Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

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 11×11 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.