Scalability and Parallelization of Monte-Carlo Tree Search

Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean*, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault**, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssière, and Ziqin Yu

TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud), France
*LIG, UMR 5217, Université de Grenoble, Grenoble, France
**LRI, UMR 8623 (CNRS - Univ. Paris-Sud), Inria; University of Tennessee

Abstract. Monte-Carlo Tree Search is now a well established algorithm, in games and beyond. We analyze its scalability, and in particular its limitations, and the implications in terms of parallelization, in particular for our program MoGo but also for our Havannah program Shakti. In particular, we get a good efficiency for the parallel versions, both for multicore machines and for message-passing machines, but in spite of promising results in self-play there are situations for which increasing the time per move does not solve anything, and therefore parallelization is not the solution either. Nonetheless, for problems on which the Monte-Carlo part is less biased than in Go, parallelization should be very efficient even without shared memory.

1 Introduction

Since 2006, Monte-Carlo Tree Search (MCTS[6, 9, 17]) is a revolution in games and planning, with applications in many fields. After big successes in Go, MCTS was applied to other games as well (Havannah[22],Amazons[19],General Game Playing[21]). It was also applied to optimization on graphs ([11] with an application to library performance tuning), and to fundamental artificial intelligence tasks like non-linear optimization and active learning[2, 20]. It has in particular the advantage that it does not require an evaluation function (whereas alpha-beta does) and is therefore suitable for problems in which no good evaluation function is known - one can consider that a random simulation, with the reward at the end, is a form of evaluation function, but one can build such tools in any game for which rules are explicitly provided.

It is also widely said that MCTS has some scalability advantages.

It is quite natural, then, to parallelize MCTS, both on multicore machines [23] and on clusters [13, 5]. In this paper, after an introduction to MCTS we (i) discuss the scalability of MCTS, showing big limitations to this scalability, and not only due to Rave (section 2); (ii) compare existing algorithms on clusters (section 3).

Monte-Carlo Tree Search

We below introduce Monte-Carlo Tree Search, i.e. MCTS. We here present the MCTS variant termed UCT [17], which is shorter to present and very general; the formulas involved in our programs are more tricky and can be found in [14, 18, 13, 22]; these details do not affect the parallelization.

UCT is presented in Algorithm 1. The reader is referred to [17] for a more detailed presentation, and to [14, 23, 9, 7] for a more comprehensive introduction in particular for the specific case of binary rewards and two-player games.

2 Scalability of MCTS

The scalability of MCTS, i.e. its ability to play better when additional computational power or time is provided[1], is often given as an argument in favor of it. Also, it is said that the parallelization is very efficient; the conclusion of these two statements is that with big clusters, programs should now be much stronger than humans in games in which single computers are already at the level of beginners. We will here give more information (limitations) on this scalability.

The number of simulations per move is usually much larger in real games than in experimental results published in papers, because of limited computational power - it's difficult, even with a cluster, to have significant results corresponding to the computational power associated to realistic time settings on a big machine. In this section, we investigate the behavior of MCTS when the time per move is increased (section 2.1), followed by counter-examples to scalability (section 2.2).

2.1 The limited scalability by numbers

It is usually said that MCTS is highly scalable, and provides improvements of constant order against the baseline when the computational power is doubled. We here show that things are not so constant; results are presented in Table 1 for the game of Go. These numbers show the clear decrease of scalability as the computational power increases. This is not specific to Go; Table 2 shows that the situation is similar in Havannah. This holds even when the opponent is a MCTS also; this is not equivalent to the case of the scalability study `http://cgos.boardspace.net/study/index.html` which considers non-MCTS opponents as well; we here see that just against the same MCTS program, we have a limit in scalability; this even happens in 19x19. In Havannah with slow simulations (the operational case, with the best performance in practice), 10 000 simulations per move give only 52% winning rate against 5 000 simulations per move (Table 2). This suggests that the scalability is smaller than expected

[1] This definition of scalability is often used in games; this is different from the definition used in parallelism, where it considers only the extent to which the sequential behavior can be recovered (faster) by the parallel algorithm. As we will see, the limited efficiency also occurs in the sequential case - the trouble is not due to communication costs or overheads of the parallel implementation.

Algorithm 1 Overview of the UCT algorithm for two-player deterministic games. The adaptation to stochastic cases or one-player games is straightforward. UCT takes as input a situation $s \in \mathcal{S}$, and outputs a decision. For any situation s and any decision d, $s' = s.d$ denotes the situation s' subsequent to decision d in situation s. T is made of two mappings (initially identically 0), N_T and S_T: N_T is a mapping from $\mathcal{S}$ to $\mathbb{N}$ (i.e. maps situations to integers) and S_T is a mapping from $\mathcal{S}$ to $\mathbb{R}$. $\mathcal{S}$ is the set of states, S_T stands for the sum of rewards at a given state and N_T stands for the number of visits at a given state.

Function $UCT(s)$
$T \leftarrow 0$
while TimeLeft> 0 **do**
  $PerformSimulation(T, s)$
end while
Return r maximizing $N_T(s.r)$
Function $reward = PerformSimulation(T, s)$
if s is a final state **then**
  return the reward of s
else
  if $N_T(s) > 0$ **then**
    Choose the move to be simulated as follows:
    if Color(s)=myColor **then**
      $\epsilon = 1$
    else
      $\epsilon = -1$
    end if
    $d = \arg\max_d Score(\epsilon.S_T(s.d), N_T(s.d), N_T(s))$
  else
    $d = MC(d)$ /* $MC(d)$ is a heuristic choice of move */
  end if
end if
$reward = PerformSimulation(T, s.d)$ // reward$\in \{0, 1\}$
Update the statistics in the tree as follows:
$N_T(s) \leftarrow N_T(s) + 1$
$S_T(s) \leftarrow S_T(s) + reward$
Return $reward$
Function $Score(a, b, c)$
Return $a/b + \sqrt{2\log(c)/b}$ /* plenty of improvements of this Eq. are published in the literature for specific problems*/

N =Number of simulations	Success rate of 2N simulations against N simulations in 9x9 Go	Success rate of 2N simulations against N simulations in 19x19 Go
1 000	71.1 ± 0.1 %	90.5 ± 0.3 %
4 000	68.7 ± 0.2	84.5 ± 0.3 %
16 000	66.5 ± 0.9 %	80.2 ± 0.4 %
256 000	61.0 ± 0.2 %	58.5 ± 1.7 %

Table 1. Scalability of MCTS for the game of Go. These results show a decrease of scalability as computational power increases.

Number of fast simulations	Success rate
100 vs 50	68.6 ± 0.68%
1000 vs 500	63.57 ± 0.76%
2000 vs 1000	59.0 ± 1.0%
4000 vs 2000	53.9 ± 1.6%
10000 vs 5000	55.2 ± 1.6%
20000 vs 10000	54.89 ± 1.25%

Number of slow simulations	Success rate
100 vs 50	63.28 ± 0.4%
1000 vs 500	57.37 ± 0.9%
2000 vs 1000	56.42 ± 1.1%
4000 vs 2000	53.24 ± 1.42%
10000 vs 5000	52 ± 1.6%

Table 2. Scaling for the game of Havannah, for fast (left) and slow (right) simulations. As we can see, the success rate is not constant; the numbers have somehow big standard deviations, but we nonetheless see that the success rate of $2N$ simulations versus N simulations decreases when N increases.

from small scale experiments. Usually people do not publish experiments with so many simulations because it is quite expensive; nonetheless, real games are played with more than this kind of numbers of simulations and the numbers in the tables above are probably greater than the scalability in realistic scenarios.

A particularity of these numbers is that they are in self-play; this provides a limitation even in the ideal case in which we only consider an opponent of the same type; it is widely known that the improvement is much smaller when considering humans or programs of a different type. Interestingly [16] has shown that his MCTS implementation reaches a plateau against GnuGo when the number of simulations goes to infinity. This shows limited scalability, to be confirmed by situations (practically) unsolved by Monte-Carlo Tree Search, presented in section below.

2.2 Counter-examples to scalability

The RAVE heuristic ([4, 14]) is known as very efficient in several games: it introduces a bias in H, based on permutations of simulations. It is nonetheless suspected that RAVE is responsible for the bad asymptotic behavior of some MCTS programs. We below recall some known counter-examples when RAVE is included, and then give a detailed presentation of other counter-examples which do not depend on RAVE.

Counter-examples based on RAVE values. Martin Müller posted in the computer-Go mailing list the situation shown in Fig. 1 (`http://fuego.svn.sourceforge.net/viewvc/fuego/trunk/regression/sgf/rave-problems/`). In this situation, their MCTS implementation Fuego does not find the only good move. This is due to the RAVE modification (discussed in section 2.2). The only good move is B2: but any simulation including B2, except if B2 is played as the first move as a simulation, is a loss for white - therefore the RAVE values are misleading. The underlying assumption of RAVE, namely the fact that a good move for now is a move which is good even when played later, is wrong here.

Other counter-examples. Importantly, Fig. 2 from [3] shows that there are some bad behaviours even without RAVE values.

Fig. 1. White to play, an example by M. Müller of bad scalability due to Rave. RAVE gives a very bad value to the move B2 (second row, second column), because it only makes sense if it's the first move, whereas this is the only move avoiding the seki (otherwise, black A5 and the two black stones A2 and B1 are alive).

Below, we propose new clear examples of limited speed-up, that have the following suitable properties:

- These situations are extremely easy for human players. Even a beginner can solve them.
- These counter-examples are independent of RAVE, as shown in our experiments.

Such situations are given in Fig. 3. These situations are semeais; it is known since [10, 18] that MCTS algorithms are weak in such cases. We show that this weakness remains without RAVE and even with inclusion of specific tactical solvers.

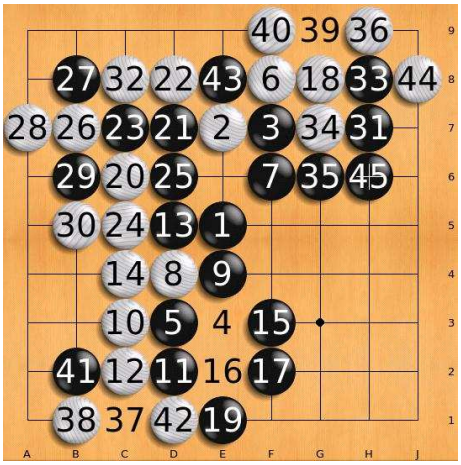

Fig. 2. White to play, an example of bad behavior shown in [3], independently of Rave values: in many cases (yet not always, this depends on the first simulations), MoGo is almost sure that he is going to win as white by playing C1, whereas it is a loss for white.

It is often said that classical solvers are able to solve semeais and therefore including expert modules should improve MCTS algorithms by including semeai solver. We therefore tested two ways of including expertise in MCTS:

- Expertise: we introduce a bias in the score, as usually performed in MCTS algorithms [6, 9, 18]. Some virtual wins are added to UCT statistics so that moves which are good according to our tactical semeai solver are more simulated; the idea, detailed in [6, 9, 18] consists in increasing the score of moves evaluated as necessary by the semeai solver, so that the heuristic H is more favorable to them. Only moves necessary for solving the semeai are given a bonus; no move at all if the semeai is won even if the player to play passes.
- Conditioning: then, all simulations which are not consistent with the solver are discarded and replayed. This means that when the solver predicts that the semeai is won for black (the solver is called at the end of the tree part, before the MC part), before the Monte-Carlo part, then the Monte-Carlo simulation is replayed until it gives a result consistent with this prediction. Human experts could validate the results (i.e. only simulations consistent with the semeai solver were included in the Monte-Carlo) and the quality of the solver is not the cause for results in Table 3; the coefficients have been tuned in order to be a minimum perturbation for having a correct solving for Fig. 3, left: the coefficients are (i) the size of semeais considered (ii) the weight of the expertise in the function H (for versions with expertise).

The results are presented in Table 3. In order to be implementation-independent, we consider the performance for fixed numbers of simulations; the slowness of the tactical solver can't be an explanation for poor results. From these negative

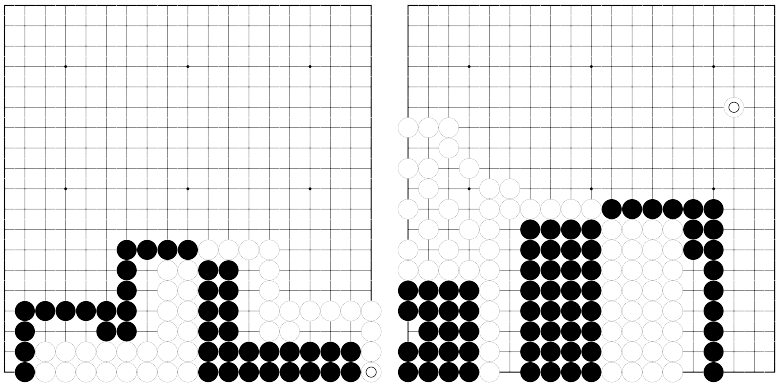

Fig. 3. Left: black to play. It is here necessary to play in the semeai. Right: black to play: playing in the semeai is useless as the semeai is won anyway (black has two more liberties than white) - good moves are outside the semeai. MoGo often makes the mistake of playing in the semeai.

results, and also for many trials with various tunings, all of them leading to success rates lower than 50 % against the baseline, we include that including expert knowledge is very difficult for semeais; it is true that tactical solvers can solve semeais, but they do not solve the impact of semeais on the rest of the board: in conditioning, if simulations are accepted as soon as they are consistent with the semeai solver, then the result of the semeai will be understood by the program but the program might consider cases in which black played two more stones than necessary - this is certainly not a good solving of the semeai.

These examples of bad behavior are not restricted to MoGo. Fig. 4 is a game played by Fuego and Aya in the 56^{th} KGS tournament (February 2010); Fuego (a very strong program by Univ. Alberta) played (1) and lost the game.

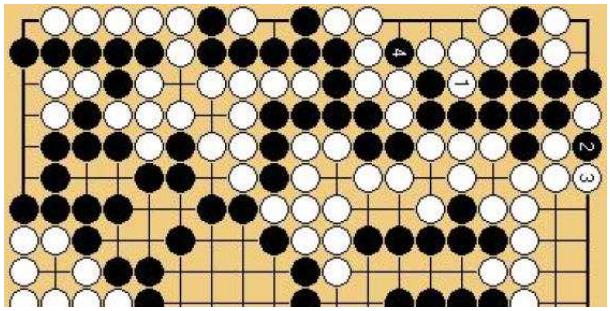

Fig. 4. Fuego as white played the very bad move (1) during the 56^{th} KGS tournament and lost the game. This is an example of situation very poorly handled by computers.

Version of the algorithm	Percentage of "good" moves
Situation in which the semeai should be played 1K sims per move	
MoGo	32 %
MoGo with expertise	**79 %**
MoGo with conditioning	24 %
MoGo with exp.+condit.	**84 %**
Situation in which the semeai should not be played 1K sims per move / 30K sims per move	
MoGo	**100%** / 58 %
MoGo with expertise	**95 %** / 51 %
MoGo with conditioning	**93 %** / 0 %
MoGo with exp.+condit.	**93 %** / 54 %

Table 3. These results are for Fig. 3; black should or should not play in the semeai (left or right situation in Fig. 3). All results are averaged over 1000+ runs. Bold is for results with more than 75 % on correct moves. We point out that the Go situations under consideration are very easy, understandable by very beginners. We see that (i) with 30K sims/move, many versions play the semeai whenever it is useless, and all versions play it with significant probability, what is a disaster as in real situations there are many time steps at which the MCTS program can have the opportunity of such a bad move and even only one such move is a disaster (loosing one stone!) (ii) removing RAVE does not solve the problem (iii) adding a tactical solver can work better (moderately better) with the traditional solution of adding expertise as virtual wins, but results remain very moderate, and far from what can do even a beginner. We also tested many parameterizations in self-play and none of these tests provided more than 50 % of success rate in self-play.

3 Message-passing parallelization

We here do not discuss the multithreaded parallelizations, which are detailed in [23, 13, 12, 8]. Multi-core machines are more and more efficient, but the bandwidth is nonetheless limited, and the number of cores is much bigger when we consider clusters than when we consider a single machine. This is why message-passing parallelization (in which communications are explicit and limited) must be considered. We'll see here that, in particular in 19x19, the technique is quite efficient from a parallelization point of view: the main issue for MCTS is not the computational power, but the limits to scalability emphasized in section 2.

The various published techniques for the parallelization of MCTS are as follows:

- *Fast tree parallelization* consists in simulating the multi-core process on a cluster; there's still only one tree in memory, on the master, and slaves (i) compute the Monte-Carlo part (ii) send the results to the master for updates. This is sensitive to Amdahl's law, and is quite expensive in terms of communication when RAVE values are used[15, 13].
- *Slow tree parallelization* consists in having one tree on each computation node, and to synchronize these trees slowly, i.e. not at each simulation but

with frequency e.g. three times per second [13]. The synchronization is not on the whole tree; it is typically performed as follows:

- Select all the nodes with
  - ∗ at least 5% of the total number of simulations of the root;
  - ∗ depth at most d (e.g. $d = 3$);
- Average the number of wins and the number of simulations for each of these nodes.

This can be computed recursively (from the root), using commands like *MPI_AllReduce* which have a cost logarithmic in the number of nodes. A special case is **slow root parallelization:** this is slow tree parallelization, but with depth at most $d = 0$; this means that only the root is considered.

– *Voting schemes.* This is a special case of tree parallelization advocated in [8], that we will term here for the sake of comparison with other techniques above **very slow root parallelization:** this is slow root parallelization, but with frequency $f = 1/t$ with t the time per move: the averaging is only performed at the end of the thinking time. There's no communication during the thinking time, and the drawback is that consequently there's no load balancing.

It is usually considered that fast tree parallelization does not perform well; we will consider only other parallelizations. We present in Table 4 the very good results we have in 19x19 and the moderately good results we have in 9x9 for slow tree parallelization.

Configuration of game	Winning rate in 9x9	Winning rate in 19x19
32 against 1	75.85 ± 2.49 %	95.10±01.37 %
32 against 2	66.30 ± 2.82 %	82.38±02.74 %
32 against 4	62.63 ± 2.88 %	73.49±03.42 %
32 against 8	59.64 ± 2.93 %	63.07±04.23 %
32 against 16	52.00 ± 3.01 %	63.15±05.53 %
32 against 32	48.91 ± 3.00 %	48.00±09.99 %

Table 4. Experiments showing the speed-up of "slow-tree parallelization" in 9x9 and 19x19 Go. We see that a plateau is reached somewhere between 8 and 16 machines in 9x9, whereas the improvement is regular in 19x19 and consistent with a linear speed-up - a 63% success rate is equivalent to a speed-up 2, therefore the results sill show a speed-up 2 between 16 and 32 machines in 19x19. Experiments were reproduced with different parameters with strong difference; in this table, the delay between two calls to the "share" functions is 0.05s, and x is set to 5%. The numbers with high numbers of machines will be confirmed in Table 5.

We can compare **slow root parallelization to the "voting scheme" very slow root parallelization:** with 40 machines and 2 seconds per move in 9x9 and 19x19, the slow root parallelization wins clearly against the version with very slow root parallelization, as shown by Table 5. with a frequency 1/0.35 against the very slow root parallelization. As a rule of thumb, it is seemingly

Framework	Success rate against voting schemes
9x9 Go	63.6 % ± 4.6 %
19x19 Go	94 % ± 3.2 %

Table 5. The very good success rate of slow tree parallelization versus very slow tree parallelization. The weakness of voting schemes appears clearly, in particular for the case in which huge speed-ups are possible, namely 19x19.

good to have a frequency such that at least 6 averaging are performed; 3 per second is a stable solution as games have usually more than 2 seconds per move; with a reasonable cluster 3 times per second is a negligible cost.

We now compare **slow tree parallelization** with depth $d = 1$, to the case $d = 0$ (slow root parallelization) advocated in [5]. Results are as follows and show that $d = 0$ is a not so bad approximation:

Time per move	Winning rate of slow-tree-parallelization (depth=1) against slow-root-parallelization
2	50.1 ± 1.1 %
4	51.4 ± 1.5 %
8	52.3 ± 1 %
16	51.5 ± 1 %

These experiments are performed with 40 machines. The results are significant but very moderate.

4 Conclusion

We revisited scalability and parallelism in MCTS.

The scalability of MCTS has often been emphasized as a strength of these methods; we show that when the computation time is already huge, then doubling it has a smaller effect than when it is small. This completes results proposed by Hideki Kato[16] or the scalability study `http://cgos.boardspace.net/study/index.html`; the scalability study was stopped at 524288 simulations, and shows a concave curve for the ELO rating in a framework including different opponents; Hideki's results show a limited efficiency, when computational power goes to infinity, against a non-MCTS algorithm. Seemingly, there are clear limitations to the scalability of MCTS; even with huge computational power, some particular cases can't be solved. We also show that the limited speed-up exists in 19x19 Go as well, and not with much more computational time than in 9x9 Go. In particular, cases involving visual elements (like big yose) and cases involving human sophisticated techniques around liberties (like semeais) are not properly solved by MCTS, as well as situations involving multiple unfinished fights. Our experiments also show that the situation is similar in Havannah with good simulations. The main limitation of MCTS is clearly the bias, and for some situations (as those proposed in Fig. 3) introducing a bias

in the score formula is not sufficient; even discarding simulations which are not consistent with a tactical solver is not efficient for semeai situations or situations in which liberty counting is crucial.

Several parallelizations of MCTS on clusters have been proposed. We clearly show that communications during the thinking time are necessary for optimal performance; voting schemes ("very" slow root parallelization) don't perform so well. In particular, slow tree parallelization wins with probability 94 % against very slow root parallelization in 19x19, showing that the slow tree parallelization from [13] or the slow root parallelization from [5] are probably the state of the art. Slow tree parallelization performs only moderately better than slow root parallelization when MCTS is used for choosing a single move, suggesting that slow root parallelization (which is equal to slow tree parallelization simplified to depth= 0) is sufficient in some cases for good speed-up - when MCTS is applied for proposing a strategy (as in e.g. [1] for opening books), tree parallelization naturally becomes much better.

Acknowledgements. We thank the various people who inspired the development of MoGo: Bruno Bouzy, Tristan Cazenave, Albert Cohen, Sylvain Gelly, Rémi Coulom, Rémi Munos, the computer-go mailing list, the KGS community, the Cgos server, the IAGO challenge, the Recitsproque company, the spanish federation of Go and in particular the organizers of the Spanish open 2009. We thank Grid5000 for providing computational resources for experiments presented in this paper.

References

1. P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud. Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go. In *Proceedings of EvoGames*, pages 323–332. Springer, 2009.
2. A. Auger and O. Teytaud. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, Accepted.
3. V. Berthier, H. Doghmen, and O. Teytaud. Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. In *Proceedings of Lion4*, page 14, 2010.
4. B. Bruegmann. Monte-carlo Go (unpublished draft http://www.althofer.de/bruegmann-montecarlogo.pdf). 1993.
5. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101, 2007.
6. G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.
7. G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.
8. G. Chaslot, M. Winands, and H. van den Herik. Parallel Monte-Carlo Tree Search. In *Proceedings of the Conference on Computers and Games 2008 (CG 2008)*, pages 60–71, 2008.

9. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.
10. R. Coulom. Criticality: a Monte-Carlo Heuristic for Go Programs, 2009. Invited talk at the University of Electro-Communications, Tokyo, Japan.
11. F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada, 2009.
12. M. Enzenberger and M. Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Proceedings of Advances in Computer Games 12*, 2009.
13. S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. The parallelization of Monte-Carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203, 2008.
14. S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.
15. M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.
16. H. Kato. Post on the computer-go mailing list, october, 2009.
17. L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
18. C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.
19. R. J. Lorentz. Amazons discover monte-carlo. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.
20. P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, 2009.
21. S. Sharma, Z. Kobti, and S. Goodwin. Knowledge generation for improving simulations in uct for general game playing. In *AI '08: Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence*, pages 49–55, Berlin, Heidelberg, 2008. Springer-Verlag.
22. F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Espagne, 2009.
23. Y. Wang and S. Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182, 2007.