Local versus Global Tables in Minimax Game Search

by

Tana Wattanawaroon

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2014

Author: ____
Department of Electrical Engineering and Computer Science
May 27, 2014

Certified by: ____
Prof. Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ____
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Local versus Global Tables in Minimax Game Search

by

Tana Wattanawaroon

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 2014 in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Minimax Game Search with alpha-beta pruning can utilize heuristic tables in order to prune more branches and achieve better performance. The tables can be implemented using different memory models: global tables, worker-local tables and processor-local tables. Depending on whether each heuristic table depends on locality in the game tree, a memory model might be more suitable than others. This thesis describes an experiment that shows that local tables are generally preferable to global tables for two game heuristics used in chess-like games: killer move and best move history. The experiment is evidence that local tables might be useful for multithreaded applications, particularly ones that involve caching and exhibit locality.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Prof. Charles Leiserson for his continuous support and guidance throughout the completion of this thesis project. Since the beginning of the project in the SuperUROP program, he did not only provide useful feedback and advice, but he also taught me important communication skills and provided me with thoughtful insights about the world of academia.

I would also like to thank Tim Kaler and I-Ting Angelina Lee for their hours of much-needed support throughout the beginning phase of the project. Without their help, I would not be able to get up to speed with the codebase, which lies at the center of this thesis project, this quickly.

I would like to thank Don Dailey and Prof. Charles Leiserson for the codebase used for this project, and MIT's 6.172 staff of the years 2012 and 2013 that polished it until it became a nice and clean codebase for me to work on. I would also like to thank members of the Supercomputing Technologies Group at MIT for their help and feedback.

Contents

Chapter 1

Introduction

Game-playing programs (Game AIs) that play games like chess can be written for multiprocessor environments [2]. Most game playing programs use a version of minimax search on the game tree with *alpha-beta pruning*, a particularly important technique that eliminates calculations on unnecessary tree branches [6]. The game tree structure where subtrees of a position node can be evaluated independently lends itself well to multithreading. Several chess playing programs that utilize multithreading, like StarTech, ⋆Socrates, and Cilkchess, all developed by the Supercomputing Technologies Research Group at MIT, have proven their strength in international computer-chess championships [2].

The most important characteristic of a good game-playing program is the ability to evaluate game trees quickly, which is contributed to by many factors. One can improve the evaluation speed of a single node by changing low-level representations and modifying the algorithms used for computation, or one can increase the number of nodes concurrently evaluated using parallelization. One can also reduce the amount of work performed to evaluate a game tree by using alpha-beta pruning.

In order to further utilize alpha-beta pruning and speed up the game playing program, heuristics come into play. In particular, heuristics are used to achieve better "move ordering," which allows alpha-beta pruning to prune more branches, thus reducing the amount of work needed to perform [6].

Heuristics in game playing programs need not be stateless: they might depend on the order of node evaluations. When heuristics are used in a multithreaded game playing program, while subtrees of a position node can still be evaluated independently, it might not benefit as much from the heuristics as its single-threaded counterpart.

Another important characteristic of a good game playing program, especially for multithreaded ones, is determinism. It is by no means necessary for a multithreaded game playing program to output the same move across multiple evaluations of the same game tree, because there might be multiple moves that are equally good. A deterministic program, however, allows programmers to catch anomalies easier when the program is modified. It is also easier to ascertain a deterministic program's correctness. Experiments would become reproducible and there would be less variance.

Although heuristics help decrease computation time, they provide a source of nondeterminism. They can make the program more dependent on the order of execution across threads. Consequently, the program is more difficult to debug, and the heuristics' nondeterminism creates another hurdle to making the multithreaded program deterministic.

This thesis project focuses on using different memory models to implement heuristic tables for a minimax game search application. The purpose is to study how local and global heuristic tables impact the overall performance of a parallel game-playing program in terms of the amount of work and the speedup gained in a multiprocessor environment. Performance is mainly affected by the tables' ability to store useful information and generate the all-important "beta cutoff," which is the most important factor in reducing the amount of work needed to perform.

Chapter 2 of this thesis paper describes the game-playing program and the heuristics used in this study in more detail. It also explains the need for "local" heuristic table and the two different types of "local" tables. Chapter 3 details the experiments and the measurements used to compare the memory models as well as the results of the experiments. Chapter 4 concludes with the findings from the experiments.

Chapter 2

Heuristic Tables and Memory Models

This thesis project focuses on the impact of "global" versus "local" heuristic tables on parallel minimax game search programs. The following sections describe the case study, a parallel minimax game search program, and explain how heuristics are crucial in their performance. Then, different memory models for the heuristic tables, as well as the reasoning for using them, are discussed.

2.1 The Case Study: Parallel Minimax Game Search

A parallel minimax game search program serves as the case study of this thesis project. It parallelizes well, and heuristics are analogous to caching. Thus, this case study might be able to provide some insight on using different memory models for caching and storing information in other applications as well.

The program used for all the experiments in this project is a minimax game search program, parallelized with Cilk, that plays *Leiserchess* on an 8-by-8 board. Leiserchess is a chess-like board game similar to *Khet*, in which there are pieces that act like mirrors. Laser will bounce off mirrors and may destroy pieces it hits. It shares many characteristics with chess in terms of gameplay. Those who are interested in the game can read the full set of rules at `http://leiserchess.csail.mit.edu`.

The base program was furnished by Don Dailey and Charles Leiserson and was modified into a nice, clean piece of code by MIT's 6.172 staff of the years 2012 and 2013. As part of a final project, with help from Leo Liu, Ruwen Liu, Patricia Suriana and other 6.172 students of Fall 2013, the program is modified and optimized into a ready-to-use state.

The program implements the *negamax* [6] game search, a variant of minimax that is simpler to implement. It uses parallelized scout search [7] with the *young siblings wait* heuristic [4], where the first branch of a position node is searched serially.

While this thesis project focuses primarily on heuristics used in parallelized minimax game search with alpha-beta pruning, it serves as a case study for parallel applications that utilize some form of heuristics, especially those that involve caching. Several heuristics that benefit from local tables in this case study use information from recently computed information. Therefore, other similar applications might benefit from this study.

2.2 The Three Heuristic Tables

In alpha-beta pruning, some good moves might generate a *beta cutoff*, indicating that several other moves cannot change a position node's computed score, and therefore need not be searched. If the children of a position node are ordered in such a way that moves generating beta cutoffs are searched first—the *move ordering* is ideal—the amount of searching work is reduced [6]. The importance is particularly clear in the serial execution of the game playing program, because a move, which might be pruned, will not be executed until the previous moves in the move ordering is completely searched.

The codebase utilizes three heuristic tables: the *transposition table*, the *killer move table*, and the *best move history table*. The first table reduces the amount of required computation by eliminating evaluations of repeated positions, while the other two tables store data that help improve the move ordering [4]. While they are all present in the codebase, the latter two are the focus of this thesis paper.

```
bool tt_is_usable(ttRec_t *tt, int depth, score_t beta) {
  // can't use this record if we are searching at depth higher than the
  // depth of this record.
  if (tt->quality < depth) {
    return false;
  }
  // otherwise check whether the score falls within the bounds
  if ((tt->bound == LOWER) && tt->score >= beta) {
    return true;
  }
  if ((tt->bound == UPPER) && tt->score < beta) {
    return true;
  }
  return false;
}
```

Figure 2.1: the code snippet for the transposition table that determines whether the stored score's quality is sufficient

Transposition Table

The transposition table uses the fact that many nodes in the game tree share the same board position to eliminate redundant computation. It stores information for the evaluated tree nodes so that it can be retrieved when necessary.

Assume that the score of subtree T_1 has been computed, and the algorithm is about to compute the score of subtree T_2. If the two subtrees' roots represent the same board position, the algorithm might skip the evaluation of T_2 and take the score from T_1 as its own score. To decide whether the algorithm can use the previously computed score, it must also consider the depths of T_1 and T_2. For game trees rooted at the same board position, trees with greater depth should provide a more accurate score, as it represents looking further ahead by a greater number of moves. Thus, each computed score has an associated *quality*, the depth of the subtree of which the score is calculated. Therefore, the score of T_2 can be taken from the previously calculated score of T_1 only if the previous score's quality is sufficient; that is, when T_1 is taller than T_2. Otherwise, while the highest scoring move from T_1 cannot be guaranteed to also be the highest scoring move for T_2, heuristically that particular move is still a good move, and the program puts it first in the move order.

The transposition table stores the scores of subtrees, indexed by the position represented at the root of the subtree. For each index, only the score with the highest quality is stored. Before calculating the score of a position, the algorithm looks up the transposition table. It either returns the stored score, if it has acceptable quality, or uses the move as the first move in the move ordering.

Killer Move Table

The killer move table utilizes the fact that the children of a position node represent very similar board positions, and heuristically a good move from one of those children's position might also be a good move for other children and, perhaps, their cousins.

The game tree exhibits some locality: nodes that are next to each other on the same level of the game tree represent board states that are fairly identical, especially if they are siblings—they only differ by a few pieces' position. For this reason, a move that generates a beta cutoff from one board position is likely to be a good beta-cutoff candidate for nearby positions as well. The killer move table stores several recent "killer" beta-cutoff moves for each tree level, and evaluations at the nearby positions in the tree give priority to these potentially good moves.

Best Move History Table

The best move history table provides a way to order the moves besides those in the two previous tables. It keeps track of the score for moving different pieces in different orientations to different squares on the board. Once a move is determined to score highest among all possible moves from a position, the scores in the table are adjusted so that the highest-scoring move is considered somewhat better than before.

More precisely, after the program finds the highest-scoring move from a board position, it gives that move an additive score boost in the best move history table, while other moves drop their scores by a multiplicative factor.

Using the Tables

When the program considers a position node, if there is a score with acceptable quality stored in the transposition table, it would not perform any searching on the subtree. It would instead immediately return the score stored in the table. The amount of work is clearly reduced in this case, as the work associated with the subtree is eliminated.

Otherwise, all the tables are be used for move ordering, which hopefully gives early beta cutoff. Among the legal moves from a position node, the one in the transposition table is first in the order, followed by moves in the killer move table, from the most recently added to the least recently added. The remaining moves are ordered in descending order of their scores in the best move history table.

2.3 Reasoning for Local Tables

When the serial game search program with heuristic tables is parallelized, the simplest thing to do, and sometimes the right thing to do, is to make the tables *global*. That is, the tables are shared by all the processors and workers. Even though a global table eliminates the need to take care of multiple tables and keep the amount of memory used for the tables the same as in the serial version, this strategy might introduce data races and cause undesired behavior to the program. There are arguments that support the use of global transposition tables, but a better solution might be required for the killer move table and the best move history table.

It seems plausible to implement the transposition table as a global table. The transposition table is indexed on the exact board position (or rather the hash of the board position, to be technically precise), and the value stored in the table may be useful anywhere else in the tree for the same board position, especially if the quality of the stored score is sufficient.

The killer move table and the best move history table, however, rely more on the nodes' position in the tree. For serial executions, the tables' usefulness relies on the fact that the tree is searched depth-first, and thus two position nodes that are close by in the order of traversal would likely represent two fairly similar positions. Unlike the serial execution, the parallel execution can have multiple workers working on different parts of the tree representing very different board positions. If the tables are shared among the workers, the value stored by one worker might not be very useful to another worker. That is, the reasoning for why these tables help do not apply when the positions can be quite different.

2.4 Worker-Local versus Processor-Local

The Cilk runtime system has many components and levels of abstraction, and it might not be immediately clear how "local" a local table should be. Cilk schedules the work across *workers*, where each worker can use a processor core to process threads, smaller serial parts of the program. In the context of minimax game search, a thread is too small of a storage unit for heuristic tables, where the stored information should be useful to many other nodes.

This thesis project focuses on two kinds of "local" tables: *worker-local* and *processor-local* tables. Specifically, the worker-local heuristic table has a local copy for each worker that is accessible only by the corresponding worker. Processor-local tables work similarly, with a local copy for each processor core. The goal is to evaluate the use of both kinds of local tables for game heuristic tables and compare it against global table versions.

Chapter 3

Experiments

In order to see whether local tables are useful in practice, experiments are set up using heuristic tables with different settings—global versus local. The following sections provide a description of how these experiments are set and run. Then, the results of the experiments are elaborated.

3.1 Implementing Local Tables

The version of Cilk used, 4.8, does not have full support of worker-local storage. Thus, for experimental purposes, the code simulates worker-local tables and processor-local tables by simply duplicating the storage structure for each worker or processor, respectively.

```
static move_t killer[MAX_PLY_IN_SEARCH][2];
move_t killer_a = killer[ply][0];
move_t killer_b = killer[ply][1];
```

```
static move_t killer[NUM_WORKERS][MAX_PLY_IN_SEARCH][2];
int thread_id = __cilkrts_get_worker_number();
move_t killer_a = killer[thread_id][ply][0];
move_t killer_b = killer[thread_id][ply][1];
```

Figure 3.1: accessing the killer move table, global (top) and worker local (bottom)

For instance, for a game search with 12 workers using worker-local tables, there are 12 killer move tables, one for each worker. In order to determine which of the worker-local tables should be used for each call to the killer move heuristic, the program explicitly asks Cilk for the worker ID and access the table associated with that ID. Likewise, the version with processor-local tables asks the system for the processor ID and use the corresponding table.

3.2 Experimental Setup

The parallelized game-playing program were run on a multiprocessor machine using different settings for heuristic tables—global and local. Experiments were run on a 12-core Intel®Xeon®X5650 2.67 GHz machine with 46 gigabytes of memory. Its L1, L2 and L3-cache sizes are 32, 256 and 12288 kilobytes, respectively. Each experiment was run with 1, 2, 4, 6, 8, 10, 12 workers for comparison. Additionally, each experiment was also run with 18 and 24 workers, causing *over-subscription*—more workers than available cores—for more insight on the difference, if any, between worker-local and processor-local storage.

For consistency, the experiments were run on the fixed-depth mode instead of the fixed-time mode also available in the codebase. The program uses *iterative deepening*: it searches the game tree incrementally, starting at a small depth and increasing the depth as it goes. Iterative deepening is a concept that is useful in the fixed-time mode, because it gives the program a "partial result," where the program makes a move based on the deepest search it could perform within the given time limit. In addition, iterative deepening is also beneficial to the fixed-depth mode. Searching the tree at a smaller depth gives the program the sense of what the game tree looks like by storing information in the heuristic tables. At a larger depth, the information is likely to help prune large subtrees, and this information is not available if the tree is not searched incrementally. Therefore, spending time incrementally searching the tree can actually help improve performance in the long run [4].

In this thesis paper, because of iterative deepening, when the program is said to perform a "depth-n search," it actually performs the search on the game tree at all integer depths 1 through n, one after another. Unless mentioned otherwise, the search begins at the starting position of the game Leiserchess; that is, the root node of the game tree represents the game's starting position.

The experiments were run for different configurations of the killer move table and the best move history table. Each configuration is described by the type of memory model (local or global) used by the two heuristic tables, using abbreviations shown below.

`GK`	global killer move table	`GB`	global best move history table
`WK`	worker-local killer move table	`WB`	worker-local best move history table
`PK`	processor-local killer move table	`PB`	processor-local best move history table

3.3 Measurements

It is useful to consider different values measurable from an execution of the game search program and determine whether they provide any insight regarding the difference between local and global tables.

Perhaps the most obvious measure is the *running time* of the program, which is directly related to the program's performance in the context of timed competitions. Another measure is the *number of searched nodes* in the game tree. The more beta cutoffs the program can utilize, the smaller the number of nodes visited during the search, and thus the better performance. In serial game search programs, it is easy to make this number deterministic, because non-randomized heuristics behave in the exact same way across executions. Parallel ones, however, might include races, and one missed beta cutoff could require the program to do an extra search on a deep subtree. Thus, the number of searched nodes, even when executing on the same input, can vary significantly.

One might also measure the *speed* at which the program evaluates nodes, in nodes-per-second. If the parallel game search program has enough parallelism, however, the speed is relatively constant no matter what particular type of the heuristic tables are used. This is because the nodes (which one might treat as a thread in an execution graph) carry approximately the same amount of work. The speed, in nodes-per-second, would scale almost linearly in the number of workers used, up to the maximum number of cores. It can therefore be concluded that the speed is not an interesting parameter to measure to determine the effectiveness of local versus global heuristic tables.

For the actual program used in this experiment, as shown in Table 3.1, parallelism is ample for a 12-core machine, resulting in an almost-linear increase of speed in the nodes-per-second, as shown in Figure 3.2.

	depth 9	depth 10
parallelism	51.40	373.66
burdened parallelism	22.44	161.04

Table 3.1: parallelism of the program as measured by Cilkview

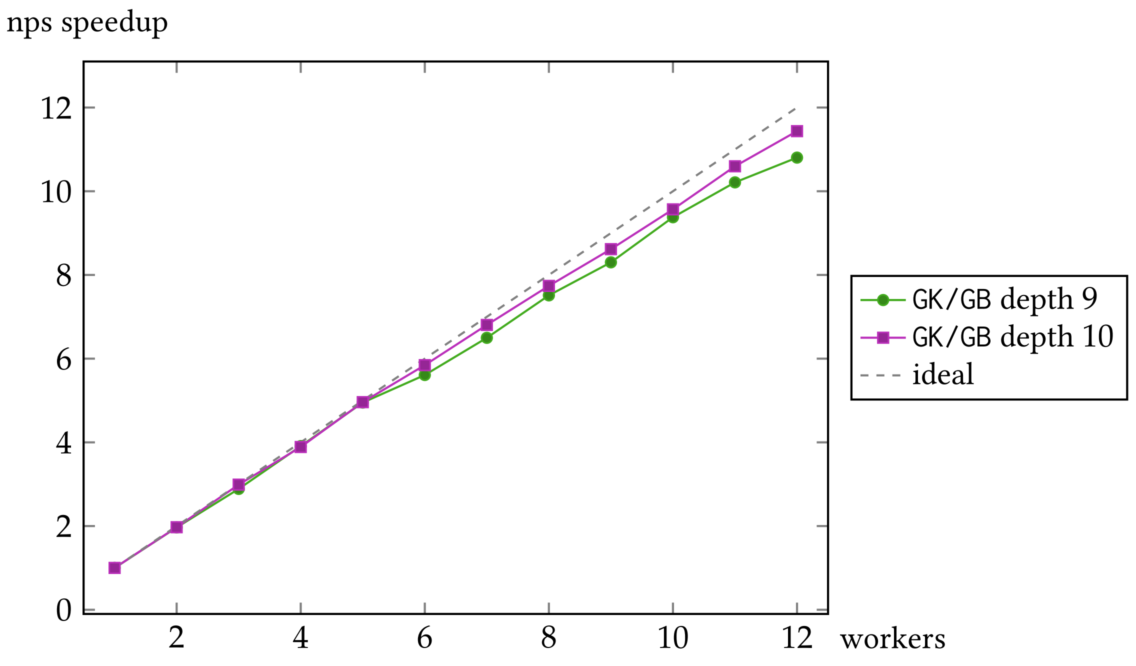


Figure 3.2: the speed in nodes-per-second, speedup compared to 1-core

It is important to note that in speculatively parallel game tree search, some of the work is performed “speculatively.” In other words, it might be clear in the serial execution of the same game tree that a certain subtree need not be evaluated, but that subtree is spawned to be evaluated in the parallel execution. The unnecessary evaluation might later be *aborted* when it has become clear, perhaps when evaluations of other subtrees are finished, that the subtree in question needs not be evaluated. While this mechanism increases parallelism and allows all the processors to be utilized efficiently, more work than necessary might be performed. The amount of extra work is also dependent on particulars of each execution, especially scheduling, and therefore the running time of multiple evaluations of the same game tree vary quite significantly.

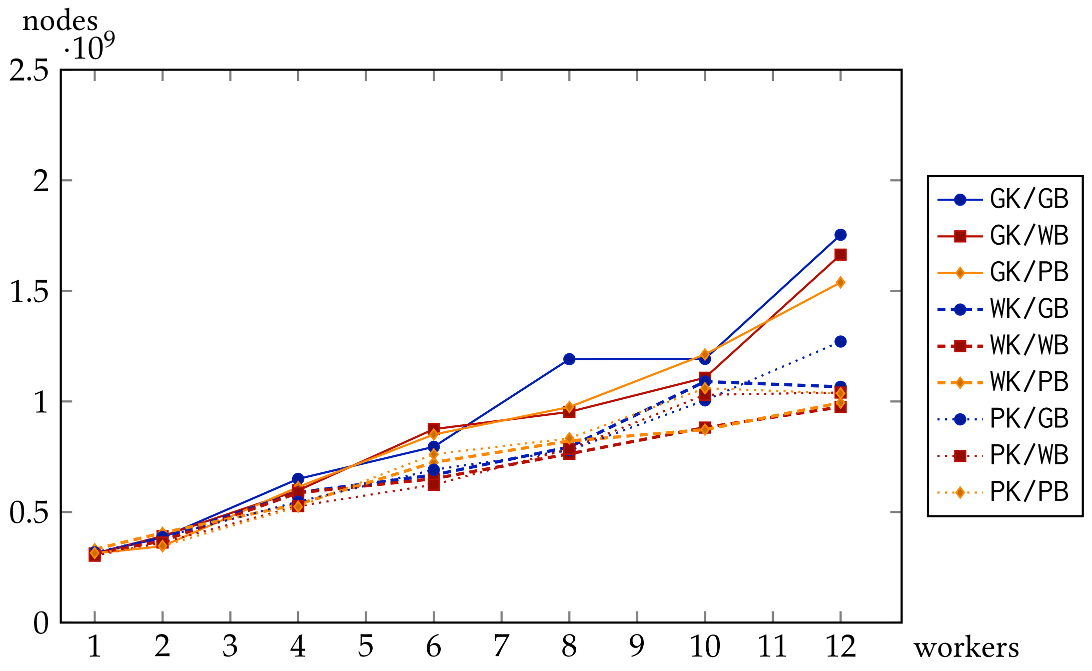
In order to measure the usefulness of the heuristic tables, the number of searched nodes seems to be a good measure, because the heuristics are used to decrease the amount of work needed to perform. Also, as stated earlier, the amount of work per node is nearly constant. It also follows that the running time would be about as good as a measure, since the speed in nodes-per-second does not vary by much.

3.4 Results

The parallel minimax game search program was executed with 1, 2, 4, 6, 8, 10, 12, 18 and 24 Cilk workers using different configurations of heuristic tables, global versus local. Recall that the experiments were run on a 12-core machine, and the 18-worker and 24-worker runs were over-subscribed.

Each experiment was repeated six times, and the running time and number of searched nodes were measured. In the plots, the value shown is the average of the six runs. The program runs both depth-9 searches and depth-10 searches for comparison.

When the program was run with at most 12 cores, as shown in Figures 3.3 and 3.5, the number of nodes visited increased approximately 2–6 times compared to the serial version



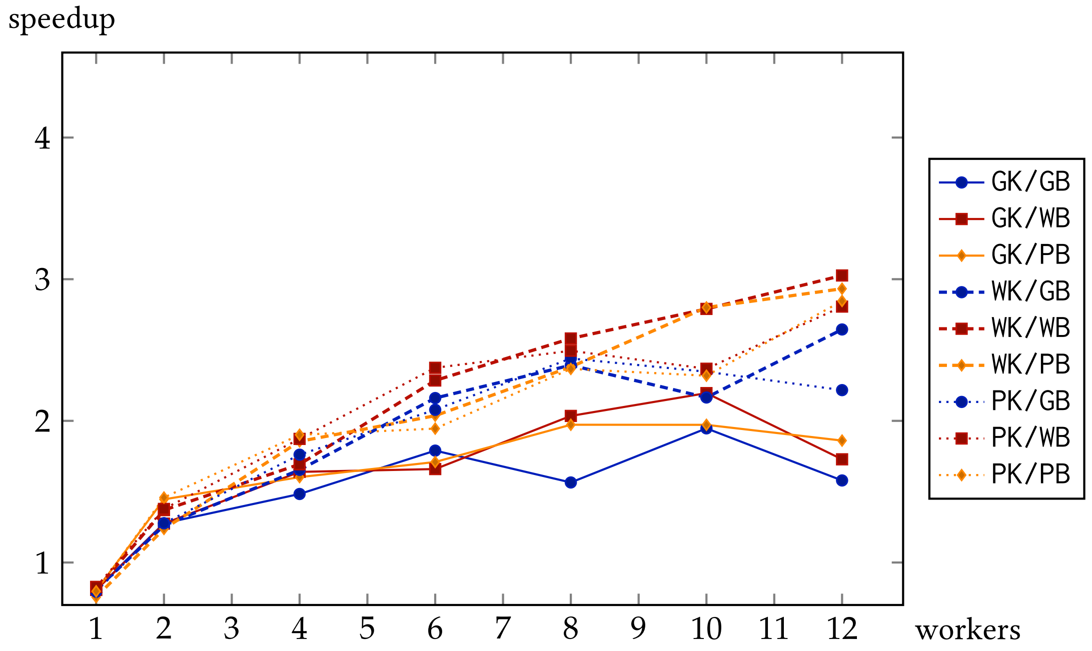
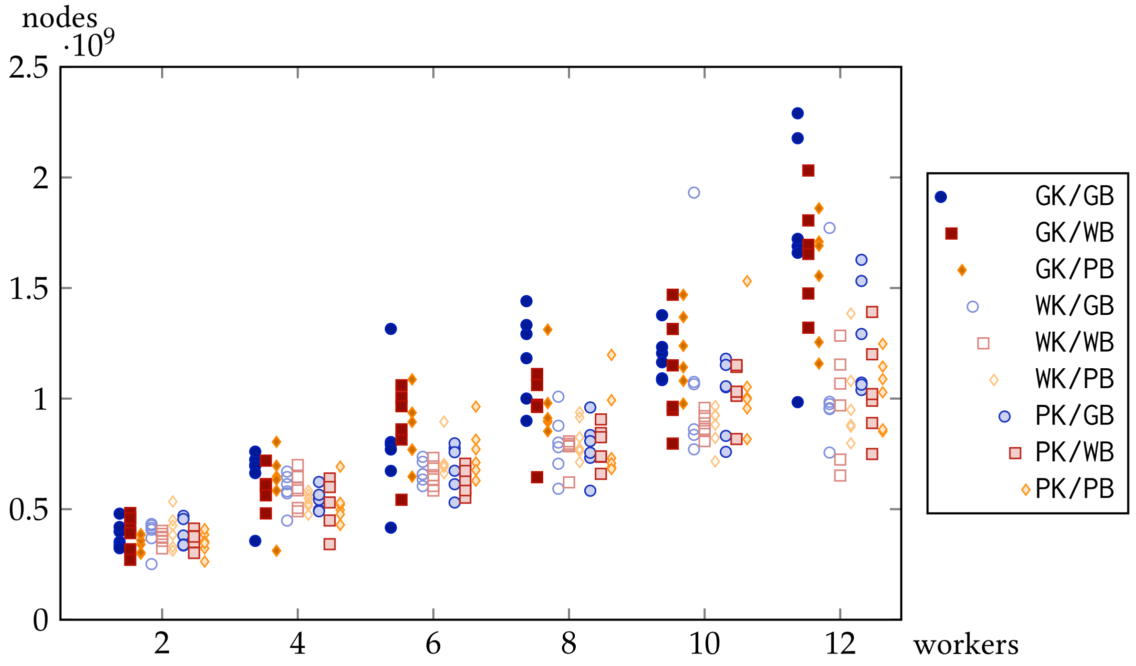


Figure 3.3: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-9** searches, when using up to **12** workers, is shown. The marks show the average of six runs.



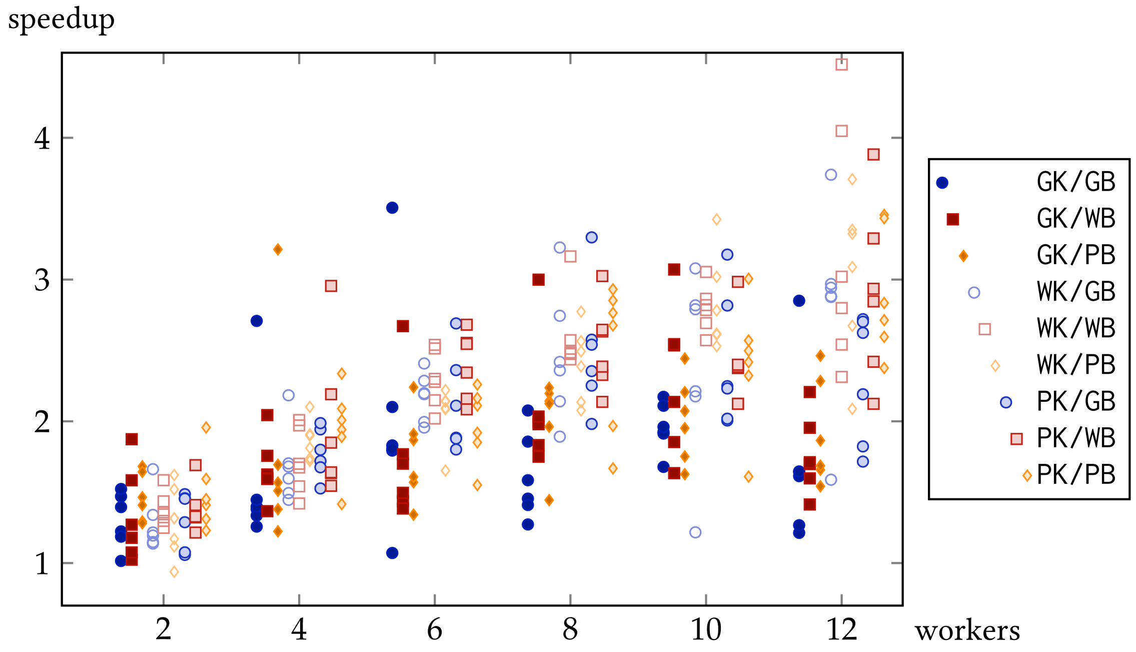
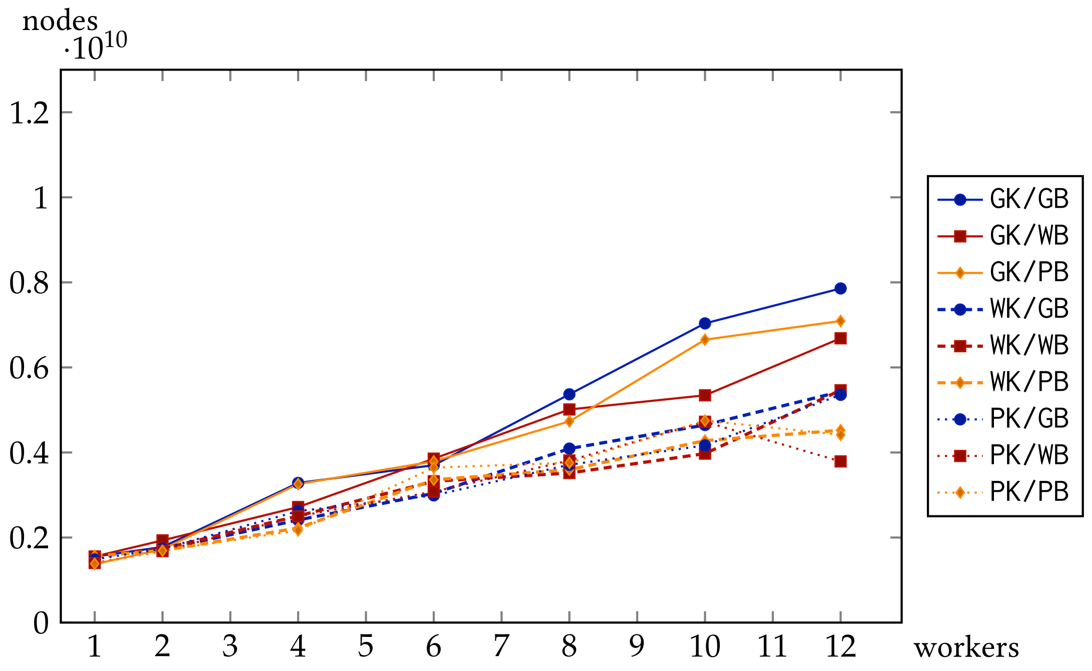


Figure 3.4: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-9** searches, when using up to **12** workers, is shown. The marks show the actual values of the six runs, stacked in columns (not shown for 1-worker). The columns are slightly shifted horizontally for clarity; they still represent exactly 2, 4, 6, 8, 10, and 12 workers.



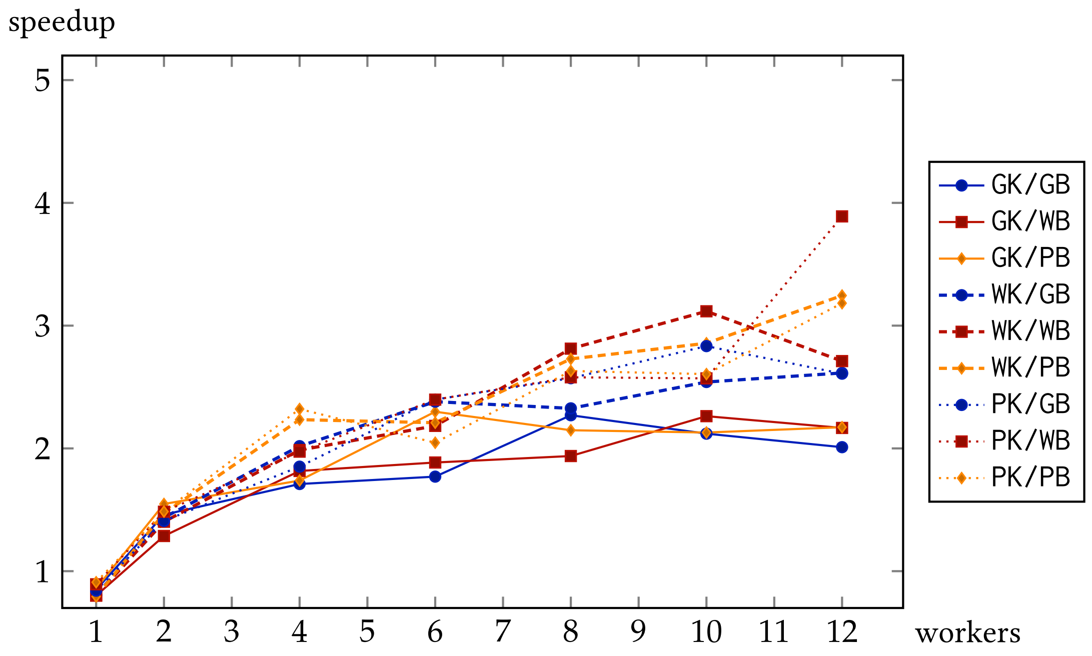


Figure 3.5: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-10** searches, when using up to **12** workers, is shown. The marks show the average of six runs.

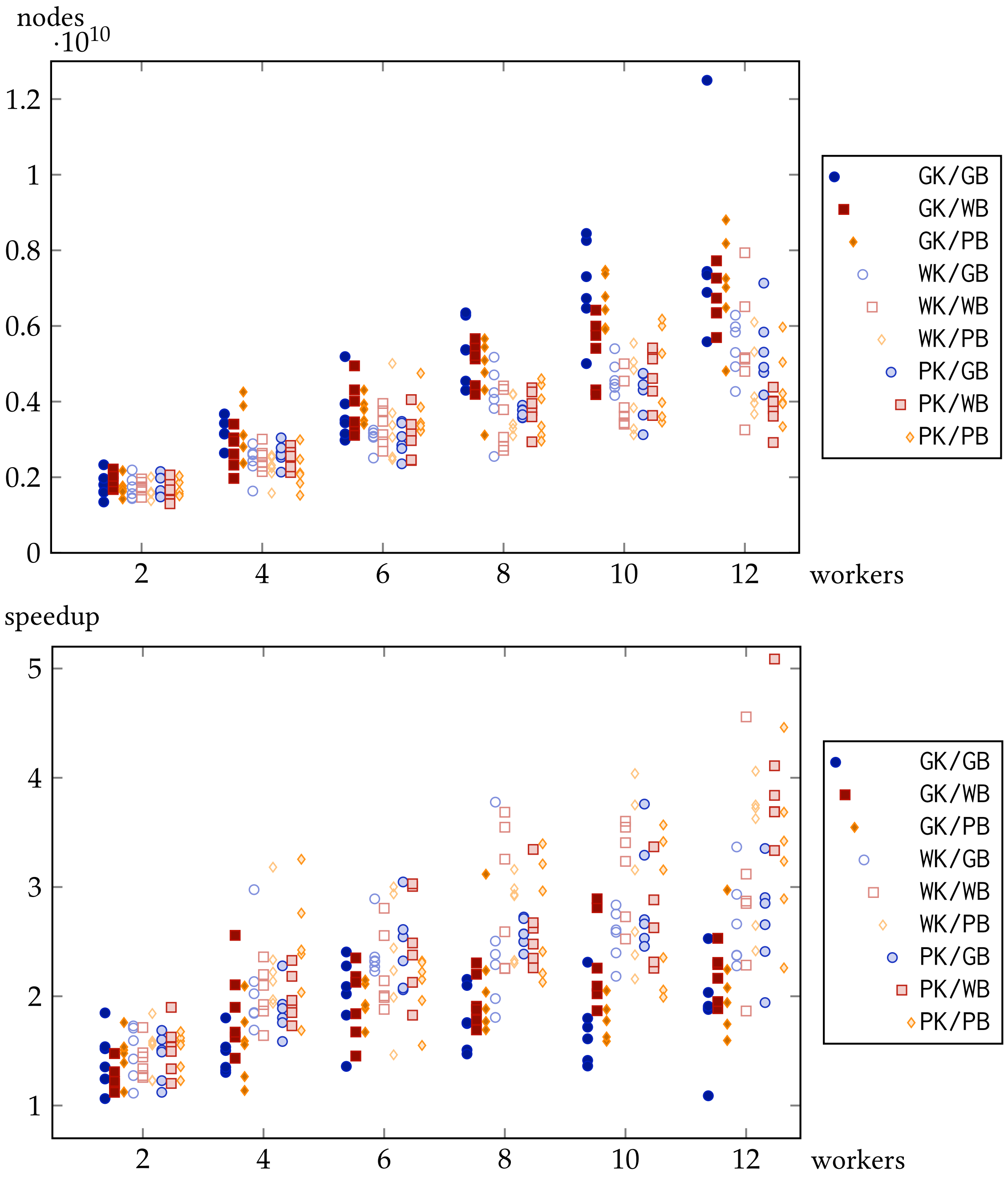
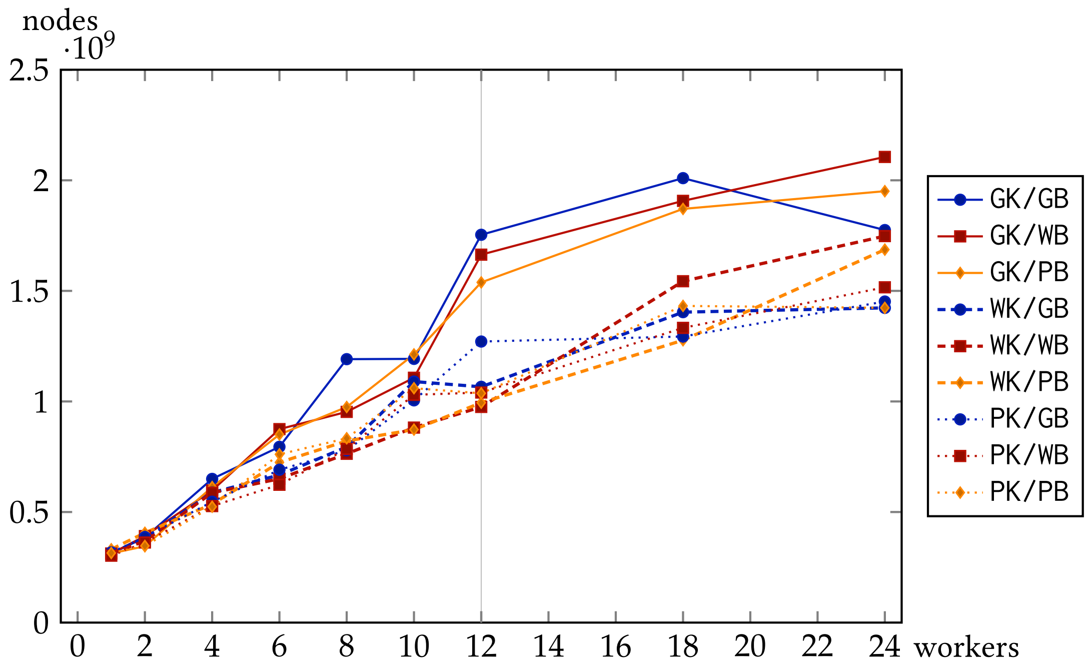


Figure 3.6: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-10** searches, when using up to **12** workers, is shown. The marks show the actual values of the six runs, stacked in columns (not shown for 1-worker). The columns are slightly shifted horizontally for clarity; they still represent exactly 2, 4, 6, 8, 10, and 12 workers.



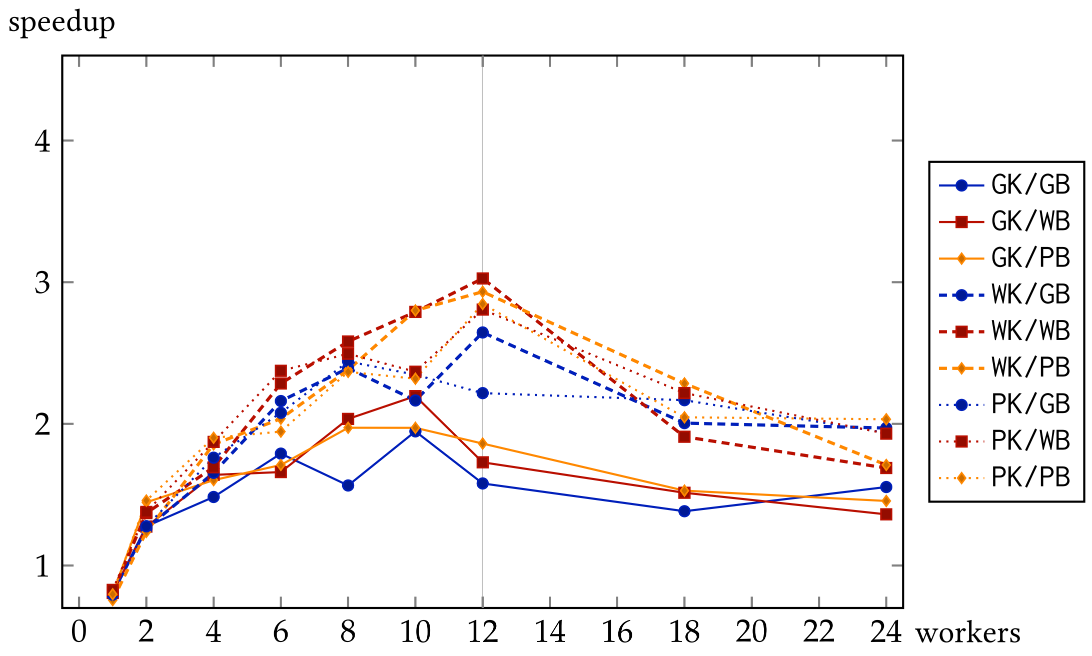
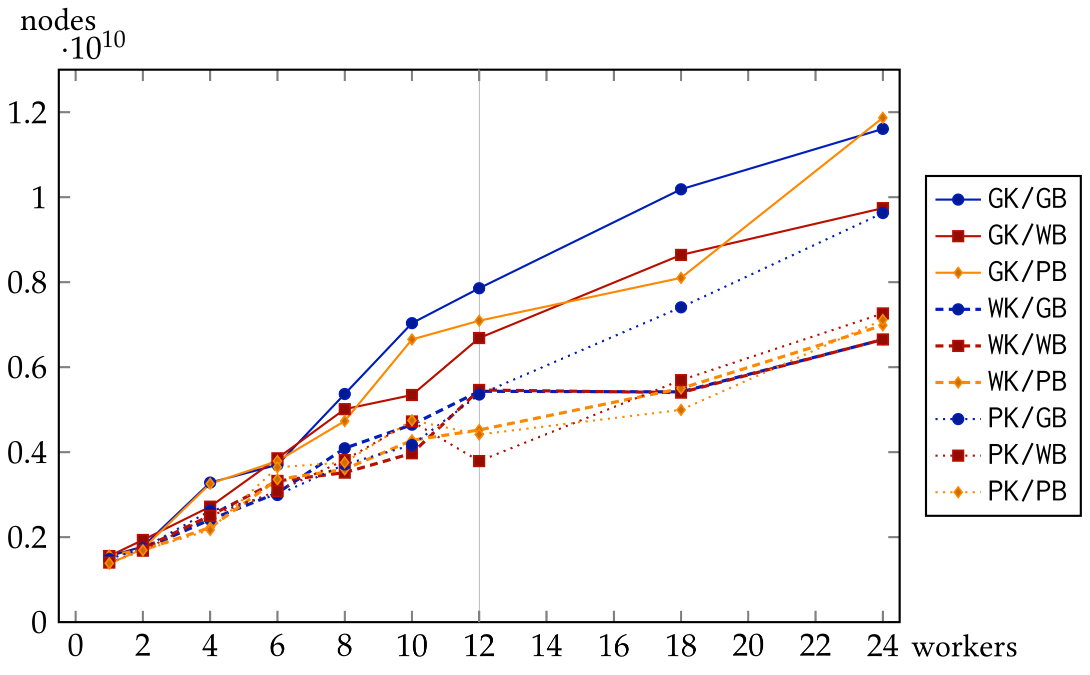


Figure 3.7: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-9** searches, when using up to **24** workers, is shown. The marks show the average of six runs.



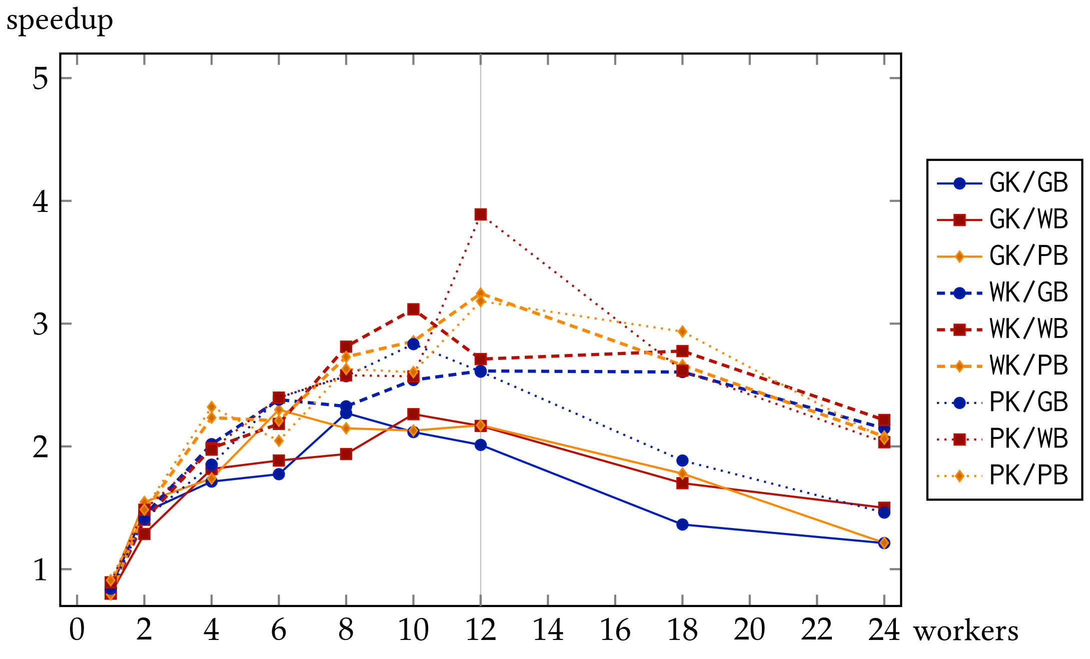


Figure 3.8: The number of searched nodes (top) and the running time speedup compared to the serial version (bottom) for **depth-10** searches, when using up to **24** workers, is shown. The marks show the average of six runs.

on both depth-9 and depth-10 searches. Also, on 12 cores, often below 3× speedup was achieved. The fact that the speedup was well below the ideal linear speedup is predictable, since the amount of speculative work performed increases with the number of workers.

Considering the impact of the local heuristic tables on the performance, the argument for using local tables mentioned earlier is supported by the results shown in the figures. Perhaps the most obvious observation one might make from the plots is that using a global killer move table seems to be a bad idea. Executions with a global killer move table, regardless of what the best move history table uses, search more nodes and have relatively small speedup when compared to other killer move table configurations. The difference in performance becomes the clearest when the number of workers is large. Also, most often the `GK/GB` configuration performs the worst.

The trend also applies for best move history tables; executions with a global best move history table usually obtains smaller speedup and searches through more nodes. However, the impact might not be as evident as that of the killer move table.

It is more difficult to be conclusive about the difference between the two variants of local tables: worker-local and processor-local. The best move history table, most of the time, performs slightly better when it is worker-local as opposed to processor-local. Not much can be said about the effect of the two local table variants on the killer move table.

The effect of the local table is much stronger on the killer move heuristic. It can provide as much as 40 percent speedup compared to the global table version. For the best move history table, the speedup obtained from using a local table is often below 10 percent.

In the case where more than 12 workers are used on a 12-core machine, the number of searched nodes still increases at the same rate, as shown in Figures 3.7 and 3.8. The speedup drops after 12 workers, however, because the computational speed in nodes-per-second is capped at 12 workers: no more than 12 workers at a time can actually perform computation. Additional context switching might also contribute to the speedup decline.

The trends seen in the case of 12 workers or less also apply when over-subscription occurs. The global killer move table still produces a big gap from its local counterparts in terms of the node count and the running time. The worker-local and the processor-local versions of the heuristic tables do not outperform each other significantly. This might be because both worker-local and processor-local tables have different disadvantages when running with more than 12 workers. Each worker in the worker-local table version is responsible for a smaller part of the tree, and therefore has less information that goes into the heuristic tables. On the other hand, context switching might render the information stored in the tables useless in the processor-local version.

Chapter 4

Conclusion

The results show that using global versus local heuristic tables affects performance. The following sections give a conclusion of the experimental results and outline some potential future work that can provide more insight for global versus local heuristic tables.

4.1 From the Results

For cache-like heuristics that rely on information recently computed or utilize locality, local versions appear to be a good choice of implementation. In the game context, the killer move table shows a clear performance improvement with either the worker-local or the processor-local version. Some improvement, although not as obvious, is seen when used for the best move history table. The distinction between the two local variants are not entirely clear, regardless of whether the number of workers exceed the number of processor cores. Some future work might be required to find the environment in which the performance of the two versions differ greatly.

4.2 Future Work

For experimental purposes, one might try to generalize the parameters used in this project. First, it would be interesting to see how local tables can be used for heuristics with locks.

Although locks make the program less parallel and lock contention can cause the program to decrease performance, it reduces the chance that the heuristic tables have data races and contain corrupted, useless information. Using local tables reduces the amount of contention. Introducing locks might not affect worker-local tables as much as processor-local tables, and therefore this experiment might help distinguish the performances of the two memory models.

It is also possible to experiment evaluating different game trees, particularly trees rooted at mid-game and end-game positions, and see whether the distance to the leaves affects the heuristics. Also, a natural extension is to work on different games, e.g., chess, and other applications utilizing tree search. The game heuristics act somewhat like caches, which could be useful in a different context.

Bibliography

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[2] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[3] Rainer Feldmann. Computer chess: Algorithms and heuristics for a deep look into the future. In *SOFSEM'97: Theory and Practice of Informatics*, pages 1–18. Springer, 1997.

[4] Rainer Feldmann, Burkhard Monien, and Peter Mysliwietz. *Game tree search on a massively parallel system.* 1994.

[5] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[6] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1976.

[7] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.