts each) outperforms a plain UCT (with $t$ playouts) [CWvdH08a]. On the other hand, Fern and Lewis were not able to reproduce this result in other domains [FL11]. They found situations where a plain UCT outperformed Ensemble UCT given the same total number of playouts. We aim to shed light on this controversy using an idea from [KPVvdH13]. Kuipers et al. argued that when the tree size in MCTS is small, more exploitation should be chosen, and with larger tree sizes, high exploration is suitable [KPVvdH13]. Therefore, the main contribution of this chapter is to show that this idea can be used in Ensemble UCT to improve its search quality by adjusting the $C_p$ parameter depending on the ensemble size.

The remainder of the chapter is organized as follows. Section 7.1 describes Ensemble UCT. Section 7.2 discusses related work. Section 7.3 gives the experimental setup, Section 7.4 describes the experimental design, and Section 7.5 provides the experimental results for this study.

## 7.1 Ensemble UCT

Ensemble UCT or the Root Parallelization algorithm belongs to the category of parallel algorithms with more than one data structure (see Subsection 2.3.2). It creates an ensemble of search trees (i.e., one for each thread). The trees are independent of each other. When the search is over, they are merged, and the action of the best child of the root is selected to be performed.

Ensemble UCT is given its place in the overview article by [BPW$^+$12]. Table 7.1 shows different possible configurations for Ensemble UCT. Each configuration has its benefits. The total number of playouts is $t$, and the size of the ensemble (number of trees inside the ensemble) is $n$. It is assumed that $n$ processors are available with $n$

Table 7.1: Different possible configurations for Ensemble UCT. Ensemble size is $n$.

| Number of playouts | | | playout speedup | | playing strength |
|---|---|---|---|---|---|
| UCT | Ensemble UCT | | n cores | 1 core | |
| | Each tree | Total | | | |
| $t$ | $t$ | $n \cdot t$ | 1 | $\frac{1}{n}$ | Yes, known |
| $t$ | $\frac{t}{n}$ | $t$ | $n$ | 1 | ? |

equal to the ensemble size.

The third line of Table 7.1 shows the situation where Ensemble UCT has $n \cdot t$ playouts in total, while plain UCT has only $t$ playouts. In this case, there would be no speedup in a parallel execution of the ensemble approach on $n$ cores, but the larger search effort would presumably result in a better search result. We call this use of parallelism *playing strength* (see Subsection 2.5.2). The fourth line of Table 7.1 shows a different possible configuration for Ensemble UCT. In this case, the total number of playouts for both UCT and Ensemble UCT is equal to $t$. Thus, each core searches a smaller tree of size $t/n$. The search will be $n$ times faster (the ideal case). We call this use of parallelism *Playout speedup* (see Subsection 2.5.1). It is important to note that in this configuration both approaches take the same amount of time on a single core. However, there is still the question whether we can reach any *playing strength*. This question will be answered in Section 7.5 as the first part of RQ5.

## 7.2 Related Work

From the introduction of this chapter we know that [CWvdH08a] provided evidence that, for Go, Root Parallelization with $n$ instances of $t/n$ iterations each outperforms plain UCT with $t$ iterations, i.e., Root Parallelization (being a form of Ensemble UCT) outperforms plain UCT given the same total number of iterations. However, in other domains, [FL11] did not find this result. [SKW10] also analyzed the performance of root parallelization in detail. They found that a majority voting scheme gives better performance than the conventional approach of playing the move with the greatest total number of visits across all trees. They suggested that the findings in [CWvdH08a] are explained by the fact that Root Parallelization performs a shallower search, making it easier for UCT to escape from local optima than the deeper search performed by plain UCT (see also Section 8.2, in relation with part two).

In Root Parallelization each process does not build a search tree larger than the sequential UCT. Moreover, each process has a local tree, which contains characteristics that differ from tree to tree. Rather recently, [TD15] proposed a new idea by

distinguishing between tactical behavior and strategic behavior. They transferred the RAVE (Rapid Action Value Estimate) ideas as developed by [GS07], from the selection phase to the simulation phase. This implies that influencing the tree policy is changed into also influencing the Monte-Carlo policy.

Fern and Lewis 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 [FL11]. So, our task is to explain the differences in their work and that by [CWvdH08a].

## 7.3 Experimental Setup

Section 7.3.1 discusses our case study and Section 7.3.2 provides the details of hardware.

### 7.3.1 The Game of Hex

The game of Hex is described in Subsection 2.4.1. Below follows complementary information needed for this chapter. The $11 \times 11$ Hex board is represented by a disjoint-set. This data structure has three operations *MakeSet*, *Find* and *Union*. In the best case, the amortized time per operation is $O(\alpha(n))$, where $\alpha(n)$ denotes the inverse Ackermann function. The value of $\alpha(n)$ is less than 5 for all remotely practical values of $n$ [GI91].

In Ensemble UCT, each tree performs a completely independent UCT search with a different random seed. To determine the next move to play, the number of wins and visits of the root's children of all trees are collected. For each child the total sum of wins and the total sum of visits are computed. The child with the largest number of wins/visits is selected.

The plain UCT algorithm and Ensemble UCT are implemented in the ParalellUCT package. In order to make our experiments as realistic as possible, we use the ParallelUCT program for the game of Hex [MPVvdH14, MPvdHV15a]. This program is highly optimized, and reaches a speed of more than 40,000 playouts per second per core on a 2,4 GHz Intel Xeon processor (see Section 2.6).

### 7.3.2 Hardware

The results were measured on a dual socket machine with 2 Intel Xeon E5-2596v2 processors running at 2.40GHz. Each processor has 12 cores, 24 hyperthreads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The pack TurboBoost frequency

Table 7.2: The performance of Ensemble UCT vs. plain UCT based on win rate.

| Approach | Win (%) | Performance vs. plain UCT | Playing Strength |
|---|---|---|---|
| Ensemble UCT | $< 50$ | Worse than | No |
| | $= 50$ | As good as | No |
| | $> 50$ | Better than | Yes |

is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program.

## 7.4 Experimental Design

As Hex is a 2-player game, the playing strength of Ensemble UCT is measured by playing versus a plain UCT with the same number of playouts. We *expect* to see an improvement for the Ensemble UCT playing strength against plain UCT by choosing 0.1 as the value of $C_p$ (high exploitation) when the number of playouts is small. We start our experiments by setting the value of $C_p$ to 1.0 for plain UCT (high exploration). Note that for the purpose of this research, it is not essential to find the optimal value of $C_p$, but to show the difference in effect on the performance when $C_p$ is varying.

The board size for Hex is $11 \times 11$. In our experiments, the maximum ensemble size is $2^8 = 256$. Thus, for $2^{17}$ playouts, when the ensemble size is 1, there are $2^{17}$ playouts per tree and when the ensemble size is $2^6 = 64$ the number of playouts per tree is $2^{11}$. Throughout the experiments, the ensemble size is multiplied by a factor of two.

## 7.5 Experimental Results

Our experimental results show the percentage of wins for Ensemble UCT with a particular ensemble size and a particular $C_p$ value. In Figure 7.1 results are shown, with $C_p$ =0 (only exploitation) and ensemble size equals 8. Each data point represents the average of 200 games with a corresponding 99% confidence interval. Table 7.2 summarizes how the performance of Ensemble UCT versus plain UCT is evaluated. The concept of *high exploitation for small UCT tree* is significant if Ensemble UCT reaches a win rate of more than 50%. (Section 7.5 will show that this is indeed the case.)

Below we provide our experimental results. We distinguish them into (A) hidden exploration in Ensemble UCT and (B) exploitation-exploration trade-off for Ensemble UCT.

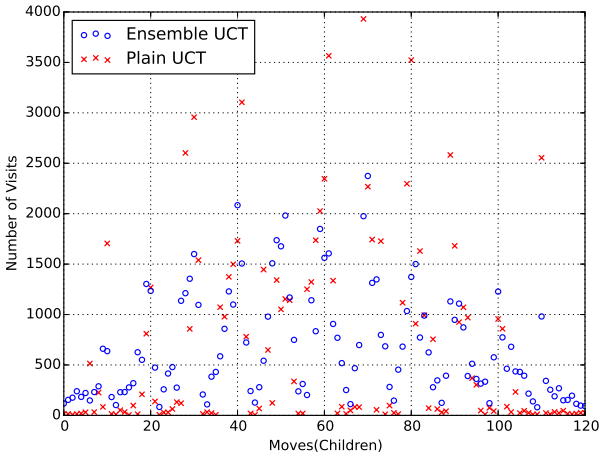
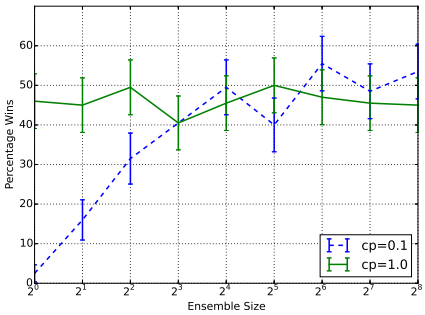



Figure 7.1: The number of visits for root's children in Ensemble UCT and plain UCT. Each child represents an available move on the empty Hex board with size $11 \times 11$. Both Ensemble UCT and plain UCT have 80,000 playouts and $C_p = 0$. In Ensemble UCT, the size of the ensemble is 8.

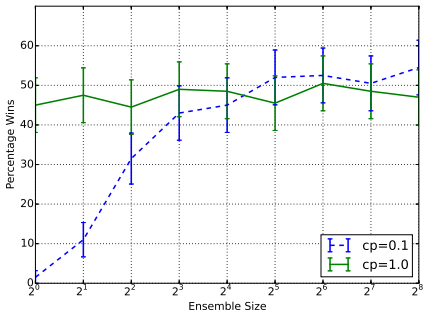
## A: Hidden Exploration in Ensemble UCT

It is important to understand that Ensemble UCT has a hidden exploration factor by nature. Two reasons are: (1) each tree in Ensemble UCT is independent, and (2) an ensemble of trees contains more exploration than a single UCT search with the same number of playouts would have. The hidden exploration is because each tree in Ensemble UCT searches in different areas of the search space.

In Figure 7.1 the difference in exploitation-exploration behavior of the Ensemble UCT and plain UCT is shown in the number of visits that one of the root's children counts when using one of the algorithmic approaches with $C_p = 0$. Both Ensemble UCT [BPW$^+$12] and plain UCT [BPW$^+$12] have 80,000 of playouts. In each experiment, a search tree for selecting the first move on an empty board is constructed. Each of the children corresponds to a possible move of an empty Hex board (i.e., 121 moves). Ensemble UCT is more explorative compared to plain UCT if it generates more data points with more distance from the $x$-axis than plain UCT. In Ensemble UCT the number of playouts is distributed among 8 separate smaller trees. Each of the trees has 10,000 playouts and for each child the number of visits is collected. When the value of $C_p$ is 0, which means the exploration part of the UCT formula is turned off, all possible moves in the Ensemble UCT receive at least a few visits. While for plain UCT with 80,000 playouts and $C_p = 0$ there are many of the moves with




(a) The total number of playouts is $2^{17} = 131072$




(b) The total number of playouts is $2^{18} = 262144$

Figure 7.2: The percentage of wins for ensemble UCT is reported. The value of $C_p$ for plain UCT is always 1.0 when playing against Ensemble UCT. To the left few large UCT trees, to the right many small UCT trees.

*no* visits. The data points when using plain UCT are closer to the x-axis compared to Ensemble UCT. However, for Ensemble UCT the peak is 2400, while it is 4000 visits for plain UCT. It means that plain UCT is more exploitative.

## B: Exploitation-Exploration trade-off for Ensemble UCT

Below we discuss two experiments: (B1) an experiment with $2^{17}$ playouts and (B2) an experiment with $2^{18}$ playouts. In Figures 7.2a and 7.2b, from the left side to the right side of the graph, the ensemble size (the number of search trees per ensemble) increases by a factor of two, and the number of playouts per tree (tree size) decreases by the same factor. Thus, at the right-hand side of the graph, we have the largest ensemble with the smallest trees. The total number of playouts always remains the same throughout an experiment for both Ensemble UCT and plain UCT. The value of $C_p$ for plain UCT is always 1.0, which means high exploration.

## B1: Experiment with $2^{17}$ playouts

Figure 7.2a shows the *relations* between the value of $C_p$ and the ensemble size, when both plain UCT and Ensemble UCT have the same number of total playouts. Moreover, Figure 7.2a shows the *performance* of Ensemble UCT for different values of $C_p$. It shows that when $C_p = 1.0$ (highly explorative) Ensemble UCT performs as good as (or mostly worse than) plain UCT. When Ensemble UCT uses $C_p = 0.1$ (highly exploitative) then for small ensemble sizes (large sub-trees) the performance of En-

semble UCT sharply drops down. By increasing the ensemble size (smaller sub-trees), the performance of Ensemble UCT keeps improving until it becomes as good as or even better than plain UCT.

### B2: Experiment with $2^{18}$ playouts

A second experiment is conducted using $2^{18}$ playouts to investigate the effect of enlarging the number of playouts on the performance of Ensemble UCT. Figure 7.2b shows that when for this large number of playouts the value of $C_p = 1.0$ is high (i.e., highly explorative) the performance of Ensemble UCT cannot be better than plain UCT, while for a small value of $C_p = 0.1$ (i.e., highly exploitative) the performance of Ensemble UCT is almost always better than plain UCT when the ensemble size is $2^5$ or larger. Hence, our conclusion is that there exists a marginal playing strength. The potential playout speedup could be up to the ensemble size if a sufficient number of processing cores is available.

## 7.6 Answer to the First Part of RQ5

This chapter aims at answering the first part of *RQ5* (i.e., adjusting the exploitation-exploration balance with respect to the tree size) of the following question.

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

The chapter described an empirical study on Ensemble UCT with different sets of configurations for the ensemble size, the tree size, and the exploitation-exploration trade-off. Previous studies on Ensemble UCT/Root Parallelization provided inconclusive evidence on the effectiveness of Ensemble UCT (see the beginning of the chapter).

Our results suggest that the reason for uncertainty (concerning the controversy in the previous studies) lies in the exploitation-exploration trade-off in relation to the size of the sub-trees. With this knowledge, it is explainable that [CWvdH08a] found an improvement in their Root Parallelization for Go (which has big search trees for small ensemble sizes where exploration can open new perspectives). For [FL11], it is also explainable that they did not arrive at the same success in other domains (which have small search trees for large ensemble sizes). Our experiments for Ensemble UCT now confirm earlier ideas as provided by [KPVvdH13] on this topic. In summary, our results provide clear evidence that the performance of Ensemble UCT is improved by selecting higher exploitation for smaller search trees given a fixed time-bound or fixed number of simulations.

Our work is particularly motivated, in part, by the observation in [CWvdH08a] of super-linear speedup in Root Parallelization. Finding super-linear speedup in two-agent games occurs infrequently. Most studies in parallel game-tree search report a battle against search overhead, communication overhead (e.g., [Rom01]), synchronization overhead, and deployment overhead (see, Chapter 1). For super-linear speedup to occur, the parallel search must search fewer nodes than the sequential search.

# 8

# An Analysis of Virtual Loss in Parallel MCTS

We reiterate the last research question, *RQ5*, and continue the research work started in Chapter 7.

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

In part one of *RQ5* (see Chapter 7) we investigated to what extent the successes of MCTS depend on the balance between exploitation and exploration (see also Section 2.2). The parallelization of MCTS intends to decrease the execution time of the algorithm, but it also affects this trade-off. Therefore, solutions are developed to control the exploitation-exploration balance when parallelizing MCTS to improve the quality of search [CWvdH08a, BPW+12, KPVvdH13]. We have partitioned the set of solutions into two parts, (1) adjusting exploitation-exploration balance with respect to the tree size, and (2) adjusting the exploitation-exploration balance by an artificial increase in exploration called *virtual loss*. We provided an answer for *RQ5* (part one) in Chapter 7. This chapter[1] addresses the second part of *RQ5*.

Each iteration of the MCTS algorithm adds a new node to a tree by first selecting a path inside the tree and then using Monte Carlo simulations. This iterative process is path-dependent, which means that the outcomes of previous iterations guide the

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, An Analysis of Virtual Loss in Parallel MCTS, in Proceedings of the 9th International Conference on Agents and Artificial Intelligence, 2017, pp 648--652.

future selections. Rather recently, several studies have addressed the topic of making parallel methods for MCTS, such as Tree Parallelization and Root Parallelization [BG11, BPW+12, SHM+16, SSS+17]. Here we focus on Tree Parallelization that distributes different iterations of MCTS among parallel workers. Therefore, it has to violate the path dependency feature of sequential MCTS to make the algorithm faster.

In Tree Parallelization, the performance is decreasing when increasing the number of parallel workers. It is widely believed that part of the performance loss is due to a redundant search being done by separate parallel workers (i.e., Search Overhead). However, if the parallel algorithm is using a lock to guarantee synchronization, the contention among parallel workers also contributes to the performance loss. Therefore, a method called *virtual loss* is proposed for lock-based Tree Parallelization [CWvdH08a]. It forces parallel workers to traverse different paths inside the MCTS tree to avoid contention around a particular node. However, virtual loss then affects the balance between exploitation and exploration in the UCT algorithm by increasing the exploration level irrespective of the value of the $C_p$ parameter.

In this chapter, we evaluate the benefit of using the virtual loss (i.e., an artificial increase in exploration against exploitation) for lock-free (instead of locked-based) Tree Parallelization. We carry out our experiments for a full range of exploitation-exploration in UCT (i.e., the $C_p$ parameter) and a varying number of parallel workers. The result is reported concerning Search Overhead (SO) and Time Efficiency (Eff). The case studies are problems from the High Energy Physics domain.

The remainder of the chapter is organized as follows. The virtual loss method is explained in Section 8.1, the related work is presented in Section 8.2, the experimental setup is described in Section 8.3, followed by the experimental design in Section 8.4, and the experimental results in Section 8.5. Finally, the answer to the second part of *RQ5* is given in Section 8.6, with the complete answer to *RQ5* in Section 8.7.

## 8.1 Virtual Loss

In Tree Parallelization one MCTS tree is shared among several threads that are performing simultaneous searches [CWvdH08a]. The main challenge in this method is using data locks to prevent data corruption. A lock-free implementation of this algorithm addresses the problem as mentioned earlier with better scaling than a locked approach [EM10]. Therefore, in our implementation of Tree Parallelization, locks are removed.

**Definition 8.1 (Virtual Loss)** *Virtual loss is a method to make a node in the tree less favorable to be selected and therefore force parallel workers to traverse different paths inside the MCTS tree.*

Here we note that in Tree Parallelization with fine-grained locks (see Subsection 5.2.1), it is still possible that different threads traverse the tree in mostly the same way. This phenomenon causes thread contention when two different threads visit the same node concurrently, and one thread is waiting for a lock that is currently being held by another thread. Increasing the number of threads exacerbates this problem. [CWvdH08a] suggested a solution to assign a temporary *virtual loss* (a marker) to a node when a thread selects it. Without the marker, there is a higher chance for thread contention.

Implementing the virtual loss is straightforward. A thread is selecting a path inside the tree to find a leaf node. It is reducing the UCT value of all the nodes that belong to the path, assuming that the playout from the leaf node results in a loss. Therefore, the virtual loss will inspire other threads to traverse different paths and avoid contention. A thread removes the assigned virtual loss immediately before the backup step when updating the nodes with the real playout result. It is worth mentioning that Tree Parallelization with virtual loss is more explorative compared to plain Tree Parallelization because the virtual loss encourages different threads to explore different parts of the tree regardless of the value of $C_p$. Regarding the virtual loss, $UCT(j)$ decreases as more threads select node $j$, which encourages other threads to favor other nodes. Algorithm 8.1 gives the pseudocode for the virtual loss technique.

**Algorithm 8.1:** The lock-free UCT algorithm with virtual loss.

```
1  Function UCTSEARCH(Node* v_0, State s_0, budget)
2      while within search budget do
3          ⟨v_l, s_l⟩ := SELECT(v_0, s_0);
4          ⟨v_l, s_l⟩ := EXPAND(v_l, s_l);
5          Δ := PLAYOUT(v_l, s_l);
6          BACKUP(v_l,Δ);
7  Function SELECT(Node* v, s) : <Node*,State>
8      while v.ISFULLYEXPANDED() do
9          ⟨w, n⟩ := v.GET();
10         v_l := argmax_{v_j ∈ children of v} v_j.UCT(n);
11         s := v.p takes action v_l.a from state s;
12         v_l.SET(+LOSS(v_l.p));
13         v := v_l;
14     return ⟨v, s⟩;
15 Function BACKUP(Node* v,Δ) : void
16     while v is not null do
17         v.SET(−LOSS(v.p));
18         v.SET(Δ⟨v.p⟩);
19         v := v.parent;
```

## 8.2 Related Work

[CWvdH08a] reported that Tree Parallelization with local locks and virtual loss performs as well as Root Parallelization in the game of Go. However, [SCP+14] suggested that adding a virtual loss to Tree Parallelization with local locks has almost no effect on the performance for the game of Lords of War. [Seg11] showed that MCTS could scale nearly perfectly to at least 64 threads when combined with virtual loss, but without virtual loss scaling is limited to just eight threads. [EMAS10] showed that the virtual loss technique is very effective for Go in FUEGO. However, [BG11] found that increasing exploration by multiple virtual losses *slightly* improves Tree Parallelization with lock-free updates for Go in Pachi.

## 8.3 Experimental Setup

We perform a sensitivity analysis of $C_p$ on the number of iterations for different thread configurations for one expression, namely HEP($\sigma$) which is a polynomial from the HEP domain with 15 variables [Ver13, KPVvdH13, RVPvdH14]. The plain UCT algorithm and parallel methods are implemented in the ParallelUCT package.

The results are measured on a dual socket machine with 2 Intel Xeon E5-2596v2 processors running at 2.40GHz. Each processor has 12 cores, 24 hyper-threads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The pack TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program.

## 8.4 Experimental Design

In our case study, we investigate Horner schemes. We consider a Horner Scheme as an optimization problem (see Subsection 2.4.2). The playing strength of Tree Parallelization for the Horner scheme is measured by the number of operations that are found for a number of playouts (see Subsection 2.5.2). Here, we define search overhead ($(SO)$) and time efficiency ($Eff$) based on the number of playouts.

$$SO = \frac{\textit{number of playouts}_{parallel}}{\textit{number of playouts}_{sequential}} - 1. \tag{8.1}$$

$$Eff = \frac{\textit{time}_{sequential}}{\textit{number of parallel workers} \cdot \ \textit{time}_{parallel}}. \tag{8.2}$$

In our experiments, the algorithm stops when it found 4,150 operations, or the limit of 10,240 playouts is reached. The numbers are set at 4,150 and 10,240 as

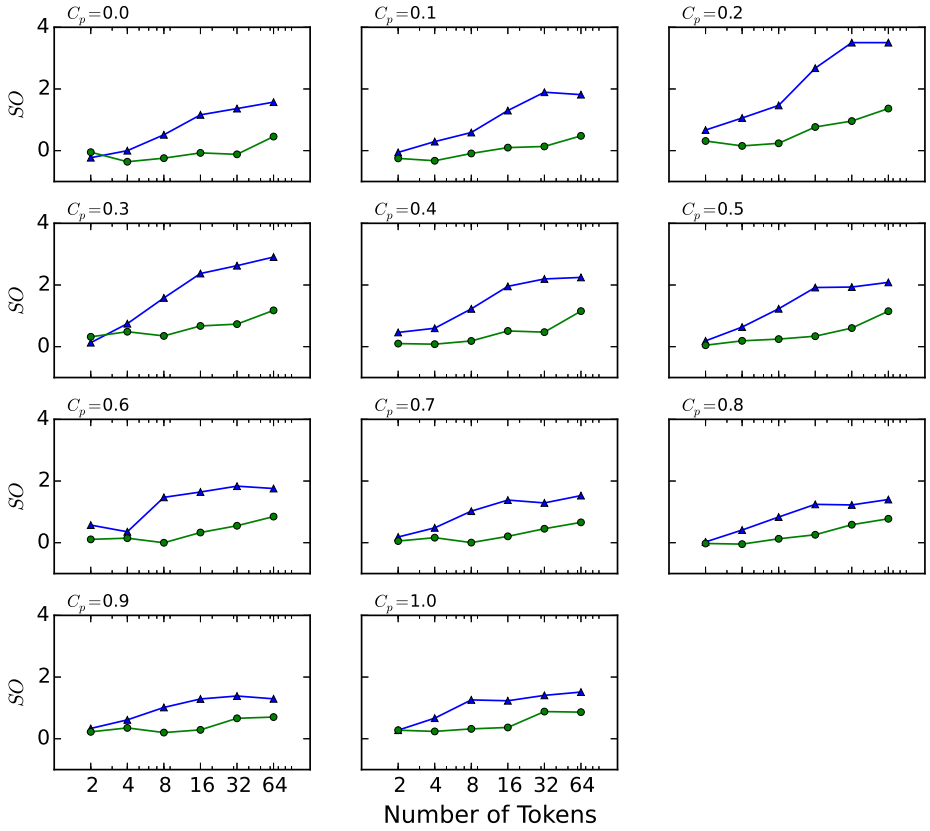



Figure 8.1: Search overhead ($SO$) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles, and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that the higher $SO$ of Tree Parallelization with virtual loss means lower performance.

"relaxed" upper bound above 4,000 and 10,000 which are found by [KPVvdH13] for the HEP($\sigma$) polynomial. Throughout the experiments, the number of tokens or tasks is multiplied by a factor of two. Each data point represents the average of 20 runs.

## 8.5 Experimental Results

Below we provide our experimental results. The first factor is Search Overhead (SO). We hope for the reduction of $SO$ by using virtual loss. Figure 8.1 shows the $SO$ of plain Tree Parallelization and Tree Parallelization with virtual loss for different values

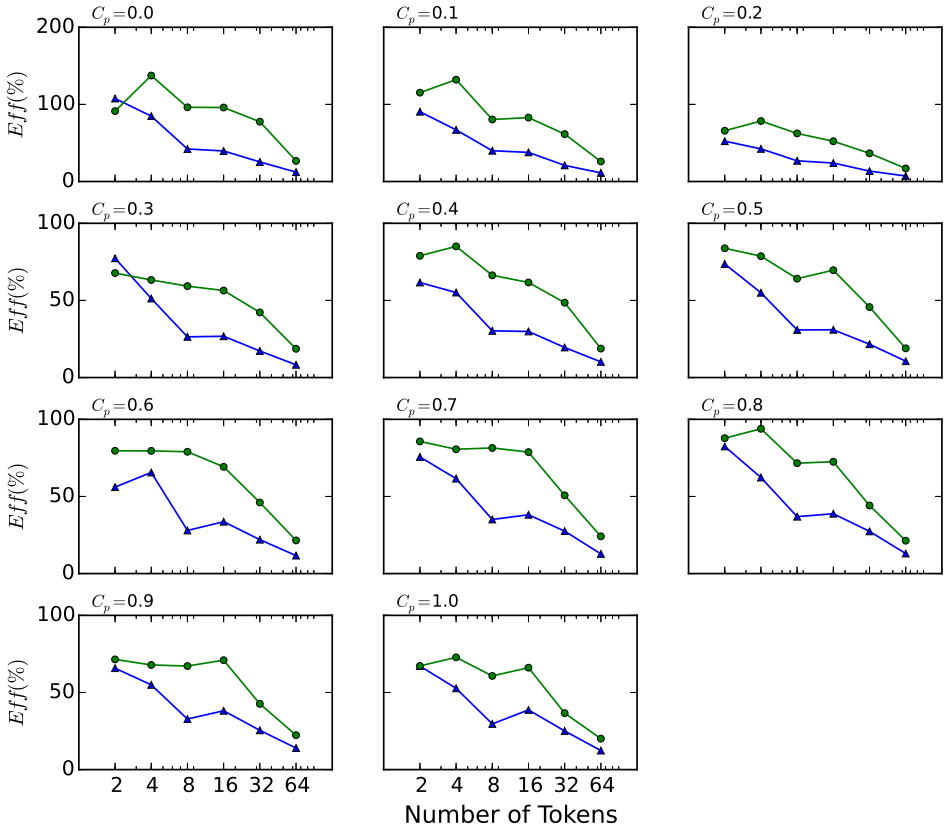



Figure 8.2: Efficiency ($Eff$) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that Tree Parallelization with virtual loss has a lower efficiency meaning lower performance.

of $C_p$. With four tokens (a parallel thread can run each token/task) we see that both methods have similar $SO$ for all values for $C_p$. However, plain Tree Parallelization has smaller $SO$ than Tree Parallelization with the virtual loss on all points, which is opposite to our expectation. The second factor is Time Efficiency (Eff). We hope for the increase of $Eff$ by using virtual loss. Figure 8.2 shows the $Eff$ of each method. We see that plain Tree Parallelization outperforms Tree Parallelization with the virtual loss in almost all tokens for all values of $C_p$, which is opposite to our expectation. The only exception is when the number of tokens is 4 and $C_p$ is 0 and 0.3.

Interestingly, adding virtual loss degrades the performance of lock-free Tree Par-

allelization in the selected problems. This outcome may be due to several factors. We mention two of them. (1) Virtual loss enables parallel threads to search different parts of the shared tree, thus reducing the synchronization overhead caused by using the locks [SKW10]. However, when the algorithm is lock-free, there is no such overhead. (2) Virtual loss disturbs the exploitation-exploration balance of the UCT algorithm.

## 8.6 Answer to the Second Part of RQ5

In this chapter, we addressed part two of *RQ5*.

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

We investigated the virtual loss method (i.e., an artificial increase in exploration) for task-level parallelization of the lock-free Tree Parallelization algorithm (see Section 8.1). Our preliminary results using an application from the High Energy Physic domain shows that when a virtual loss is used for lock-free Tree Parallelization, there is almost no improvement in performance. That is our provisional conclusion of part two of *RQ5*. We showed that (1) the virtual loss method suffered from a high search overhead and that (2) it suffered from a low time efficiency (see Section 8.5).

Originally virtual loss was designed to improve the performance of lock-based Tree Parallelization for the game of Go and not for lock-free Tree Parallelization. If this trend continues, then the new setting (without virtual loss) is (according to our findings) to be preferred. Therefore, we recommend not to use virtual loss along with the task-level parallelization of the lock-free Tree Parallelization algorithm to achieve higher performance.

## 8.7 A Complete answer to RQ5

In Chapter 7 and Chapter 8, we answered *RQ5* in two parts. Here we provide a complete answer for *RQ5*.

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

Chapter 7 provided an answer for part one of *RQ5*. We investigated to what extent a solution (i.e., adjusting the exploitation-exploration balance with respect to the tree size) is for improving the quality of search in Ensemble UCT/Root Parallelization. Previous studies on Ensemble UCT provided inconclusive evidence on the effectiveness

of Ensemble UCT. Our results suggest that the reason for uncertainty (concerning the controversy in the previous studies) lies in the exploitation-exploration trade-off in relation to the size of the sub-trees (or ensemble size). Our results provide clear evidence that the performance of Ensemble UCT will be improved by selecting higher exploitation for smaller search trees given a fixed number of playouts or a fixed search budget.

Chapter 8 presented an answer for part two of *RQ5*. We analyzed a solution (i.e., an artificial increase in exploration called virtual loss) for the lock-free Tree Parallelization algorithm (see Section 8.1). Our preliminary results using an application from the HEP domain showed that when a virtual loss is used for lock-free Tree Parallelization, there is almost no improvement in performance. Moreover, we showed that the virtual loss method suffered from (1) a high search overhead and (2) a low time efficiency (see Section 8.5). As stated in Section 8.6, we recommend not to use virtual loss along with the task-level parallelization of the lock-free Tree Parallelization algorithm to achieve higher performance.

# 9 Conclusions and Future Research

This chapter is built up as follows. Section 9.1 provides a summary of all answers to the five research research questions posed in Chapter 1. Moreover, a definitive answer to the Problem Statement (PS) is formulated in Section 9.2. After that, two limitations concerning the research are discussed in Section 9.3. They are considered as directions along which we will suggest future research. Finally, two additional directions for future research are suggested in Section 9.4.

## 9.1 Answers to the RQs

Below we answer five RQs in the Subsections 9.1.1 to 9.1.5. We start by repeating the RQ, then we provide the answer in brief, and meanwhile references to the relevant sections are given.

### 9.1.1 Answer to RQ1

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

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

### 9.1.2 Answer to RQ2

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

The performance of task-level parallelization to implement the GSCPM algorithm on a multi-core machine with 24 cores was adequate (see Paragraph B of Section 4.9). It reached a speedup of 19, and the FIFO scheduling method showed good scalability for up to 4096 tasks. The performance of task-level parallelization on a many-core co-processor, with the high level of optimization of our sequential code-base, was also good; a speedup of 47 on the 61 cores of the Xeon Phi was reached (see Paragraph C of Section 4.9). Moreover, the FIFO and *task_group* methods showed good scalability for up to 4096 tasks on the Xeon Phi (see Section 4.10). However, our scalability study showed that there is still potential for improving performance and scalability by removing synchronization overhead.

### 9.1.3 Answer to RQ3

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

To answer *RQ3* we have found our way step by step. We did so in three steps. First, we remark that the existing Tree Parallelization algorithm for MCTS uses a shared search tree to run the iterations in parallel (see Subsection 5.1.1). Here we face that the shared search tree has potential race conditions (see Subsection 5.1.2). Our second step is to overcome this obstacle (see Section 5.3). In this section, we have shown that having a correct lock-free data structure is possible. To achieve this goal, we have used methods from modern memory models and atomic operations (see Section 5.3). Using these methods allows removing of synchronization overhead. Hence, we have implemented the new lock-free algorithm that has no race conditions (see Section 5.4). The third step was to evaluate the lock-free algorithm. Therefore we performed an extensive experiment in a small area (Hex on an $11 \times 11$ board), see Sections 5.5 and 5.6. The experiment showed that the lock-free algorithm had a better performance and a better scalability when compared to other synchronization methods (see Section 5.7). The performance of task-level parallelization to implement the lock-free GSCPM algorithm on a multi-core machine with 24 cores was very good. It reached a speedup of 23 and showed very good scalability for up to 4096 tasks. The performance on a many-core co-processor was also very good; a speedup of 83 on the 61 cores of the Xeon Phi was reached. It showed very good scalability for up to 4096 tasks.

### 9.1.4 Answer to RQ4

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

Our research showed that the task-level parallelization method combined with a lock-free data structure for the GSCPM algorithm achieved 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 which 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 (the answer to the first part of *RQ4*) 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 (the answer to the second part of *RQ4*). Different pipeline constructions provided the higher levels of flexibility that allow fine-grained managing of the execution of operations in MCTS (see Subsection 6.6.2).

### 9.1.5 Answer to RQ5

In Chapter 7 and Chapter 8, we answered *RQ5* in two parts. Here we provide a complete answer for *RQ5* which is a summary of both answers.

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

Chapter 7 investigated a solution (i.e., adjusting the exploitation-exploration balance with respect to the tree size) for improving the quality of search in Ensemble UCT or Root Parallelization. Previous studies on Ensemble UCT provided inconclusive evidence on the effectiveness of Ensemble UCT. Our results suggest that the reason for uncertainty (concerning the controversy in the previous studies) lies in the exploitation-exploration trade-off in relation to the size of the sub-trees (or ensemble size). Our results provide clear evidence that the performance of Ensemble UCT is improved by selecting higher exploitation for smaller search trees given a fixed number of playouts or a fixed search budget.

Chapter 8 analyzed a solution (i.e., adjusting the exploitation-exploration balance by an artificial increase in exploration called virtual loss) for the lock-free Tree Parallelization algorithm (see Section 8.1). Our preliminary results using an application

from the HEP domain shows when a virtual loss is used for lock-free Tree Parallelization, there is almost no improvement in performance. We showed that the virtual loss method suffered from a high search overhead and showed a low time efficiency (see Section 8.5). We recommend not to use virtual loss along with the task-level parallelization of the lock-free Tree Parallelization algorithm to achieve higher performance.

## 9.2 Answer to the PS

- ***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 can design a structured parallel programming approach for MCTS in three levels: (1) implementation level, (2) data structure level, and (3) algorithm level.
In the implementation level, we proposed task-level parallelization over thread-level parallelization (see Chapters 3 and 4). Task-level parallelization provides us with efficient parallelism for MCTS to utilize cores on both multi-core and many-core machines.
In the data structure level, we presented a lock-free data structure that guarantees the correctness (see Chapters 5). A lock-free data structure removes the overhead of using data locks when a parallel program needs a lot of tasks to utilized cores.
In the algorithm level, we explained how to use patterns (e.g., pipeline) for parallelization of MCTS to overcome search overhead (see Chapter 6).

Hence the answer to the PS is provided through a step by step approach.

## 9.3 Limitations

There are two limitations in this study, viz. hardware and case studies. We address them below and consider them as topics of future research. In Subsection 9.3.1 we consider the hardware limitations and in Subsection 9.3.2 we briefly discuss the limitations of the case studies.

### 9.3.1 Maximizing Hardware Usage

The first limitation is that the current study used the native mode of the programming paradigm for the execution of the parallel MCTS on the many-core co-processor (i.e., the Xeon Phi). The native mode is the natural first step because it is a fast way to get the existing parallel code running on the Xeon Phi with a minimum of code

changes. However, approaching the co-processor in native mode limits access to only on the Xeon Phi and ignores the resources available on the CPU host or possibly other computing resources. Overcoming this limitation is possible by using offline mode. With offline mode, the parallel program is launched on the CPU side and there data initialization also takes place. The program subsequently pushes (offloads) data and specialized code to the co-processor for executing. After execution, results are pulled back to the CPU. The offload mode allows parallel code to exploit both the CPU and the co-processor. It prepares the application for any foreseeable developments of products.

The future of parallel computing are machines with Systems on Chips (SoC). An SoC is specially designed to incorporate the required electronic circuits of numerous computer components, such as CPU, GPU, or Field-Programmable Gate Array (FPGA), onto a single integrated chip. Therefore, future parallel code for artificial intelligence applications should consider it. For example, an artificial intelligence application based on the MCTS algorithm and deep neural networks has three phases: (1) perception, (2) decision making, and (3) execution. The perception phase can be carried out by a deep neural network. A suitable hardware choice could be a GPU. The decision making phase is handled by MCTS which is running on a CPU. Finally, the execution phase that may need a real-time action can be run on an FPGA co-processor.

### 9.3.2 Using More Case Studies

The second limitation is that the current study used two case studies (i.e., Hex and Horner Scheme). We consider it a limitation for our research, especially for making a definitive conclusion about the performance of the 3PMCTS algorithm. Therefore, a future study should consider using more case studies.

## 9.4 Future Research

Below we give two additional suggestions for future studies.

- From our results, 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, though others are. Our 3PMCTS algorithm gives us full flexibility for offloading a variety of different operations of MCTS to a target processor. Therefore, it is suggested to adapt 3PMCTS for heterogeneous computing.

- For future work, we also suggest exploring other parts of the parameter space, to find optimal $C_p$ settings for different combinations of tree size and ensemble

size. Moreover, we suggest to study the effect in different domains. Even more important will be the study on the effect of $C_p$ in Tree Parallelization.

# Bibliography


[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[AHH10] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[BCC+11] Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssi, Thomas Hérault, Paul Vayssière, and Ziqin Yu. Scalability and Parallelization of Monte-Carlo Tree Search. In *Proceedings of the 7th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 6515, pages 48–58, 2011.

[BG11] Petr Baudiš and Jean-Loup Gailly. Pachi: State of the Art Open Source Go Program. In *Advances in Computer Games 13*, Lecture Notes in Computer Science (LNCS) 7168, pages 24–38, 2011.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Effi-

cient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPOPP '95*, volume 30, pages 207–216. ACM Press, 1995.

[BPW+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. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CJ08] T. Cazenave and N. Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In *Computers and Games*, Lecture Notes in Computer Science (LNCS) 5131, pages 60–71, 2008.

[Cou06] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 4630, pages 72–83, 2006.

[CWvdH08a] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. In *the 6th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 5131, pages 60–71, 2008.

[CWvdH+08b] Guillaume M. J. B. Chaslot, Mark H. M. Winands, H. J. van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[EM10] M. Enzenberger and M. Müller. A Lock-free Multithreaded Monte-Carlo Tree Search algorithm. In *Advances in Computer Games*, Lecture Notes in Computer Science (LNCS) 6048, pages 14–20, 2010.

[EMAS10] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego-An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[FL11] Alan Fern and Paul Lewis. Ensemble Monte-Carlo Planning: An Empirical Study. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 58–65, 2011.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning Series. MIT Press, 2016.

[GI91] Zvi Galil and Giuseppe F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[GS07] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *the 24th International Conference on Machine Learning*, pages 273–280. ACM Press, 2007.

[Hei01] E.A. Heinz. New self-play results in computer chess. In *Computers and Games*, Lecture Notes in Computer Science (LNCS) 2063, pages 262–276, 2001.

[HLL10] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '10*, pages 145–156, 2010.

[HS17] Demis Hassabis and David Silver. Alphago's next move. https://deepmind.com/blog/alphagos-next-move/, 2017.

[HT19] Ryan B Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019.

[Int13] Intel. Intel Xeon Phi Processor Competitive Performance. http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html, 2013.

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science, 2013.

[KPVvdH13] J. Kuipers, A. Plaat, J. A. M. Vermaseren, and H. J. van den Herik. Improving Multivariate Horner Schemes with Monte Carlo Tree Search. *Computer Physics Communications*, 184(11):2391–2395, 2013.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, Lecture Notes in Computer Science (LNCS) 4212, pages 282–293, 2006.

[KUV15] J. Kuipers, T. Ueda, and J. A. M. Vermaseren. Code optimization in FORM. *Computer Physics Communications*, 189(October):1–19, 2015.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Li13] Shou Li. Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework. https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise, 2013.

[LP98] Charles E. Leiserson and Aske Plaat. Programming Parallel Applications in Cilk. *SINEWS: SIAM News*, 31(4):6–7, 1998.

[MKK14] S. Ali Mirsoleimani, Ali Karami, and Farshad Khunjush. A Two-Tier Design Space Exploration Algorithm to Construct a GPU Performance Predictor. In *Architecture of Computing Systems–ARCS 2014*, pages 135–146. Springer, 2014.

[MPvdHV15a] S. Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos 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)*, pages 77–83, 2015.

[MPvdHV15b] S. Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos Vermaseren. Scaling Monte Carlo Tree Search on Intel Xeon Phi. In *The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 666–673, 2015.

[MPVvdH14] S. Ali Mirsoleimani, Aske Plaat, Jos Vermaseren, and Jaap van den Herik. Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi. In *The 2014 European Simulation and Modeling Conference (ESM'2014)*, pages 88–94. Eurosis, 2014.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[OS12] D. O'Shea and R. Seroul. *Programming for Mathematicians*. Universitext. Springer Berlin Heidelberg, 2012.

[Rah13] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[Rei07] J. Reinders. *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[RJ14] James Reinders and James Jeffers. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 4. Elsevier Science, 2014.

[RJM+15] James Reinders, Jim Jeffers, Iosif Meyerov, Alexander Sysoyev, Nikita Astafiev, and Ilya Burylov. *High Performance Parallelism Pearls*. Elsevier, 2015.

[Rob13] Arch D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[Rom01] John W. Romein. *Multigame – An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universiteit, 2001.

[RPBS99] John Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *The 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 725–731, 1999.

[RVPvdH14] Ben Ruijl, Jos Vermaseren, Aske Plaat, and Jaap van den Herik. Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. *Proceedings of ICAART Conference 2014*, 1(1):724–731, 2014.

[RVW+13] A. Ramachandran, J. Vienne, R. V. D. Wijngaart, L. Koesterke, and I. Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *2013 42nd International Conference on Parallel Processing*, pages 736–743, 2013.

[SBDD+02] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Gwendolyn Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[SÇ12] Erik Saule and Umit V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 1629–1639, 2012.

[SCP+14] N. Sephton, P. I. Cowling, E. Powley, D. Whitehouse, and N. H. Slaven. Parallelization of Information Set Monte Carlo Tree Search. In *The 2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2290–2297, 2014.

[Seg11] Richard B. Segal. On the Scalability of Parallel UCT. In *Proceedings of the 7th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 6515, pages 36–47, 2011.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[SKW10] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating Root Parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287, 2010.

[SP14] 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*, 6(3):1–15, 2014.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354, 2017.

[Suk15] Jim Sukha. Brief announcement: A compiler-runtime application binary interface for pipe-while loops. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 83–85. ACM, 2015.

[TD15] Fabien Teytaud and Julien Dehos. On the Tactical and Strategic Behaviour of MCTS When Biasing Random Simulations. *ICCA Journal*, 38(2):67–80, 2015.

[TV10] M. Tentyukov and J. A. M. Vermaseren. The multithreaded version of FORM. *Computer Physics Communications*, 181(8):1419–1427, 2010.

[TV15] Ashkan Tousimojarad and Wim Vanderbauwhede. Steal locally, share globally. *Int. J. Parallel Program.*, 43(5):894–917, 2015.

[vdHPKV13] Jaap van den Herik, Aske Plaat, Jan Kuipers, and Jos Vermaseren. Connecting Sciences. In *In 5th International Conference on Agents and Artificial Intelligence (ICAART)*, volume 1, pages IS–7–IS–16, 2013.

[Ver13] J. A. M. Vermaseren. Hepgame-description of work. https://www.nikhef.nl/ form/maindir/HEPgame/HEPgame.html, 2013.

[Wei17] Eric W. Weisstein. Game of Hex. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/GameofHex.html, 2017.

[Wil12] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning, 2012.

[Woo14] Matthew Woodcraft. Gomill Python Library. http://mjw.woodcraft.me.uk/gomill/, 2014.

[WZS+14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *High-Performance Computing on the Intel® Xeon Phi*. Springer International Publishing, 2014.

[YKK+11] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable Distributed Monte-Carlo Tree Search. In *Fourth Annual Symposium on Combinatorial Search*, pages 180–187, 2011.

# Appendices

# A

# Micro-benchmark Programs

```
double Performance(unsigned int const ITR) {
    unsigned int const SIZE = 16;
    double start_time, duration;
    int i, j;
    __declspec(aligned(64)) double a[SIZE], b[SIZE], c[SIZE];
    for (i = 0; i < SIZE; i++) {
        a[i] = b[i] = c[i] = (double) rand();
    }
#pragma omp parallel for
    for (i = 0; i < ITR; i++) {
#pragma vector aligned (a,b,c)
#pragma unroll(16)
        for (int j = 0; j < SIZE; j++)
        {   a[j] = b[j] * c[j] + a[j]; }
    }
    start_time = elapsedTime();
#pragma omp parallel for
    for (i = 0; i < ITR; i++) {
#pragma vector aligned (a,b,c)
#pragma unroll(16)
        for (int j = 0; j < SIZE; j++)
        {   a[j] = b[j] * c[j] + a[j]; }
    }
    duration = elapsedTime() - start_time;
    double gflop = ((double) 2.0 * SIZE * ITR) / 1e+9;
    double gflops = gflop / duration;
    return gflops;
}
```

Listing A.1: Micro-benchmark code for measuring performance of Xeon Phi.

```
void Bandwidth(unsigned int const ITR) {
    unsigned int const SIZE = 48 * 1000 * 1000;
    double start_time, duration;
    int i, j;
    __declspec(aligned(64)) static double a[SIZE], b[SIZE], c[SIZE];
    for (i = 0; i < SIZE; i++) {
        c[i] = 0.0f;
        a[i] = b[i] = (double) 1.0f;
    }
    for (i = 0; i < 1; i++) {
#pragma omp parallel for
        for (j = 0; j < SIZE; j++)
        {   c[j] = a[j] * b[j] + c[j]; }
    }
    start_time = elapsedTime();
    for (i = 0; i < ITR; i++) {
#pragma omp parallel for
        for (j = 0; j < SIZE; j++)
        {   c[j] = a[j] * b[j] + c[j]; }
    }
    duration = elapsedTime() - start_time;
    double gb = (SIZE * sizeof (double)) / 1e+9;
    double gbs = 4 * ITR * gb / duration;
    return gbs;
}
```

Listing A.2: Micro-benchmark code for measuring memory bandwidth of Xeon Phi.

# B

# Statistical Analysis of Self-play Experiments

Suppose $p$ as true wining probability of a player [Hei01]. The value of $p$ is estimated by $0 \leq w = x/n \leq 1$ which results from $x \leq n$ wins in a match of $n$ games. Therefore, we may simply assume $w$ the sample mean of a binary-valued random variable that counts two draws as a loss plus a win.
The expected value of $w$ is $E(w) = p$ and the variance of $w$ is $Var(w) = p(1-p)/n$. According to central limit theorem approximately, $w \approx Normal(p, p(1-p)n)$, so $(w - p)/\sqrt{p(1-p)/n} \approx Normal(0, 1)$. Let $z_{\%}$ denote the upper critical value of the standard $N(0, 1)$ normal distribution for any desired %-level of statistical confidence($z_{90\%} = 1.645$, $z_{95\%} = 1.96$). Then, the probability of $w - 1.96\sqrt{p(1-p)/n} \leq p \leq w + 1.96\sqrt{p(1-p)/n}$ is about 95%. Therefore, the 95% confidence interval on the true wining probability $p$ is $[w - 1.96\sqrt{p(1-p)/n}, w + 1.96\sqrt{p(1-p)/n}]$. There are two ways to substitute the value of $p$ which is unknown:

1. substitute $p$ for $w$: $[w - 1.96\sqrt{w(1-w)/n}, w + 1.96\sqrt{w(1-w)/n}]$
2. substitute $p$ for $1/2$ which gives wider confidence interval: $[w - 0.98\sqrt{n}, w + 0.98\sqrt{n}]$

# C Implementation of GSCPM

This section will show how the GSCPM algorithm is implemented with three different threading libraries. Furthermore, the implementations for shared search tree and random number generation are explained.

## C.1 TBB

Listing C.1 gives a TBB implementation of GSCPM. TBB has *task_group* class for **fork-join** pattern. Method *run* marks where a fork occurs; method *wait* marks a join.

```
1 tbb::task_group g;
for (int t = 0; t < nTasks; t++) {
3     g.run(UCTSearch(r,m));
}
5 g.wait();
```

Listing C.1: Task parallelism for GSCPM using TBB (*task_group*).

## C.2 Cilk Plus

Two Cilk Plus implementations for GSCPM are given in Listing C.2 and C.3 . Cilk Plus has keywords for marking fork and join points. In the first implementation, the *cilk_spawn* marks the fork and the *cilk_sync* marks an explicitly join operation. The spawning tasks are within a *for* loop. A *cilk_sync* waits for all spawned calls in the loop.

```
for (int t = 0; t < nTasks; t++) {
    cilk_spawn UCTSearch(r,m);
}
cilk_sync;
```

Listing C.2: Task parallelism for GSCPM using Cilk Plus (*cilk_spawn*).

In the second implementation, the *cilk_for* construct uses recursive forking even though it looks like a loop. The *cilk_sync* (joint) at the end of the loop is implicit.

```
cilk_for (int t = 0; t < nTasks; t++) {
    UCTSearch(r,m);
}
```

Listing C.3: Task parallelism for GSCPM using Cilk Plus (*cilk_for*).

## C.3 TPFIFO

In TPFIFO the tasks are put in a queue. It implements work-sharing, but the order that the tasks are executed is similar to *child stealing*. The first task that enters the queue is the first task that gets executed.

In our thread pool implementation (called TPFIFO) the task functions are executed asynchronously. A task is submitted to a FIFO task queue and will be executed as soon as one of the pool's threads is idle. *Schedule* returns immediately and there are no guarantees about when the tasks are executed or how long the processing will take. Therefore, the program waits for all the tasks to be completed.

```
for (int t = 0; t < nTasks; t++) {
    TPFIFO.schedule(UCTSearch(r,m));
}
TPFIFO.wait();
```

Listing C.4: Task parallelism for GSCPM, based on TPFIFO.

# D Implementation of 3PMCTS

In this section, we present the implementation of our 3PMCTS algorithm. In section D.1 we present the concept of *token* (when used as type name, we write *Token*). Section D.2 describes the implementation of 3PMCTS using TBB.

## D.1 Definition of Token Data Type (TDT)

A token represents a path inside the search tree during the search. Algorithm D.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) $\Delta$ represents the reward value of the state. The definition of lock-free data structure *Node* is given in Algorithm 5.1. In Algorithm D.2, the serial UCT algorithm (which is already presented in Algorithm 2.2) is provided using token data type.

**Algorithm D.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 D.2:** The serial UCT algorithm using Token, with stages SELECT, EXPAND, PLAYOUT, and BACKUP.

```
1  Function UCTSEARCH(s_0)
2      v_0 = create root node with state s_0;
3      t_0.s = s_0;
4      t_0.v = v_0;
5      while within search budget do
6          t_l = SELECT(t_0);
7          t_l = EXPAND(t_l);
8          t_l = PLAYOUT(t_l);
9          BACKUP(t_l);
10 Function SELECT(Token t) : <Token>
11     while t.v →IsFullyExpanded() do
12         t.v := argmax_{v' ∈ childrenofv} v'.UCT(C_p);
13         t.s →SetMove(t.v → move);
14     return t;
15 Function EXPAND(Token t) : <Token>
16     if !(t.s →IsTerminal()) then
17         moves := t.s →UntriedMoves();
18         shuffle moves uniformly at random;
19         t.v →Init(moves);
20         v' := t.v →AddChild();
21         if t.v ≠ v' then
22             t.v :=v';
23             t.s →SetMove(v' → move);
24     return t;
25 Function PLAYOUT(Token t)
26     RANDOMSIMULATION(t);
27     EVALUATION(t);
28     return t
29 Function RANDOMSIMULATION(Token t)
30     moves := t.s →UntriedMoves();
31     shuffle moves uniformly at random;
32     while !(t.s →IsTerminal()) do
33         choose new move ∈ moves;
34         t.s →SetMove(move);
35     return t
36 Function EVALUATION(Token t)
37     t.Δ := t.s → Evalute();
38     return t
39 Function BACKUP(Token t) : void
40     while t.v ≠ null do
41         t.v → Update(t.Δ);
42         t.v := t.v → parent;
```

## D.2 TBB Implementation Using TDD

In our implementation for 3PMCTS, each stage (task) performs its operation on a token. We can also specify the number of in-flight tokens.

Each function constitutes a stage of the non-linear pipeline in 3PMCTS. There are two approaches for parallel implementation of a non-linear pipeline [MRR12]:

- *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.

```
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);
        })
    );
...}
```

Listing D.1: An implementation of the 3PMCTS algorithm in TBB.

Our implementation for 3PMCTS algorithm is based on a bind-to-item approach. Figure 6.5 depicts a five-stage pipeline for 3PMCTS that can be implemented using TBB *tbb::parallel_pipeline* template [Rei07]. 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*. Listing D.1 shows the key parts of the TBB code with the syntactic details for the 3PMCTS algorithm.

# Summary

The thesis is part of a bigger project, the HEPGAME (High Energy Physics Game). The main objective for HEPGAME is the utilization of AI solutions, particularly by using MCTS for simplification of HEP calculations. One of the issues is solving mathematical expressions of interest with millions of terms. These calculations can be solved with the FORM program, which is software for symbolic manipulation. Since these calculations are computationally intensive and take a large amount of time, the FORM program was parallelized to solve them in a reasonable amount of time.

Therefore, any new algorithm based on MCTS, should also be parallelized. This requirement was behind the problem statement of the thesis: “How do we design a structured pattern-based parallel programming approach for efficient parallelism of MCTS for both multi-core and manycore shared-memory machines?”.

To answer this question, the thesis approached the MCTS parallelization problem in three levels: (1) implementation level, (2) data structure level, and (3) algorithm level.

In the implementation level, we proposed task-level parallelization over thread-level parallelization. Task-level parallelization provides us with efficient parallelism for MCTS to utilize cores on both multi-core and manycore machines.

In the data structure level, we presented a lock-free data structure that guarantees the correctness. A lock-free data structure (1) removes the synchronization overhead when a parallel program needs many tasks to feed its cores and (2) improves both performance and scalability.

In the algorithm level, we first explained how to use pipeline pattern for parallelization of MCTS to overcome search overhead. Then, through a step by step approach, we were able to propose and detail the structured parallel programming approach for Monte Carlo Tree Search.

# Samenvatting

Het proefschrift maakt deel uit van een groter project, het HEPGAME (High Energy Physics Game) project. Het hoofddoel van HEPGAME is het gebruik van AI-oplossingen, met name door MCTS te gebruiken voor de vereenvoudiging van HEP-berekeningen. Een van de problemen is het oplossen van relevant wiskundige expressies met miljoenen termen. Deze berekeningen kunnen worden verricht met het FORM-programma, dat is een specifiek software-pakket voor symbolische manipulatie. Omdat de berekeningen rekenintensief zijn en daardoor veel tijd kosten, is het FORM-programma parallel uitgevoerd om berekeningen binnen een redelijke tijd te executeren. Daarom moet elk nieuw algoritme op basis van MCTS ook parallel kunnen worden uitgevoerd. Deze eis ligt direct onder de probleemstelling van het proefschrift: “Hoe ontwerpen we een gestructureerde, op patronen gebaseerde parallelle programma-aanpak voor efficiënt parallellisme van MCTS voor zowel *multi-core* als *manycore* machines met een gedeeld geheugen?”. Om deze vraag te beantwoorden, benadert het proefschrift de MCTS de parallellliserings problemen op drie niveaus: (1) implementatieniveau, (2) datastructuurniveau, en (3) algoritmeniveau. Op het implementatieniveau hebben we parallellisatie op taakniveau verkozen boven parallellisatie op *thread*niveau. Parallellisatie op taakniveau biedt ons efficiënte parallelliteit voor MCTS om kernen (*cores*) te gebruiken op zowel *multi-core* als *manycore* machines. Op het niveau van de datastructuur hebben we een *lock-free* datastructuur voorgesteld die de korrektheid garandeert. Een *lock-free* gegevensstructuur (1) verwijdert de synchronisatie-overhead wanneer een parallel programma veel taken nodig heeft om cores te gebruiken en (2) verbetert zowel de prestaties als de schaalbaarheid. Op het algoritmeniveau hebben we eerst uitgelegd hoe een pijplijnpatroon moet worden gebruikt voor parallellisatie van MCTS om de search overhead te overwinnen. Daarna konden we via een stapsgewijze aanpak de gestructureerde parallelle programmering voor Monte Carlo Tree Search gestalte geven.

# Acknowledgment

First and foremost, I would like to express my sincere appreciation to my first advisor Professor Jaap van den Herik for the endless support of my Ph.D. study and the research involved, for his patience, motivation, enthusiasm, and extensive knowledge. His supervision helped me in all times of research and writing of this thesis.

Besides my first advisor, I would like to thank my other advisors Professor Aske Plaat and Dr. Jos Vermaseren, for their encouragement, insightful comments, and endless support.

Besides my advisors, I would like to gratefully recognize the members of my thesis committee: Professor P. J. G. Mulders, Professor F. J. Verbeek, Professor H. A. G. Wijshoff, Dr. F. Khunjush, Dr. W. A. Kosters, and Dr. A. L. Varbanescu, for their time to read the thesis.

Moreover, I acknowledge my fellow labmates at Leiden University: Ben Ruijl, Bilal Karasneh, Hafeez Osman. Also, I pay tribute to my friends at Leiden University: Sobhan Niknam and Ramin Etemadi.

Furthermore, my sincere thanks also go to Joke Hellemons for offering tremendous help during my first three years of study. Notably, the first year, by helping me to settle in Tilburg.

Then, my special thanks also go to Letty Raaphorst for her warm acceptance of me at their home during the last phase of writing the thesis.

I want to thank my family, especially my parents Mehdi Mirsoleimani and Masoomeh Zarinkolah, for supporting me throughout my entire life. My thanks also go to my parents-in-law. Here, I also want to thank specifically my grandmother Keyhan Masoudi for encouraging me in pivotal moments of life.

Last but not least, I would like to thank my dear wife Elahe, for her endless support, especially. Without her love, I would not have been able to overcome the challenges during this period.

# Curriculum Vitae

Sayyed Ali Mirsoleimani was born in Abadeh, Iran, on May 28, 1986. He obtained his bachelor degree in Computer Software Engineering at the Islamic Azad University in Najafabad, Iran. In 2013, he continued with a Master in Software Engineering at the Shiraz University in Shiraz, Iran. His Master's thesis was titled "Proposing and Evaluation of a Performance Prediction Model for Graphics Processing Units".

Immediately thereafter, Ali started to work in an ERC Advanced Grant funded Ph.D. project at the Tilburg University under supervision of Professor Jaap van den Herik and Professor Aske Plaat and at Nikhef in Amsterdam under supervision of Dr. Jos Vermaseren (the PI of the project). In April 2014, he accompanied his supervisors to the Leiden University. He defended his Ph.D thesis in 2020.

Currently, Ali is working at ASML, the world's leading manufacturer of lithography machines, in Eindhoven. Ali is mainly responsible for developing calibration models to contribute to realizing Moore's Law.

# Publications

- S. A. Mirsoleimani, A. Plaat, J. Vermaseren, and H. J. van den Herik, Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi, in Proceedings of the 2014 European Simulation and Modeling Conference (ESM 2014), 2014, pp. 88--94.
- S. A. Mirsoleimani, A. Plaat, H. J. van den Herik, and J. Vermaseren, Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors, in Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, vol. 3, pp. 77--83.
- S. A. Mirsoleimani, A. Plaat, H. J. van den Herik, and J. Vermaseren, Scaling Monte Carlo Tree Search on Intel Xeon Phi, in Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2015, pp. 666--673.
- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, Ensemble UCT Needs High Exploitation, in Proceedings of the 8th International Conference on Agents and Artificial Intelligence, 2016, pp. 370--376.
- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, An Analysis of Virtual Loss in Parallel MCTS, in Proceedings of the 9th International Conference on Agents and Artificial Intelligence, 2017, pp. 648--652.
- S. A. Mirsoleimani, H. J. van den Herik, A. Plaat and J. Vermaseren, A Lock-free Algorithm for Parallel MCTS, in Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2, 2018, pp. 589--598.
- 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.

# SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
09 Tim de Jong (OU), Contextualised Mobile Media for Learning
10 Bart Bogaert (UvT), Cloud Content Contention
11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets

15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization

39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems

13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications

10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications

32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information

15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital

40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork

17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training

08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations

33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly

08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity

34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
02 Felix Mannhardt (TUE), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology

11 Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TUE), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TUE), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The ”K” in ”semantic web” stands for ”knowledge”: scaling semantics to the web

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms

08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU), Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics

32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming
36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills
37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OUN), Learning visually grounded and multilingual representations

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TUE), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OUN), Towards a software architecture for reusable game components
08 Sayyed Ali Mirsoleimani (UL), Structured Parallel Programming for Monte Carlo Tree Search