A Study of Software Framework for Parallel Monte Carlo Tree Search

Ting-Fu Liao[†1] I-Chen Wu[†1] Guan-Wun Chen[†1] Chung-Chin Shih[†1]
Po-Ya Kang[†1] Bing-Tsung Chiang[†1] Ting-Chu Ho[†1] Ti-Rong Wu[†1]

Abstract: Monte-Carlo tree search (MCTS) has been successful on improving the strength of the game Go as well as many other game playing programs. For MCTS, one of the critical issues to further improve strength is parallelization. In order to deal with parallel MCTS generally, this paper designs a software framework for developing computer game programs with parallel MCTS. This framework hides the details of game-independent designs from computer game developers, so that developers can concentrate on improving heuristics related to game-specific knowledge. In this framework, we used lock-free tree parallelization inside a shared-memory system, and root parallelization over a distributed-memory system. For demonstration, we implemented a Go program named AMIGO, a Chinese dark chess program, and a puzzle program on top of this framework. The experimental results of AMIGO for 9x9 Go also show reasonable speedups and the strength improvement.

Keywords: Monte-Carlo tree search, parallelization, Go, Chinese dark chess, puzzle.

1. Introduction

The strengths of Computer Go programs were greatly improved in the past decade. *Monte-Carlo tree search* (*MCTS*), an innovative search method, plays a key role of the improvement [1)]. In addition to its success in Computer Go, the search method has also been successfully applied to other board games [2)-4)], such as NoGo, Hex, Havannah, Chinese dark chess, and even other games or applications, such as Pacman [5)], general game playing [6)], flexible job shop scheduling problem (FJSP) [7)].

In order to improve strength of MCTS, some researchers proposed to do parallelization for MCTS [8),9)]. Chaslot et al [10)] classified it into three kinds of parallelism for MCTS, *leaf parallelization*, *root parallelization* and *tree parallelization*. Leaf parallelization is disregarded in this paper, since it was clearly outperformed by the other two according to [10)].

Root parallelization incurs very little synchronization overhead for the following reason. All threads are performed independently, and the winrates of children of the root are the summations of them from all threads. Surprisingly, as the research done by Yusuke Siejima [11)], root parallelization performs reasonably well. Thus, it is also suitable for threads over a distributed system, not just in a shared-memory system.

In [10)], the article shows that tree parallelization incurs significant synchronization overhead by using mutexes. If a global mutex is used to lock the whole tree, the overhead for thread synchronization is too high; and if local mutexes are used to lock individual tree nodes, the overhead is much lower. For greatly reducing the overhead, Muller devised lock-free tree parallelization when implementing Fuego [12)]. From above, a reasonable compromise is to implement lock-free tree parallelization inside a shared-memory system, while implementing root parallelization over a distributed-memory system.

Unfortunately, it is still nontrivial and time-consuming to design/implement lock-free tree parallelization in MCTS which includes threading control and memory management, as well as distributed communication mechanism for root parallelization. In the past, for alpha-beta search, several general game systems were designed and implemented to facilitate the development of game-playing programs for developers efficiently [13),14)]. In order to deal with parallel MCTS generally, this paper designs a software framework for developing computer game programs with parallel MCTS. This framework hides the details of game-independent designs from computer game developers, so that developers can concentrate on improving heuristics related to game-specific knowledge. In this framework, we used lock-free tree parallelization inside a shared-memory system, and root parallelization over a distributed-memory system.

The framework is described in Section 2. For demonstration of this framework, we designed and implemented a Go program, named AMIGO, as well as a Chinese dark chess program and a puzzle program in Section 3. The experimental results of AMIGO for 9x9 Go also showed reasonable speedups and the strength improvement. The concluding remarks are given in Section 4.

2. Framework

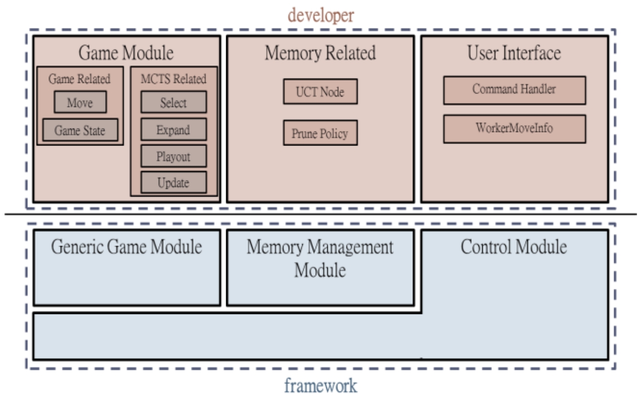

Fig. 1. Software Framework.

The architecture of the software framework is shown in Fig. 1. Software Framework. The framework mainly supports three modules, generic game module, memory management module, and control module. For threading in a host or machine, this framework uses a master thread in the control module to manage slave threads, each of which runs the MCTS algorithm

[†1] Department of Computer Science, National Chiao Tung University, Taiwan.

depending on the game module implemented by developers.

In principle, this framework hides the details of game-independent designs from computer game developers, so that developers can concentrate on improving heuristics related to game-specific knowledge. So, the computer game developers only need to develop the corresponding modules on top of them.

The three modules, generic game module, memory management module, and control module are described in Subsections 2.1, 2.2 and 2.3, respectively. The distributed system for parallelism is discussed in Subsection 2.4.

2.1 Game Module

In the game module, developers only need to implement game-related objects and MCTS-related operations.

The game-related objects include moves and game states, whose classes are respectively named `Move` and `GameState`. The former represents an action in the game, while the latter represents a game state. For the latter, developers also need to implement the following four operations: reset to the initial state, rollback to the root, backup for future rollback, and play a move.

The MCTS-related operations include those in the four common MCTS phases, selection, expansion, playout and update. The operation for playout is encapsulated in a class, named `UctAccessor`, while the other three are encapsulated in a class, named `PlayoutAgent`.

2.2 Memory Management Module

In the memory management module, developers only need to implement the UCT node and the pruning policy, described in Subsection 2.2.1 and 2.2.2, respectively. The details of node management and tree pruning mechanism are hidden from developers.

2.2.1 UCT Node

The data structure of UCT node only includes the basic part for running MCTS algorithm. Developers need to extend their own data fields additionally.

In the phase of expansion, a node allocation request is invoked to allocate nodes. From [12)], node allocation can be done in a lock-free manner, as long as the number of expanded children is deterministic. In this framework, node allocation is also based on the lock-free method. The details of node allocation mechanism are omitted in this paper.

2.2.2 Tree Pruning

When the size of allocated nodes is sufficiently large to some extent or nearly full in the memory system, it becomes necessary for the system to reclaim more space for more node allocation. Our method is to prune the subtrees that contains less interesting or significant nodes. Developers are allowed to define their own pruning policies in a routine, named `PrunePolicy`. In general, the policy is to determine whether to prune the subtrees rooted at given nodes. Then, the system is responsible for all the details of saving the needed nodes and pruning unnecessary nodes.

2.3 Control module

In control module, the system provides developers with an application programming interface to implement user interfaces and control the computation of each thread.

As described above, the framework uses one master thread as well as slave threads in a host (machine). Master threads are in charge of the work such as communicating with users, pruning tree and controlling slave threads, each of which runs MCTS algorithm.

Following are the four major interactions between master threads and slave threads, initializing, move generating, tree pruning and background thinking.

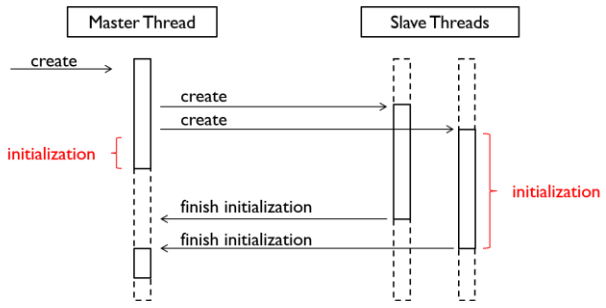

Fig. 2. Initialization

The initialization of each thread is shown in Fig. 2, the master thread will create the slave threads and initialize itself immediately after all slave threads are created.

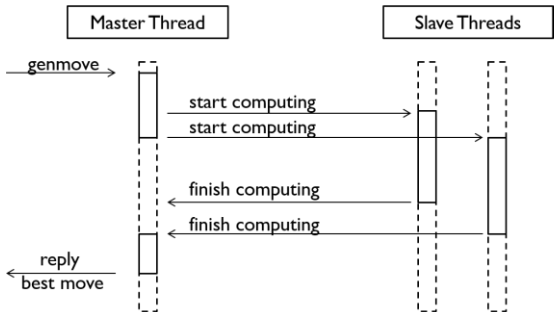

Fig. 3. Move generation.

The interaction for generating a move is shown in Fig. 3. The master thread asks all the slave threads to compute the moves and chooses the best one as the answer.

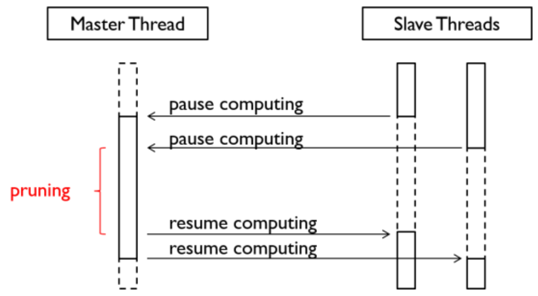

Fig. 4. Tree pruning

The interaction for tree pruning is shown in Fig. 4. Once it is out of pages, slave threads will stop the computation and wait for the master thread finishing the tree pruning.

Fig. 5 (below) shows the interaction between the master thread and the slave threads during background thinking. After asking the slave threads to think in background, the master thread keeps interacting with the user until the game state is changed.

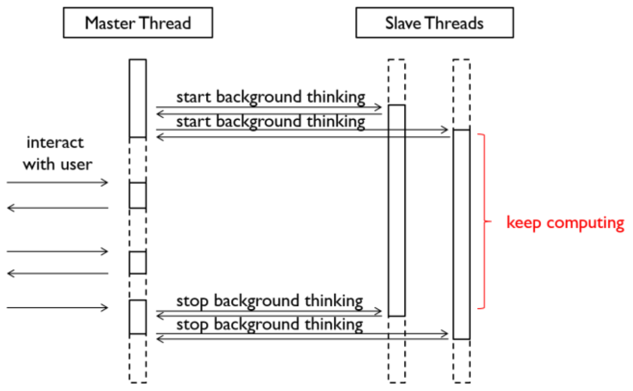

Fig. 5. Background thinking.

2.4 Distributed System Parallelization

Our framework supports to run in a distributed system. As shown in Fig. 6 (below), the architecture of the system consists of one server and multiple workers. Each worker runs MCTS algorithm in the manner of background thinking as described above. The server collects information from workers and interacts with users.

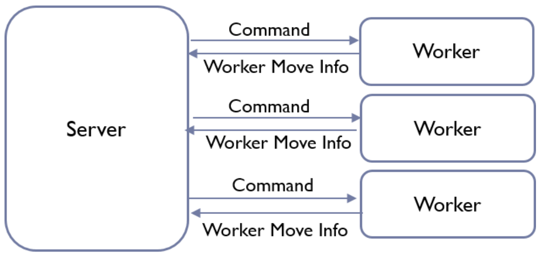

Fig. 6. Architecture of distributed system.

The server sends commands to workers and handles the actions of receiving information from workers. The information is encapsulated in a class named `WorkerMoveInfo` which is serialized when sent by workers. The server parses (or deserialize) `WorkerMoveInfo` and maintains the latest report in server side to determine which is the best move to play.

2.4.1 Server

In the distributed system, the server is in charge of two jobs only, communicating with users and controlling workers. In order to communicate with users smoothly, developers need to implement the `resolve` function, which is used to interpret the command sent by users.

In order to control workers, developers should implement a function named `dispatch`, which is able to send the correct commands to workers according to the information provided by users or the report maintained by servers.

2.4.2 Worker

The workers keep thinking in the background unless they receive commands from the server. Once they receive commands, they will suspend background thinking and do the actions whichever the server required. After reacting the commands, the workers send a message to the server by using `WorkerMoveInfo`.

Several examples for the interaction between server and worker including board cleaning, move playing and state notifying are shown as follow.

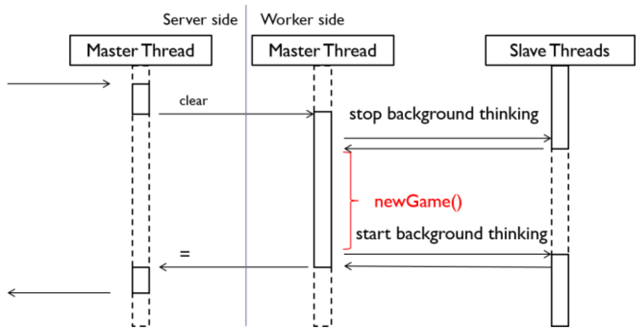

Fig. 7. Cleaning the game board.

Fig. 7 shows the interaction for cleaning the game board. Once the user starts a new game, the server will send messages to the workers. After receiving the messages, the workers stop background thinking and renew their game boards.

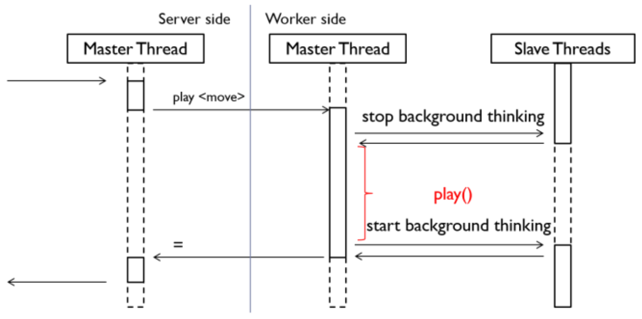

Fig. 8. Making a move

Fig. 8 shows the interaction of making a move. When the user or the server plays a new move, the server will send a command called `play` to the workers. The workers stop background thinking and change their game boards after receiving the command.

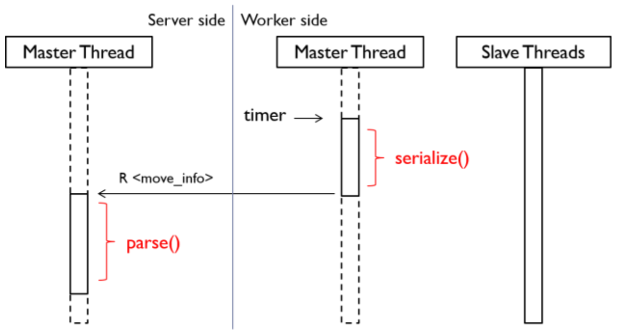

Fig. 9. State notification.

Fig. 9 shows how workers feedback their computing states to the server. The workers transform their game states into the `WorkerMoveInfo` format automatically every T_R second. The server will keep the information for choosing the best move to play.

3. Experiments

To demonstrate the framework, we implemented a Go program, named AMIGO[†2], a Chinese dark chess program, a puzzle program, and a Tic-Tac-Toe program. Table 1 shows the code size of the framework as well as additional code sizes for

[†2] The program was renamed CGI later due to name conflict with an old Go.

implementing these game-playing programs. For example, developers only need about 300 lines to implement a basic MCTS algorithm for Tic-Tac-Toe. These game-playing programs (except Tic-Tac-Toe) are described in the subsequent subsections, respectively.

	Parallel MCTS Framework	Go	Dark Chess	Puzzle	Tic-Tac-Toe
Code length	5559 lines	~2500 lines	~1500 lines	~500 lines	~300 lines

Table 1. Number of lines implemented by developer.

3.1 Go

The computing environment was experimented on a supercomputer, named ALPS, at the National Center for High-performance Computing (NCHC), Taiwan. Each machine (also called node) in ALPS is equipped with AMD Opteron Processor 6174 with 48 cores and 128G memory in total.

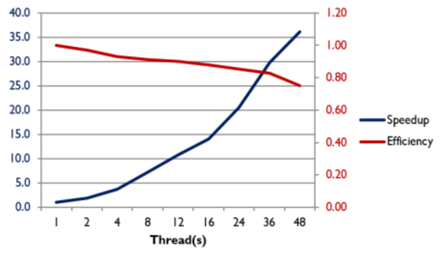

Fig. 10. Speedups and efficiencies for using different numbers of threads.

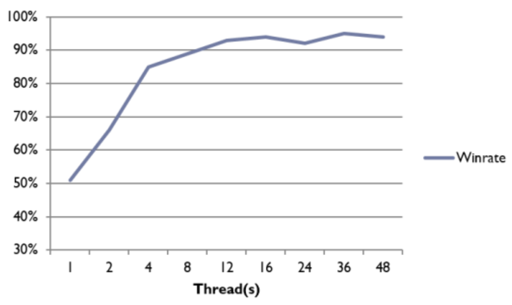

Fig. 11. Winrates for different numbers of threads.

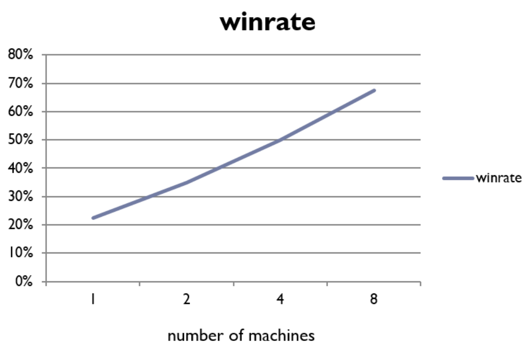

Fig. 12. Winrates against Fuego.

In our first experiment, the program for 9x9 Go ran for 10 seconds using different numbers of threads inside one node in ALPS. Fig. 10 shows the speedups and efficiencies by running in 10 seconds. Fig. 11 demonstrates the strength improvement of the program by showing the winrates against the one with one thread. In Fig. 11, all programs runs in one second for each move. The results show that in shared-memory parallelization the speedup is 36.08 for 48 threads, and the winrate against the single-core version is more than 90% when using 12 threads or more.

Fig. 12 shows the strength improvement of the program by using different numbers of ALPS nodes (machines) against a single-thread Fuego. All programs runs in five seconds for each move. The result shows that in distributed-memory parallelization the winning rate for one machine with 36 cores against Fuego is 23%, while that for 8 machines each with 36 cores is 68%, about 45% higher.

AMIGO also attended 9x9 Go tournament in the 17th Computer Olympiad, held in Yokohama, Japan, 2013. In this tournament, AMIGO with a total of 576 cores won the 4th place. The above result demonstrates that our framework keeps good quality while supporting general game-playing programs.

3.2 Chinese Dark Chess

Chinese dark chess is a two-player imperfect information game. Nondeterministic Mote Carlo tree search algorithm was proposed by Yen et al. [15)]. To demonstrate our framework for the game, we used a two-layer structure to represent the nondeterministic state nodes mentioned in the paper.

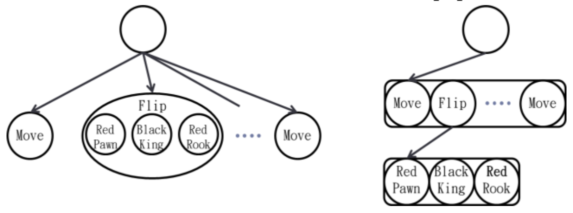

Fig. 13. Transformation between nondeterministic node and two-layer structure.

Fig. 14 shows the speedups for different numbers of threads. The curve in Fig. 14 is almost linearly when the core number goes higher. The result shows that the program based on this framework works smoothly for Chinese dark chess.

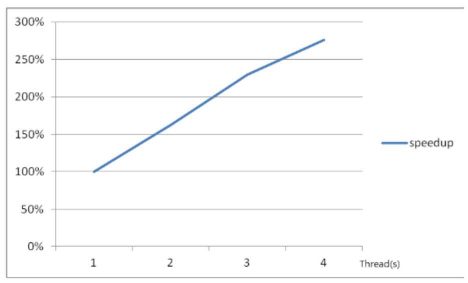

Fig. 14. Speedups among different numbers of threads.

3.3 15-Puzzle

15-Puzzle [17)] is a well-known single-player game. We chose this game as a test case of single-player game for our framework. Different with two-player games, single-player games do not have the concept of winning, which is necessary for winrate computation. Therefore, we computed the winrate of each puzzle based on their Mahanttan distance. Namely, the Manhattan distance was transformed into winrates as shown in

Fig. 15, similar to that in the research done in [7)].

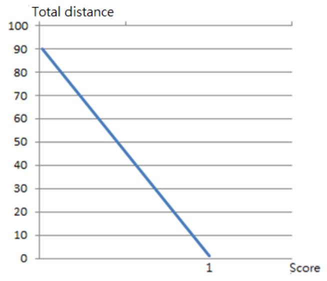

Fig. 15. Transformation between Manhattan distance and winrate.

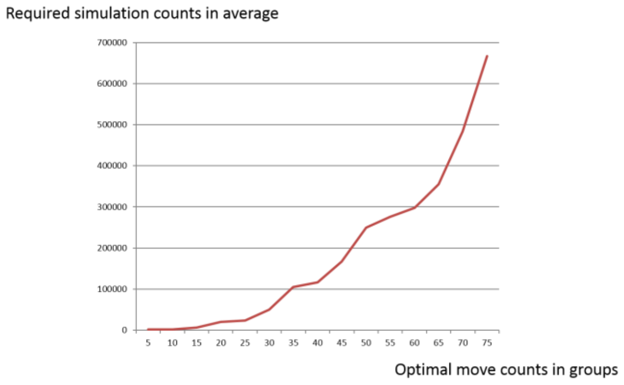

Fig. 16. Required simulation count and moves of best solution.

The problem of 15-puzzle is to achieve the goal state with the optimal solution which requires the minimum number of moves. In this experiment, we used MCTS to find the solutions with the number of moves as small as possible. In order to measure how good the MCTS program is, we obtained the number of simulations in which the program can reach the goal state with the optimal solution. In this experiment, we collected 150 15-puzzles all of which can be solved within 80 steps. Then, these puzzles were separated into 15 groups depending on the move counts of their optimal solutions. For example, the puzzles with 55-60 moves in the optimal solution were grouped into 60. Fig. 16 shows the relation between simulation counts (in average) and puzzle groups. This experiment only tested the feasibility of developing single-player game programs on MCTS, but not for performance.

4. Conclusion

This paper designs a software framework for developing computer game programs that are based on parallel *Monte-Carlo tree search* (*MCTS*). This framework hides the details of game-independent designs from computer game developers, so that developers can concentrate on improving heuristics related to game-specific knowledge. In this framework, we used lock-free tree parallelization inside a shared-memory system, and root parallelization over a distributed-memory system.

To demonstrate the framework, we implemented a Go program, named AMIGO, a Chinese dark chess program, and a puzzle program. The experimental results of AMIGO for 9x9 Go also demonstrated reasonable speedups and the strength improvement. It is expected that more applications can be designed based on the framework.

References

1) Coulom, R., Monte Carlo tree search in crazy stone. *Game Programming Workshop*, pp. 74–75, Tokyo, Japan, 2007.
2) Arneson, B., Hayward, R. B., and Henderson, P., Monte Carlo Tree Search in Hex, *IEEE Transactions on Computational Intelligence and AI in Game*, Vol. 2, No. 4, December 2010.
3) Lorentz , R. J., Improving Monte-Carlo Tree Search in Havannah, *Computers and Games Lecture Notes in Computer Science,* Vol. 6515, pp. 105-115, 2011.
4) Yen, S.-J., Chou, C.-W., Chen, J.-C., Wu, I-C., Kao, K.-Y., The Art of the Chinese Dark Chess Program DIABLE, *Proceedings of the International Computer Symposium ICS, 2012*.
5) Ikehata N. and Ito T., Monte-Carlo Tree Search in Ms. Pac-Man, *IEEE Conference on Computational Intelligence and Games (CIG),* 2011.
6) Finnsson H. and Bjornsson Y., Simulation-Based Approach to General Game Playing, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008.
7) Tung-Ying Wu, I-Chen Wu, Chao-Chin Liang, Multi-Objective Flexible Job Shop Scheduling Problem Based on Monte-Carlo Tree Search, the *2013 Conference on Technologies and Applications of Artificial Intelligence* (*TAAI 2013*), Taipei, Taiwan, December 2013.
8) Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, Yutaka Ishikawa. Scalable Distributed Monte Carlo Tree Search. Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, 2011.
9) Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers.Parallel Monte-Carlo Tree Search for HPC Systems.In Proceedings of the 17th International Conference Euro-Par, 2011.
10) Chaslot G.M.J.B., Winands M.H.M., and H.J. van den Herik. Parallel monte-carlo tree search. *Proceedings of the Conference on Computers and Games* 2008.
11) Yusuke Soejima, Akihiro Kishimoto and Osamu Watanabe. Evaluating Root Parallization in Go, IEEE Transactions on Computational Intelligence and AI in Games, Volume 2, Number 4, pages 278-287, 2010.
12) Enzenberger M., Muller M., Fuego - An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search, *IEEE Transactions Computational Intelligence and AI in Games*, 2009.
13) Barney Pell. Strategy Generation and Evaluation for Meta-Game Playing, PhD Thesis, University of Cambridge, 1993.
14) John W. Romein. Multigame - An Environment for Distributed Game-Tree Search, Vrije Universiteit Amsterdam, 2001.
15) Yen, S.-J., Chou, C.-W., Chen, J.-C., Wu, I-C., Kao, K.-Y., The Art of the Chinese Dark Chess Program DIABLE, Proceedings of the International Computer Symposium ICS, 2012.
16) Zillions of Games, http://www.zillions-of-games.com/
17) R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27:97–109, 1985.

Acknowledgments The authors would like to thank the Ministry of Science and Technology of the Republic of China (Taiwan) for financial support of this research under the contract numbers NSC-102-2221-E-009-069-MY2 and NSC-102-2221-E-009-080-MY2, and the National Center for High-performance Computing (NCHC) for computer time and facilities.