PACHI: State of the Art Open Source Go Program[⋆]

Petr Baudiš and Jean-loup Gailly

Faculty of Mathematics and Physics,
Charles University Prague
pasky@ucw.cz, jloup@gailly.net

Abstract. We present a state of the art implementation of the Monte Carlo Tree Search algorithm for the game of Go. Our PACHI software is currently one of the strongest open source Go programs, competing at the top level with other programs and playing evenly against advanced human players. We describe our implementation and choice of published algorithms as well as three notable original improvements: (1) an adaptive time control algorithm, (2) dynamic komi, and (3) the usage of the criticality statistic. We also present new methods to achieve efficient scaling both in terms of multiple threads and multiple machines in a cluster.

1 Introduction

The board game of Go has proven to be an exciting challenge in the field of Artificial Intelligence. Programs based on the Monte Carlo Tree Search (MCTS) algorithm and the RAVE variant in particular have enjoyed great success in the recent years. In this paper, we present our Computer Go framework PACHI with the focus on its RAVE engine that comes with a particular mix of popular heuristics and several original improvements.

In section 2, we briefly describe the PACHI software. In section 3, we detail the MCTS algorithm and the implementation and heuristics used in PACHI. Section 4 contains a summary of our original extensions to the MCTS algorithm — an adaptive time control algorithm (Sec. 4.1), the dynamic komi method (Sec. 4.2) and our usage of the criticality statistic (Sec. 4.3). In section 5, we detail our scaling improvements, especially the strategy behind our distributed game tree search. Then section 6 summarizes PACHI's performance in the context of the whole Computer Go field.

1.1 Experimental Setup

We use several test scenarios for the presented results with varying number of simulations per move. Often, results are measured only with much faster time settings than that used in real games — by showing different relative contributions of various heuristics, we demonstrate that the aspect of total time available may matter significantly for parameter tuning.

[⋆] Supported by the GAUK Project 66010 of Charles University Prague.

In our "low-end" time settings, we play games against GNU Go level 10 [4] with single threaded PACHI, 500 seconds per game, i.e., about 5,000 playouts per move. In the "mid-end" time settings, we play games against FUEGO 1.1 [9] with four PACHI threads, fixed 20,000 playouts per move. In the "high-end" time settings, we play against FUEGO 1.1 as well but using 16 threads, fixed 250,000 playouts per move.

The number of playouts (250,000 for PACHI and 550,000 for FUEGO[1] in "high-end") was selected so that both use approximately the same time, about 4 seconds/move on average. We use the 19×19 board size unless noted otherwise. In the "high-end" configuration PACHI is 3 stones stronger than FUEGO so we had to impose a large negative komi -50.5 with PACHI always taking white. However, while PACHI scales much better than FUEGO, in the "mid-end" configuration FUEGO and PACHI are about even.[2] The "low-end" PACHI is stronger than GNU GO, therefore PACHI takes white and games are played with no komi. Each test was measured using 5000 games, except for the "low-end" comparisons; we used a different platform and had to take smaller samples.

2 The Pachi Framework

The design goals of PACHI have been (1) simplicity, (2) minimum necessary level of abstraction, (3) modularity and clarity of programming interfaces, and (4) focus on maximum playing strength.

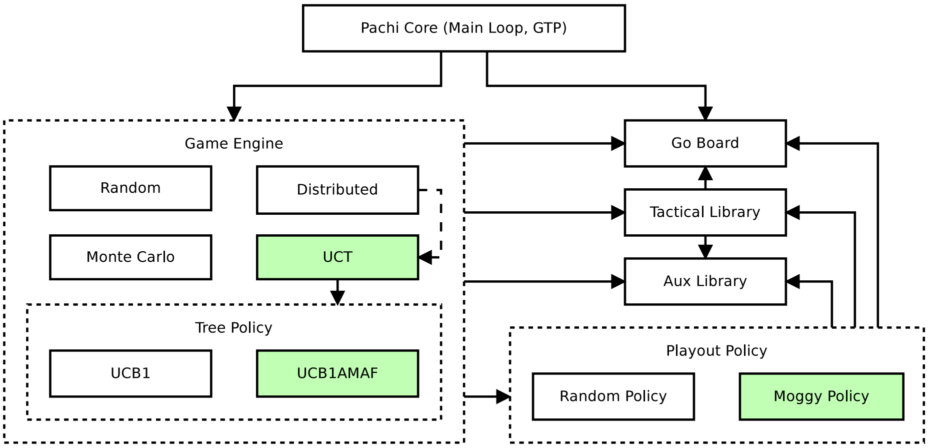

Fig. 1. Block schema of the PACHI architecture. When multiple options of the same class are available, the default module used is highlighted.

[1] The interpretation of *max_games* changed in FUEGO 1.0 such that it includes the count of simulations from reused trees. PACHI does not include them.

[2] When we tried to match FUEGO against the "low-end" PACHI, FUEGO was 110 Elo stronger.

Pachi is free software licenced under the GNU General Public Licence [11]. The code of Pachi is about 17000 lines of pure C, most of the code is richly commented and follows a clean coding style. Pachi features a modular architecture (see Fig. 1): the move selection policy,[3] simulation policy, and other parts reside in pluggable modules and share libraries providing common facilities and Go tools. Further technical details on Pachi may be found in [2].

3 Monte Carlo Tree Search

To evaluate moves, Pachi uses a variant of the Monte Carlo Tree Search (MCTS) — an algorithm based on an incrementally built probabilistic minimax tree. We repeatedly descend the game tree, run a Monte Carlo simulation when reaching the leaf, propagate the result (as a boolean value)[4] back to the root and expand the tree leaf when it has been reached $n = 8$ times.

Algorithm 1. NodeValue

```
Require: $\mathrm{sims}, \mathrm{sims}_{\mathrm{AMAF}}$ are numbers of simulations pertaining the node.
Require: $\mathrm{wins}, \mathrm{wins}_{\mathrm{AMAF}}$ are numbers of won simulations.
  $\mathrm{NormalTerm} \leftarrow \frac{\mathrm{wins}}{\mathrm{sims}}$
  $\mathrm{RAVETerm} \leftarrow \frac{\mathrm{wins}_{\mathrm{RAVE}}}{\mathrm{sims}_{\mathrm{RAVE}}} = \frac{\mathrm{wins}_{\mathrm{AMAF}}}{\mathrm{sims}_{\mathrm{AMAF}}}$
  $\beta \leftarrow \frac{\mathrm{sims}_{\mathrm{RAVE}}}{\mathrm{sims}_{\mathrm{RAVE}} + \mathrm{sims} + \mathrm{sims}_{\mathrm{RAVE}} \cdot \mathrm{sims}/3000}$
  $\mathrm{NodeValue} \leftarrow (1 - \beta) \cdot \mathrm{NormalTerm} + \beta \cdot \mathrm{RAVETerm}$
```

The MCTS variants differ in the choice of the next node during the descent. Pachi uses the RAVE algorithm [5] that takes into account not only per-child winrate statistics for the move being played *next* during the descent, but also (as a separate value) *anytime later*[5] during the simulation (the so-called AMAF statistics). Therefore, we choose the node with the highest value given by Algorithm 1 above, a simplified version of the RAVE formula [5] (see also Sec. 4.2).

3.1 Prior Values

When a node is expanded, child nodes for all the possible followup moves are created and pre-initialized in order to kick-start exploration within the node. The value (expectation) for each new node is seeded as 0.5 with the weight of several virtual simulations, we have observed this to be important for RAVE stability. The value is further adjusted by various heuristics, each contributing ε fixed-result virtual simulations ("equivalent experience" $\varepsilon = 20$ on 19×19, $\varepsilon = 14$ on 9×9). (This is similar to the progressive bias [7], but not equivalent.)

[3] The default policy is called "UCT", but this is only a traditional name; the UCT exploration term is not used by default anymore.

[4] In the past, we have been incorporating the final score in the propagated value, however this has been superseded by the Linear Adaptive Komi (Sec. 4.2).

[5] Simulated moves played closer to the node are given higher weight as in Fuego [9].

Table 1. Elo performance of various prior value heuristics on 19 × 19

Heuristic	Low-end	Mid-end	High-end
w/o eye malus	-31 ± 32	-3 ± 16	$+1 \pm 16$
w/o ko prior	-15 ± 32	-3 ± 16	$+10 \pm 16$
w/o 19 × 19 lines	-15 ± 33	-4 ± 16	-6 ± 16
w/o CFG distance	-66 ± 32	-66 ± 16	-121 ± 16
w/o playout policy	-234 ± 42	-196 ± 16	-228 ± 16

Most heuristics we use and the mechanism of equivalent experience are similar to the original paper on MoGo [5]. Relative performance of the heuristics is shown in Table 1. The Elo difference denotes the change of the program strength when the heuristic is disabled. It is apparent that the number of simulations performed is important for evaluating heuristics. The following six heuristics are applied.

- The “eye” heuristic adds virtual lost simulations to all moves that fill single-point true eyes of our own groups. Such moves are generally useless and not worth considering at all; the only exception we are aware of is the completion of the “bulky five” shape by filling a corner eye, this situation is rare but possible, thus we only discourage the move using prior values instead of pruning it completely.
- We encourage the evaluation of ko fights by adding virtual wins to a move that re-takes a ko no more than 10 moves old.
- We encourage sane 19 × 19 play in the opening by giving a malus to first-line moves and bonus to third-line moves if no stones are in the vicinity.

Table 2. The ε values for the CFG heuristic

	$\delta = 1$	$\delta = 2$	$\delta = 3$
19 × 19	55	50	15
9 × 9	45	40	15

- We encourage the exploration of local sequences by giving bonus to moves that are close to the last move based on δ, the length of the shortest path in the Common Fate Graph [13], with variable ε set as shown in Fig. 2. This has two motivations — first, with multiple interesting sequences available on the board, we want to ensure the tree does not swap between situations randomly but instead reads each sequence properly. Second, this is well rooted in the traditional Go strategy where a large portion of moves is indeed “sente”, requiring a local reply.
- We give joseki moves twice the default ε (using the joseki dictionary described below). This currently has no performance impact.

Table 3. Elo performance of some playout heuristics on 19×19

Heuristic	Low-end	Mid-end	High-end
w/o Capture	-563 ± 106	-700	-949
w/o 2-lib.	-86 ± 34	—	—
w/o 3×3 pats.	-324 ± 37	-447 ± 34	-502 ± 36
w/o self-atari	-34 ± 33	—	-31 ± 16

– We give additional priors according to the suggestions by the playout policy. The default ε is halved for multi-liberty group attack moves.

3.2 Playouts

In the game simulations (playouts) started at the leaves of the Monte Carlo Tree, we semi-randomly pick moves until the board is completely filled (up to one-point eyes). The move selection should be randomized, but heuristics allowing for realistic resolution of situations in various board partitions are highly beneficial.

We use the MoGo-like rule-based policy [12] that puts emphasis on localized sequences and matching of 3×3 "shape" board patterns. Heuristics are tried in a fixed order and each is applied with certain probability p, by default $p = 0.8$ for 19×19 and $p = 0.9$ for 9×9.[6] A heuristic returns a set of suggested moves; if the set is non-empty, a random move from the set is picked and played, if the set is empty (the common case), the next heuristic is tried. If no heuristic matches, a uniformly random[7] move is chosen.

See Table 3 for relative performance of the heuristics with the largest impact[8] (the Elo difference again denotes strength change when the heuristic is disabled).

We apply the following three main rules.

– With $p = 0.2$, ko is re-captured if the opponent played a ko in the last 4 moves.
– Local checks are performed — heuristics applied in the vicinity of the last move.
  • With $p = 0.2$, we check if the liberties of the last move group form a "nakade" shape.[9]
  • If the last move has put its own group in atari, we *capture* it with $p = 0.9$. If it has put a group of ours in atari, we attempt to escape or counter-capture other neighboring groups.

[6] Some of the precise values below are 19×19 only, but that is mainly due to a lack of tuning for 9×9 on our part.

[7] Up to one-point eye filling and the self-atari filter described later.

[8] Some of the low-probability heuristics represent only a few Elo of improvement and could not have been re-measured precisely with the current version.

[9] I.e., if we could kill the group by playing in the middle of the group eyespace.

  - If the last move has reduced its own group to just *two liberties*, we put it in atari, trying to prefer atari with low probability of escape; if the opponent has reduced our group to two liberties, we attempt either to escape or to put some neighboring group in atari, aiming to handle the most straightforward capturing races.
  - With $p = 0.2$, we attempt to do a simplified version of the above (safely escaping or reducing liberties of a neighboring group) for groups of three and four liberties.
  - Points neighboring the last two moves are (with $p = 1$) matched for 3×3 board patterns centered at these points similar to patterns presented in [12], extended with information on "in atari" status of stones. We have made a few minor empirical changes to the pattern set.
- We attempt to play a joseki[10] followup based on a board quadrant match in a hash table. The hash table has been built using the "good variations" branches of the Kogo Joseki Dictionary [17]. This has a non-measurable effect on the performance against other programs, but makes Pachi's play prettier for human opponents in the opening.

The same set of heuristics is also used to assign prior values to new tree nodes (as described above). Bad self-atari moves are pruned from heuristic choices and stochastically also from the final random move suggestions: in the latter case, if the other liberty of a group that is being put in self-atari is safe to play, it is chosen instead, helping to resolve some tactical situations involving shortage of liberties and false eyes.

4 MCTS Extensions

Below we discuss three MCTS extensions: Time Control (in 4.1), Dynamic Komi (in 4.2), and Criticality (in 4.3).

4.1 Time Control

We have developed a flexible time allocation strategy when the total thinking time is limited, with the goal of the longest search in the most critical parts of the game — in the middle game and particularly when the best move is unclear.

We assign two time limits (and a fixed delay for network lag and tree management overhead) for the next move — the *desired* time t_d and *maximum* time t_m. Only t_d time is initially spent on the search, but this may be extended up to t_m in case the tree results are too unclear (which triggers very often in practice).

Given the main time T and estimated number of remaining moves in the game[11] R, the default allocation is $t_d = T/R$ and $t_m = 2\,t_d$, recomputed on each move so that we account for any overspending.

[10] Common move sequence, usually played in a corner in the game beginning.

[11] We assume that on average, 25% of board points will remain unoccupied in the final position. We always assume at least 30 more moves will be required.

Furthermore, we tweak this allocation based on the move number so that t_d peaks in the middle game. Let t_M be the maximum time t_m at the end of the middle game (40% of the board has been played). In the beginning, we linearly increase the default t_d up to t_M until 20% of the board has been played (the beginning of the middle game) and set $t_d = t_M$ for the whole middle game. After this, or if we are in byoyomi, the remaining time is spread uniformly as described above.

For overtime (byoyomi), we use our *generalized overtime* specification: after the main time elapses, fixed-length overtime T_o for each next m moves is allocated, with n overtime periods available. Japanese byoyomi is a specific case with $m = 1$ while Canadian byoyomi implies $n = 1$. If overtime is used, the main time is still allocated as usual, except that $t_m = 3\,t_d$; furthermore, the lower time for t_d of the main time is the t_d for byoyomi, and the first $n - 1$ overtime periods are spent as if they were part of the main time. The time per move in the last overtime period is allocated as $t_d = T_o/m$ and $t_m = 1.1\,t_d$.

The search is terminated early if the expectation of win is very high $\mu \geq 0.9$ or the chosen move cannot change[12] anymore. In contrast, the tree search continues even after t_d time already elapsed if the current state of the tree is unclear, i.e., either of the following three conditions triggers.

- The expectations of the best move candidate μ_a and its best reply μ_{aa} are too different, $best_r = |\mu_a - \mu_{aa}| > 0.02$.
- The two best move candidates are equally simulated, i.e., $best_2 = \varepsilon_a/\varepsilon_b < 2.5$ for the playout counts $\varepsilon_a, \varepsilon_b$ of the two best moves.
- The best move (root child chosen based on the most simulations) is not the move with the highest win expectation.

Figures 2 and 3 show that using such a flexible time strategy results in an increase of an up to 80 Elo points performance compared to the baseline.[13]

The recently published time allocation by Erica [14] is similar, but while we focus on over-spending strategies, Erica focuses more on the middle game time

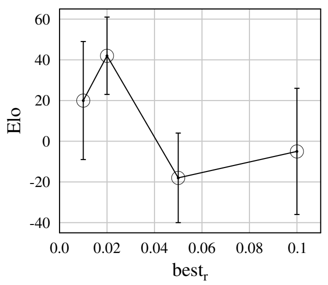

Fig. 2. Best — best child ratio $best_r$

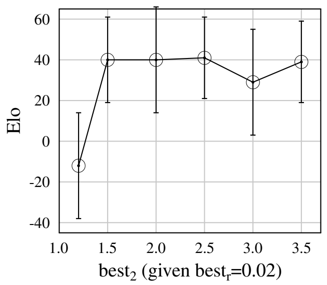

Fig. 3. Best — second best delta $best_2$

[12] We choose the most simulated node as the move to play. We terminate search early if the most simulated node cannot change even if it did not receive any more simulations for the rest of the t_m time.

[13] We used a very fast “low-end” scenario with 300 seconds per game.

allocation (adjusting time non-linearly, for a peak, not a plateau). Our middle game time allocation algorithm is not a source of significant performance benefits and we expect that our and ERICA's algorithm could be reconciled.

4.2 Dynamic Komi

The MCTS algorithm evaluates the possible moves most accurately when the winning rate is near 50%. If most simulations are won or lost, the resolution of the evaluation naturally gets more coarse. Such "extreme situations" are fairly common in Go — especially in handicap games or in the endgame of uneven matches. The dynamic komi technique [1] aims to increase MCTS performance in such situations by shifting the win-loss score threshold (komi) from zero to a different value; if a player winning 90% of simulations is required to win by a margin of 10 points instead of just 1 point, we can expect the winning rate to drop and the game tree will obtain information with a slight bias but also with a much higher resolution.

In PACHI, we have previously successfully employed mainly the Linearly Decreasing Handicap Compensation (LDHC) and Value-based Situational Compensation (VSC) [1]. Recently, we have introduced another novel method: the *Linear Adaptive Komi*. It uses LDHC up to a fixed number of moves, then increases the komi against PACHI if it wins with probability above a fixed sure win threshold (we use 0.85). This retains a good performance of LDHC for handicap games and allows winning by a large point margin when PACHI has a comfortable lead. Without this, PACHI is indifferent for the discrepancies between the winning moves, often steering to a 0.5 point win by playing what humans consider silly moves. At the same time, this strategy is more straightforward and more robust than VSC.

4.3 Criticality

RAVE improves over the plain MCTS by using approximate information on move performance gathered from related previous simulations. We can supply further information using the *point criticality* — the covariance of owning a point and winning the game [8,18],

$$Crit(x) = \mu_{win(x)} - (2\mu_{b(x)}\mu_{\mathrm{b}} - \mu_{b(x)} - \mu_{\mathrm{b}} + 1)$$

with $\mu_{b(x)}$ and $\mu_{w(x)}$ being the expectations of Black and White owning the coordinate, and μ_{b} and μ_{w} the expectations of a player winning the game.

The criticality measure itself has been already proposed in the past. We introduce an effective way to incorporate criticality in the RAVE formula — increasing the proportion of won RAVE simulations in the nodes of critical moves.

$$sims_{RAVE} = (1 + c \cdot Crit(x)) \cdot sims_{AMAF}$$

$$wins_{RAVE} = (1 + c \cdot Crit(x)) \cdot wins_{AMAF}$$

We track criticality in each tree node based on the results of simulations coming through the node. We use criticality only when the node has been visited at least $n = 2000$ times. $c = 1.1$ yields an improvement of approximately 10 Elo points in the "high-end" scenario, but $n = 192$ can achieve as much as 69 ± 32 Elo points in the "low-end" scenario. More details on this way of criticality integration may be found in [2]. A dynamic way of determining n may make the improvement more pronounced in the future.

5 Parallelization

Pachi supports both shared memory parallelization and cluster parallelization. It is highly scalable with more time or more threads, and scales relatively well with more cluster nodes. Fig. 4 shows the general scaling of Pachi. In all the distributed experiments and most single machine experiments, Fuego and Pachi both use 16 threads per machine and a fixed number of playouts. The measurements are done on a 19×19 board.

5.1 Shared Memory Parallelization

Historically, various thread parallelization approaches for MCTS have been explored [6]. In Pachi, we use the *in-tree parallelization,* with multiple threads performing both the tree search and simulations in parallel on a shared tree and performing lock-free tree updates [9]. To allocate all children of a given node, Pachi does not use a per-thread memory pool, but instead a pre-allocated global

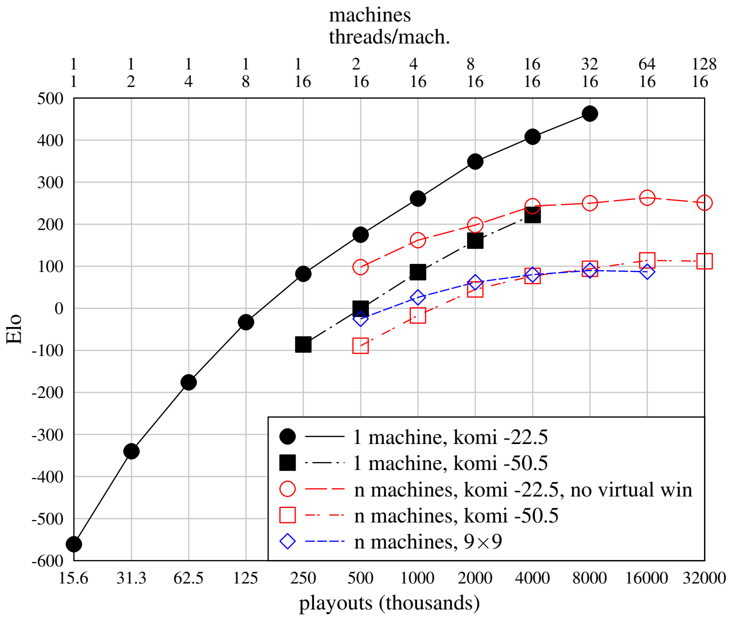

Fig. 4. Thread and distributed scalability

node pool and a single atomic increment instruction updating the pointer to the next free node. The leftmost curve in Fig. 4 shows scaling performance on a single machine when raising the number of playouts per move from 15,625 to 8,000,000 against a constant opponent (FUEGO 1.1 with 550,000 playous per move). To allow an average time per move of at least 2 seconds, the number of threads was reduced for PACHI up to 125,000 playouts per move; above this, the number of threads was kept constant at 16 and so the time per move increased (up to 2 minutes per move for 8,000,000 playouts).

The results show that PACHI is extremely scalable on a single machine. The strength improvement is about 100 Elo points per doubling in the middle of the range where PACHI and FUEGO have equal resources (16 threads each and same time per move). The improvement drops to about 50 Elo points per doubling at the high end, but shows no sign of a plateau[14]. Segal [20] reports a similar scalability curve, but with measurements limited to self-play and 4 hours per game. To measure the effect of the opponent strength, experiments were performed with both komi -22.5 and -50.5. As seen in Fig. 4, the curves are quite similar in both cases, only offset by a roughly constant number of Elo points.

To connect the concept of negative komi and handicap stones, we measured the Elo variation with a variable handicap and komi, as shown in Figs. 5 and 6. Against FUEGO, one extra handicap stone is measured to be worth approximately 70 Elo points and 12 points on the board. (The effect of one handicap stone would be larger in self-play.)

Most experiments were done with a constant number of playouts to improve the accuracy of the Elo estimates. We also measured the performance with fixed total time (15 minutes per game plus 3 seconds/move byoyomi) and a variable number of cores for PACHI.[15] The timed experiments in Fig. 7 demonstrate excellent scalability up to 22 cores[16].

Figures 7 and 8 show inflation of self-play experiments compared to games against a different reference opponent. Scalability results are most often reported only in self-play. This is in our opinion rather misleading. Self-play scalability is far easier than scalability against an opponent that uses different algorithms, for example +380 Elo points instead of +150 Elo points when doubling from 1 to 2 cores. The same effect is also visible in the distributed mode as shown in Fig. 10, where the 128-machine version is 340 Elo points stronger than the 2-machine version in self-play but only 200 Elo points stronger against FUEGO. Given the availability of at least two strong open-source Go programs (PACHI and FUEGO), we strongly encourage other teams to report scalability results against other opponents rather than self-play.

[14] We could not go beyond 8,000,000 playouts/move because of the resource requirements for 5000 games at more than 9 hours per game each.

[15] Since FUEGO lost on time far too frequently even with byoyomi, only PACHI used fixed time and FUEGO used a constant number of playouts.

[16] The timed experiments were run on 24-cores machines, with at least 2 cores reserved for other processes.

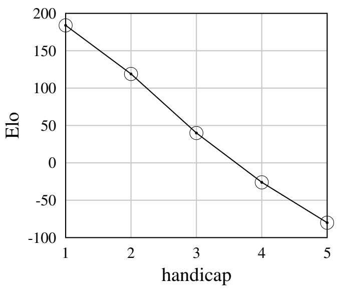

Fig. 5. Strength variation with handicap

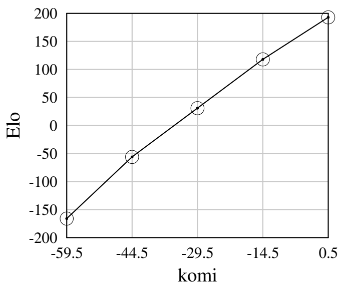

Fig. 6. Strength variation with komi

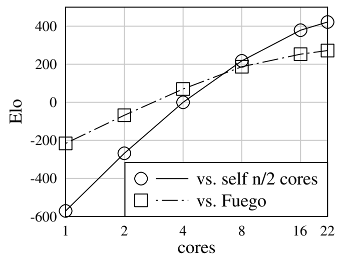

Fig. 7. Self-play scalability (fixed time)

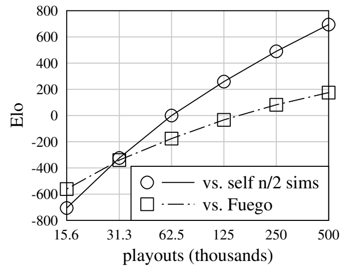

Fig. 8. Self-play scalability (fixed sims)

Fig. 9 shows the strength speedup relative to the number of cores, i.e., the increase in playing time needed to achieve identical strength play [6,20]. The tests are done using the "high-end" scenario, but Pachi uses a variable number of cores. The speedup is perfect (within the error margin) up to 22 cores — for 22 cores the measured speedup is 21.7 ± 0.5.

5.2 Cluster Parallelization

The MCTS cluster parallelization is still far from being clearly solved. Pachi features elaborate support for distributed computations with information exchange between the nodes, but it still scales much slower when multiplying the number of nodes rather than processors with a low-latency shared tree. The cluster version with 64 nodes is about 3 stones stronger than the single machine version. Node statistics are sent using TCP/IP from slave machines to one master machine, merged in the master, and the merged results are sent by the master back to the slaves. Only updates relative to the previously sent results are exchanged, to minimize the network traffic. The network is standard 1 Gb/s Ethernet, so it was critical to optimize it. Statistics are sent only for the first n levels in the tree. Surprisingly, we found the value $n = 1$ to be optimal (i.e., only the information about the immediate move candidates is shared). Understanding this should be the subject of further study. Mogo [3] goes up to $n = 3$ but it uses a

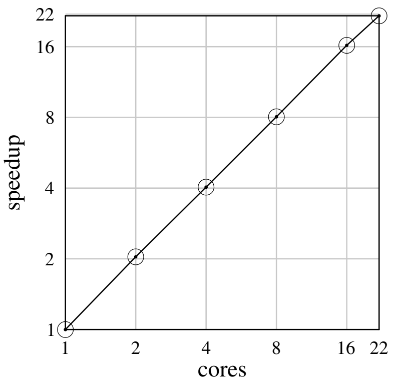

Fig. 9. Strength speedup.

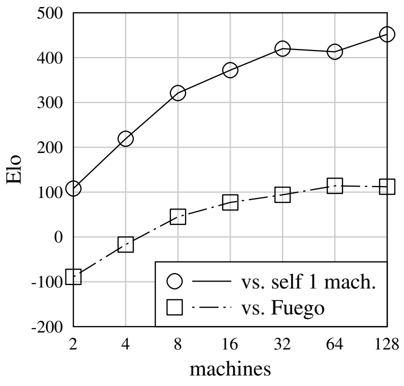

Fig. 10. Self-play scalability in the distributed mode

high performance network (such as Myrinet or InfiniBand) whereas Pachi uses standard Ethernet.

The distributed protocol was designed to be extremely fault tolerant. Nodes can be shut down arbitrarily. The Pachi processes run at the lowest possible priority and can be preempted at any time. The master sums the contributions of all slaves and plays the move most popular among them. In timed games, the master plays when more than half of the slaves indicate that they are willing to play now, or when the time runs out. In experiments with fixed number of playouts, the master plays when half of the slaves have reached their threshold, or when the total number of playouts from all slaves reaches a global threshold. These two tests can trigger quite differently in the presence of flaky slaves. We have used the former method for all the experiments reported here, but the latter method improves scalability further.

Virtual loss [6] aims to spread parallel tree descents — a virtual lost simulation is added to each node visited during the descent and removed again in the update phase. We have found that cluster parallelization is significantly more efficient if multiple lost simulations are added; we use $n = 6$. This encourages different machines to work on different parts of the tree, but increasing exploration by multiple virtual losses slighly improves the single machine case as well.

To encourage further diversity among machines, we introduced the concept of *virtual win*. Each node is given several virtual won simulations in a single slave (if the node number modulo number of slaves equals the slave number), therefore different nodes are encouraged to work on different parts of the tree, this time in a deterministic manner. We use a different number of virtual wins for children of the root node ($n = 30$) and for other nodes ($n = 5$). We also tried to use losses instead of wins; the results were similar so we kept using wins to avoid confusion with the quite different concept of virtual loss. Virtual losses encourage diversity between threads on a single machine; virtual wins encourage diversity between machines.

Fig. 4 shows that virtual wins measurably the improved scalability in a distributed mode. Going from 2 to 64 machines improved the strength by 160 Elo points without virtual win and by 200 Elo points with virtual win (compare the lines for komi -22.5 and komi -50). Distributed Depth-First UCT [23] probably performs better but it is significantly more complex to implement, while multiple virtual loss and virtual win only require a few lines of code.

Fig. 4 also shows that distributed scalability for 9×9 games is harder than for 19×19 games, confirming reports by the MOGO team [21]. The average depth of the principal variation was measured as 28.8 on 9×9 with 4 minutes total time per game, and 14.7 on 19×19 with 29 minutes total time per game.

6 Overall Performance

PACHI's primary venue for open games with the members of the public is the KGS internet Go server [19]. Instances running with 8 threads on Intel i7 920 (hyperthreading enabled) and 6 GiB of RAM can hold a solid 1-dan rank; a cluster of 64 machines with 22 threads each is ranked as 3-dan. (The top program on KGS ZEN has the rank of 5-dan.) Distributed PACHI regularly participates in the monthly KGS tournaments [22], usually finishing on the second or third place, but also winning occasionally, e.g., in the August 2011 KGS Bot Tournament.

The cluster PACHI participated in the Human vs. Computer Go Competition at SSCI 2011, winning a 7-handicap 19×19 match against Zhou Junxun 9-dan professional [16]. Zhou Junxun commented that PACHI played on a professional level when killing an invading white group (the bulk of the game). In the European Go Congress 2011 Computer Go tournament [10], distributed PACHI tied with ZEN for the first place in the 19×19 section.

In addition to algorithmic improvements, an enormous amount of tuning of over 80 different parameters also significantly improved PACHI's strength. However, at most one out of ten experiments results in a positive gain. Moreover, improvements become harder as the program gets stronger. For example, multiple virtual loss and virtual win initially provided a significant performance boost (30 Elo points each), but after other unrelated algorithmic improvements, their combined effect is now under 10 Elo points per doubling.

For this reason, we have also omitted full graphs of performance based on the values of various constants but describe just the optimal values. While we have originally explored the space of each parameter, resource limitations do not allow us to re-measure the effect of most parameters after each improvement. We can only make sure that we remain in the local optimum in all dimensions.

7 Conclusion

We have described a modern open source[17] Computer Go program PACHI. It features a modular architecture, a small and lean codebase, and a top-performing

[17] The program source can be downloaded at `http://pachi.or.cz/`.

implementation of the Monte Carlo Tree Search with RAVE and many domain-specific heuristics. The program continues to demonstrate its strength by regularly playing on the internet, with both other programs and people.

We have also introduced various extensions of the previously published methods. Our adaptive time control scheme allows PACHI to spend most time on the most crucial moves. Dynamic komi allows the program to cope efficiently with handicap games. A new way to apply the criticality statistic enhances the tree search performance. PACHI scales well thanks to multiple-simulation virtual loss and to our distributed computation algorithm including virtual win.

Acknowledgments. We have borrowed useful implementation tricks and interesting ideas from other open source programs FUEGO, GNU GO and LIBEGO [15]. Jan Hric offered useful comments on early versions of the paper. We would also like to thank the anonymous referees for their helpful suggestions.

References

1. Baudiš, P.: Balancing MCTS by dynamically adjusting komi value. ICGA Journal (in review)
2. Baudiš, P.: Information Sharing in MCTS. Master's thesis, Faculty of Mathematics and Physics, Charles University Prague (2011)
3. Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hoock, J.-B., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P., Yu, Z.: Scalability and Parallelization of Monte-Carlo Tree Search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 48–58. Springer, Heidelberg (2011)
4. Bump, D., Farneback, G., Bayer, A., et al.: GNU Go, `http://www.gnu.org/software/gnugo/gnugo.html`
5. Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., Teytaud, O.: Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 1–13. Springer, Heidelberg (2010)
6. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
7. Chaslot, G., Winands, M., van den Herik, H.J., Uiterwijk, J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. In: Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session (2007), `http://www.math-info.univ-paris5.fr/~bouzy/publications/CWHUB-pMCTS-2007.pdf`
8. Coulom, R.: Criticality: a Monte-Carlo heuristic for Go programs. University of Electro-Communications, Tokyo, Japan (2009), Invited talk, `http://remi.coulom.free.fr/Criticality/`
9. Enzenberger, M., Müller, M., Arneson, B., Segal, R.: 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)
10. European Go Federation: European Go Congress 2011 in Bordeaux, Computer Go (2011), `http://egc2011.eu/index.php/en/computer-go`

11. Free Software Foundation: GNU General public licence (1991), `http://www.gnu.org/licenses/gpl-2.0.html`
12. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA (2006), `http://hal.inria.fr/inria-00117266/en/`
13. Graepel, T., Goutrié, M., Krüger, M., Herbrich, R.: Learning on Graphs in the Game of Go. In: Dorffner, G., Bischof, H., Hornik, K. (eds.) ICANN 2001. LNCS, vol. 2130, pp. 347–352. Springer, Heidelberg (2001)
14. Huang, S.C., Coulom, R., Lin, S.S.: Time management for monte-carlo tree search applied to the game of go. In: 2010 International Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 462–466 (November 2010)
15. Lew, Ł.: libEGO — Library of effective Go routines, `https://github.com/lukaszlew/libego`
16. National University of Taiwan: Human vs. Computer Go competition. In: SSCI 2011 Symposium Series on Computational Intelligence (2011), `http://ssci2011.nutn.edu.tw/result.htm`
17. Odom, G., Ay, A., Verstraeten, S., Dinerstein, A.: Kogo's joseki dictionary, `http://waterfire.us/joseki.htm`
18. Pellegrino, S., Hubbard, A., Galbraith, J., Drake, P., Chen, Y.P.: Localizing search in Monte-Carlo Go using statistical covariance. ICGA Journal 32(3), 154–160 (2009)
19. Schubert, W.: KGS Go Server, `http://gokgs.com/`
20. Segal, R.: On the Scalability of Parallel UCT. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 36–47. Springer, Heidelberg (2011)
21. Teytaud, O.: Parallel algorithms (2008), `http://groups.google.com/group/computer-go-archive/msg/d1d68aaa3114b393`
22. Wedd, N.: Computer Go tournaments on KGS (2005-2011), `http://www.weddslist.com/kgs/index.html`
23. Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H., Ishikawa, Y.: Scalable distributed Monte-Carlo Tree Search. In: Borrajo, D., Likhachev, M., Lopez, C.L. (eds.) SOCS, pp. 180–187. AAAI Press (2011)