Performance Engineering of Software Systems November 8, 2021
Massachusetts Institute of Technology 6.172
Saman Amarasinghe, Charles E. Leiserson, and Jonathan Ragan-Kelley Handout 17

Project 4: Leiserchess

Last Updated: November 18, 2021

In this project, you will optimize and improve a bot that plays a variant of laser-chess — Leiserchess (pronounced "LYE-sir-chess"). Compared to past projects, this final project has a large and complex code base; navigating and optimizing it will put to test all the skills you have gained over this term.

Note: *It is a good idea to reread the Course Information handout before you get started! Also, reread this document regularly, since your understanding of it will improve as you become more familiar with the code base.*

Contents

1 Due dates

- ☐ *Team contract:* 11:59 P.M. on Wednesday, November 10, 2021
- ☐ *Recitation checkoff:* Friday, November 12, 2021
- ☐ *Design document submission:* 11:59 P.M. on Monday, November 15, 2021
- ☐ *MITPOSSE meeting deadline:* 11:59 P.M. on Monday, November 22, 2021
- ☐ *Beta submission:* 11:59 P.M. on Wednesday, November 24, 2021

- ☐ *Beta write-up:* 11:59 P.M. on Thursday, November 29, 2021
- ☐ *Presentation slides:* 11:59 P.M. on Monday December 6, 2021
- ☐ *Presentation:* Tuesday, December 7, 2021
- ☐ *Final submission:* **11:59 p.m.** on Wednesday, December 8, 2021
- ☐ *Final write-up:* **5:00 p.m.** on Thursday, December 9, 2021
- ☐ *Exhibition tournament:* 2:30 P.M. on Friday, December 10, 2021

Remember that status reports are due weekly at 11:59 P.M. every Thursday. All times are in the Eastern Time zone (ET).

2 Introduction

In this final assignment, you start with a high quality game-playing bot for Leiserchess 2021, a two-player laser-chess game similar to Laser Chess and Khet. The document *Leiserchess 2021: A Laser-Chess Game*, which you can find in the file `Leiserchess_2021.pdf` once you download the code base, describes the rules of the game. The code base is much larger than the code bases of Projects 1–3. The core AI encompasses many algorithms and heuristics, not just one or two simple algorithms.

In a sense, this project is like what you may encounter in real life as a software performance engineer. You must find opportunities to improve the performance of a sizable and complex program implemented by domain experts. It is your job to improve performance without compromising the program's correctness. But performance isn't just running time (although it is correlated). It is how well your bot plays against other bots. And correctness is not a binary mathematical notion. You may have some leeway to change what the program does, as long as it conforms to a more human measure of correctness. As in real-world situations, you have a limited amount of time to figure out what is slow and how to improve performance. Of course, you'll want to address the low-hanging fruit first (there's plenty), but eventually, you'll need to implement significant design changes. You must be judicious in what you take on, doing experiments and back-of-the-envelope-calculations to select wise courses of action that yield the best increment of performance for the time spent.

Your starting point is a software bot that plays the Leiserchess game. The AI we provide already does a good job of playing the game, and if you play games against it, you'll likely find that it is hard to beat. It implements Principal Variation Search (or PVS), a search algorithm commonly used in high-performance chess engines. We recommend that you browse the materials on the Chess Programming Wiki (`https://www.chessprogramming.org`) to learn about PVS. The website also contains a wealth of information about chess-playing programs, many of which are similar in software structure to our Leiserchess bot, as well as to game engines for other two-person games with perfect information, such as chess, checkers, and Go.

The code base includes several other major software components besides a search algorithm. The player software includes a transposition table, a static evaluation function, and a move generator. It also contains UI code that allows you or another program to interact with the game engine via a UCI-like interface (https://en.wikipedia.org/wiki/Universal_Chess_Interface). Outside of the player code itself, there is also ancillary software for autotesting, autotuning parameters, evaluating playing ability, and other useful tools.

Your mission is to improve the bot's ability to play Leiserchess. You may be able get improvements from implementing a smarter AI, but this approach turns out to be harder than it may seem at first. Since the game engine evaluates a player's available moves using heuristic search of a game tree, the more the positions and the deeper the search, the stronger the bot tends to be. Consequently, if two bots implement exactly the same functionality they run at different speeds, the faster one tends to be stronger because it can explore more positions and search deeper. Thus, for a reasonably good AI such as the one we have provided you, your can expect software performance engineering to be vastly more effective at quickly improving the strength of the bot than modifying the AI itself, although there may be some tweaks to the AI that can help.

You are free to employ any strategy you wish to improve your program's game-playing performance. The final measure of performance will be your program's ability to beat other game-playing programs, including other students' bots. But you will not compete with other students for your grade. (We'll resolve this apparent paradox in Section 6.) As always, we want you and your classmates consider yourselves a single learning team, sharing ideas and resources on Piazza (and receiving credit for doing so). But, after your final bot has been submitted, the course staff will run run the programs head to head in a friendly exhibition tournament for bragging rights, and a prize for the winners. Also, the course staff will set up a "scrimmage server" where you can try out the latest version of your bot against your classmates' bots, as well as against reference bots provided by the course staff.

3 Getting started

You can use your team name to get the project code:

```
$ git clone git@github.mit.edu:6172-fall21/project4_<team-name>.git project4
```

You can also browse the code using a read-only repository:

```
$ git clone git@github.mit.edu:6172-fall21/project4.git project4
```

You will also need to install some dependencies on your 6.172 virtual machine to use some of the project scripts, which you can do by running:

```
$ ./scripts/setup.sh
```

There is significantly more documentation in this project than in other projects; for convenience, here are additional supporting documents you will find useful:

1. `Leiserchess_2021.pdf`: Introduction to Leiserchess rules, available as linked in the Piazza announcement and also on Canvas under Files.
2. `README.md`: Description of the code base structure, how to launch a game server and how to test your code.
3. `player/README.md`: Summary of code files that you may find yourself modifying.
4. `tester/README.md`: Guide on how to use the autotesting framework.
5. `engine-interface.txt`: description of commands you can send Leiserchess at the command line. See `/player/leiserchess.c` for more information.
6. Slides from Leiserchess code walk.

The main codebase that you will be working with is the game engine in the `player/` directory. You can use `make` to compile the player code, which generates `leiserchess`. This program follows the Universal Chess Interface (UCI), which is a set of text commands sent via `stdin` that a Leiserchess engine must respond to at any point during runtime. Those commands are described in the UCI document, `engine-interface.txt`. The given implementation supports an additional command, `perft`. This command counts all possible moves to a given depth, which makes it a valuable debugging tool for your move-generation code.

Familiarize yourself with the Leiserchess game. The rules document provides some tips. Play a few games against the reference bot, a teammate, or a friend. You can run the game in your own browser using instructions included in the top-level `README.md` file in the codebase. You might be surprised how often students in the past have reported discovering a bug in their own code that they would not have discovered had they not learned the basic player strategies first hand by playing. Besides, in addition to helping you to learn how to write fast code, the course staff wants it to be fun. This project is about performance-engineering a *game* engine after all!

4 Background: alpha-beta search

The PVS algorithm used by the Leiserchess bot for game-tree searching is a refinement of the alpha-beta pruning algorithm, which is itself an optimization of a brute-force minimax search algorithm. Let us begin with a brief overview of alpha-beta algorithm.

Game trees

We can view each possible game ***position*** — configuration of pieces on a board, which player's turn it is to move, and sometimes other information — as a node of a ***game tree***, where the starting position is the root. The two players are usually called White and Black, corresponding to the traditional colors of chess pieces. Each child of a given node is the position that arises from making one of the player-on-move's legal moves from the node's position. If a player makes a

move that wins the game, the child position is a leaf in the game tree. A leaf may also arise because making a given move in a position results in a draw. Each level of the game tree is called a ***ply***. In chess literature, a ***move*** is a move by White followed by a move by Black, that is, two plies starting with White.

The same position may correspond to multiple nodes in a game tree. A ***transposition*** is a sequence of moves from a given starting position that results in a position that may also be reached by a different sequence of moves from the starting position. For example, suppose that White makes a move a, Black responds with b, and White then moves c. In many situations, the position that arises could also arise if White moves c, Black moves b, and then White moves a. The Leiserchess bot contains a ***transposition table*** to identify when a node has been encountered earlier in the search.

For simple games like Tic-Tac-Toe, the game tree is small enough that it can be completely generated, even without a computer, allowing for perfect knowledge to compute a winning (or at least non-losing) strategy. The game tree for complex games like chess and Leiserchess is far too large to generate completely because it grows exponentially with ply. For example, there are 20 possible positions after White's first move in chess, 400 possible positions after Black's first move, 8,902 positions after White's second move, etc. By the 15th ply, the number of positions is 2,015,099,950,053,364,471,960. No one knows exactly how many different chess positions can arise from the starting position, but in 1950, Claude Shannon estimated the number at 10^{120}. A more recent statistical estimate by John Tromp puts the number at $4.48 \times 10^{44} \pm 0.3710^{44}$ with a confidence of 95%. If you could explore 1 trillion board positions every picosecond, it would still take a trillion decades to explore them all. The Leiserchess game tree appears extensive as well. If you'd like to try estimating its size, please share your results on Piazza.

Alpha-beta pruning

Alpha-beta search uses a concept called ***pruning*** to avoid searching the entire tree. There are two types of pruning: ***mathematical*** and ***heuristic***. A mathematical prune occurs when a subtree of a game-tree search need not be explored because the values in the subtree cannot affect the score at the root. Alpha-beta and PVS are examples of mathematical pruning. Heuristic pruning occurs when a subtree is not searched because the game AI deems it unlikely to affect the score at the root. Examples of heuristic pruning in Leiserchess include futility pruning. For chess, mathematical pruning can reduce the average number of moves per position near the beginning of the game from about 31 to closer to 6, and heuristic pruning can reduce that even further to below 2.

To understand alpha-beta pruning, imagine a game tree where you are exploring the two subtrees of the root, as in Figure 1, from left to right. We begin to explore the left subtree and analyze the moves our opponent could make. One position is an immediate loss for us, labeled B in the figure. Assuming our opponent is as intelligent as we are, if we make the move corresponding to position A, our opponent will make us lose by making the move which leads to B. There is no point in exploring any of the other children of A, as we already know our opponent already has

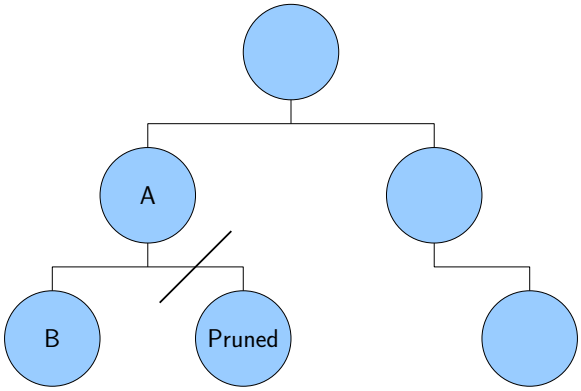

Figure 1: Alpha-beta pruning.

a good move B. We stop exploring this subtree, or prune away the subtree, and instead focus on the other subtree in the hopes of finding a better move. Another way of thinking about this is that once you have found out that a move is bad, there is no need to find out exactly how bad it is.

Static evaluation

Pruning helps cut away part of the game tree, but it doesn't solve the problem that the tree is still too big to explore to the point of a loss. Each position is assigned a score by a static evaluator function, where this score is a measure of how good the position is for the player. The static evaluator can be expensive, and it is often difficult to write an accurate one. Instead we explore the game tree to a given depth, and we use our static evaluator on the leaves. Using these scores, we can establish lower and upper bounds, `alpha` and `beta`, respectively, on the scores of the moves available to us at any position. The `alpha` value is the highest score we are guaranteed to get, and the `beta` is the lowest score our opponent can force us to get. In Figure 2, the `alpha` and `beta` values of the root node are initialized to negative infinity and positive infinity, since we have no knowledge of possible scores. We evaluate the left child to have the score of 5. Now we know we are guaranteed at least a 5 and update our `alpha` value accordingly. We then begin to explore the right child. The right child's values are initialized to that of the parent, 5 for `alpha` and infinity for `beta`. Node A evaluates to -3, meaning our opponent can force us into a score at least as bad as -3 if we choose the move that takes us to the right child's position. We update the `beta` value accordingly. Pruning occurs whenever `alpha` exceeds `beta`. In this example we saw that we had a move with a score of 5 available in the left subtree. By entering the right tree, our opponent can force us to get at most -3. There is no point in exploring the rest of the moves to find out as no better outcome can occur.

While having an accurate static evaluator looks to be the most important thing, there is a big trade-off between having an accurate static evaluator and a fast one. A fast static evaluator allows for a deeper search of the tree in a given time limit, which usually translates into a stronger player, as it can look more moves ahead. The provided evaluation function contains

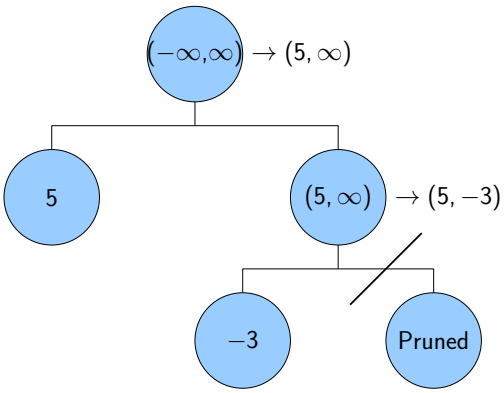

Figure 2: Pruning with alpha-beta values.

the best heuristics that the course staff could come up with. We recommend that you start by performance engineering the provided evaluator, though you're also free to modify it and add your own heuristics.

Parallelizing the search

For your final submission, your bot should be parallel. The lecture on Tuesday, November 16, will explain strategies for parallel game-tree search. **The beta will only evaluate your program on one core, and we will not evaluate parallel performance until the final submission.** While you are working on the design document and beta, however, you should identify sources of nondeterminism in the serial code, including operations involving the transposition table, the best-move history table, and the killer-move history table. You would be well advised to put in switches to turn off nondeterminism for some kinds of testing.

Here are a few ideas about how to parallelize your code. Keep them in mind as you optimize the serial code.

The idea behind parallelizing alpha-beta and Principal Variation Search is to search some selected subtrees of the game tree in parallel, which involves ***speculative parallelism*** (the subject of the lecture on November 19) because you might search a subtree that would have been pruned due to a beta cutoff. You should consider two things when introducing speculative parallelism into your program. The first is to *abort* computations that are not needed as soon as possible so that a beta cutoff terminates unnecessary searches of subtrees. The second is to avoid the execution of computations that are likely to be aborted, which allows your parallel program to obtain the full benefits of parallelization. After all, even if searches are correctly aborted, your program will not actually search deeper in the game tree if your additional processing cores are used to search subtrees that are highly likely to be aborted.

An additional consideration when parallelizing your code is contending with races. Care must be taken, for example, to ensure there are no races on the `alpha` and `beta` values. You will also find that the search algorithm maintains some tables for memoizing moves, etc. Races can occur

on these tables and you will have to decide whether the races are tolerable. ***Nota bene:*** **Unlike other programs you have seen so far, some races in game engines are OK.** In particular, you should consider how concurrency impacts the transposition table, killer-move table, and best-move history used by your programs. Some races may be tolerable, others may result in incorrect behavior, and others might be correct but compromise the benefits of various heuristics.

5 Deliverables

This final project is structured slightly differently from past projects. Mostly notably, this project starts with a design phase where each group will submit a design document. This document will be made public to the rest of the class much like beta code releases, and it will be the topic for the MITPOSSE review meeting. In addition, for the final submission, we require each group to create slides and deliver a 10-minute presentation involving all team members to the course staff. This section provides a breakdown of each part of the project.

5.1 Preliminaries

A team-formation Google form has been posted on Piazza. You are expected to work in groups of **four** students. (If you wish to work in a group of other than four students, please petition on the Google sign-up form.) If you cannot find enough classmates to form a full team of four, please sign up for the team that you have assembled (even if it is you alone), and the course staff will match you with others in the class. You may work with anybody in the class, including previous project partners and classmates who have the same MITPOSSE Deputy. Team assignments will be based on the team/partial team preferences entered on the form, and a public Google sheet of all teams will be posted on Piazza.

Team contract

All teammates must agree to a team contract, which is to be submitted on Gradescope by Wednesday, November 10. For guidance in writing your team contract, refer to Project 1's description. **Only one copy of the contract needs to be submitted on Gradescope. Please add all teammates to the submission.**

Lectures

The lecture on Tuesday, November 9, will overview the Project 4 code base and describe how the Leiserchess engine works. You would be well advised to attend this lecture, because this code base is substantially larger than the code bases for Projects 1–3. In addition, the lecture on Tuesday, November 16, will explain strategies for parallelizing the search.

Recitation checkoffs

The recitation on Friday, November 12, will go over the Leiserchess code base and present some helpful tips for the project. The following checkoffs are meant to familiarize you with parts of the code base, point out possible gotchas, and help you get started on modifying the code. Each team member should complete the checkoff items individually.

Checkoff Item 1: Modify the UCI interface to add a new command `fen`, which should print out the FEN string representation of the current game position.

Hint: See `main()` in `leiserchess.c`, and `pos_to_fen()` in `fen.c`. You can enter the UCI interface by running `make` in the `player/` directory, then running `./leiserchess`.

Checkoff Item 2: Open the UCI interface and type `go depth 3`, which searches for the best move to a depth of 3 and prints out each "best move so far" it finds during the search. Redo this process several times by quitting and restarting the UCI, then retyping the command. Continue until you observe some nondeterminism. Look through the source code and identify where the nondeterminism arises. Modify the code so that the relevant parts of the output (i.e., the parts other than timing information) are deterministic. Describe how you can support turning this source of nondeterminism on and off. (You need not write the code at this time.) Why do you think the original programmers made the program nondeterministic in the ways it is?

Hint: See `searchRoot()` in `search.c`.

Checkoff Item 3: Keeping your change from the previous part, now type `go depth 3` several times in the UCI interface *without* quitting and restarting in between. Which parts of the output are still changing from run to run? Why? Again without writing code, identify the source of the nondeterminism in the code, and describe how you could turn it off.

Hint: What global state is being modified?

Checkoff Item 4: In the UCI interface, type `perft`. This commands prints out the number of legal move sequences there are of a given depth, starting from the current position. Running

`perft` is useful as a quick "smoke test" indicating if you broke the move generation when you modified the code. (If your modified code passes `perft`, however, it still may contain a bug. There are no false positives, but there may be false negatives.) Modify the move generation to introduce a minor bug, and confirm that `perft` now returns different counts.

Hint: One way is to make a small change to `generate_all()` in `move_gen.c`.

Checkoff Item 5: Change the transposition table from direct mapped to k-way associative for a value k of your choosing. Looking at the infrastructure available in the project, how could you go about picking the best value of k, as well as the best size for the transposition table?

Hint: See `tt.c`.

Checkoff Item 6: Run the autotester on two bots that search to different depths. What would you need to make it easy to run the autotester on two bots that use transposition tables of different associativity or sizes?

Checkoff Item 7: *(Optional)* The function `get_centrality()` in the evaluation function computes the Manhattan distance of a square to the nearest corner of the board. Write a new function `get_euclidean_centrality()` to compute the Euclidean distance to the nearest corner. Modify the evaluation function to use `get_euclidean_centrality()` instead of `get_centrality()`. Use `perft` to verify that the change preserves the node count.

5.2 Design

Your team must submit a design document by Monday, November 15, which describes performance bottlenecks you discover in the game engine, outlines your initial ideas for how to address those bottlenecks, and describes a division of work going forward. This design phase is intended to help you start working with the codebase, become familiar with what each component of the codebase does, and observe how the components interact.

Design document

Your design document should describe your initial plan of attack for optimizing the game engine. Please describe how you analyze the performance of the game engine, including any back-of-the-envelope calculations, experiments, and preliminary changes you carried out to do your analyses. Include the results of your analyses in the design document to justify your initial optimization plans. Use a profiler to analyze the performance of the game engine and document any hot spots you see. Also, investigate the following parts of the codebase:

- the best-move history table,
- the transposition table,
- the killer-move table,
- the evaluation function,
- the move generator,
- the board representation.

Describe the performance enhancements to these components that you expect to do, how you think your improvements will affect your game engine's performance, and how much time you think it will take to make the improvements. If no changes to a component are warranted, explain why.

Focus on serial optimizations to your game engine for this design document, but include a brief description of your plans for parallelizing the program for your final submission. Make note of issues you expect to face during the parallelization process, such as what sources of nondeterminism in the program you expect to deal with. You will likely need to adjust your plans over the course of the project, especially after project parallelization strategies are presented in lecture. We encourage you to update your design document regularly, as it can serve as an effective mode of communication and planning among team members.

As the project is large and complex, you should make a coherent plan for testing your code, both overall and targeting individual small pieces of the program. By the design-document due date, you should have started on a basic testing framework to run unit tests, regression tests, integration tests, etc. At this point, you need not have written many tests, but we would like to see a "skeletal" testing framework that you can easily augment.

Planning the division of labor for this project is particularly important, because the project involves larger teams and a larger codebase. Please include in your design document a rough plan for how labor and responsibilities will be divided among the members of your team, using codenames for the team members.

Checklist for design document

To summarize, your design document should contain the following:

1. No self-identifying information! This includes your team name.
2. A summary of your understanding of how different parts of the program fit together.
3. Profiling data on the reference implementation and any initial efforts made in response to this data.
4. Two additions that you made to the UCI interface (beyond the checkoff).
5. One or more flags added to the makefile that turns off any sources of nondeterminism.
6. A modification to one of the tables and its effect on performance and/or Elo. Can you demonstrate that your change is significant? How many trials did you test on?
7. Optimizations you plan to make (and how you prioritize them), supported by profiling data. You should also estimate the impact of each optimization based on your profiling and estimate the amount of time each optimization will take. For large tasks, describe your plan for checking whether the task is worth the effort.
8. Outline the plan for eventual parallelization. Describe sources of nondeterminism and other hurdles you expect to encounter.
9. Work done on your testing framework and test suites so far, and plans for testing going forward. In addition, answer how you will compare two versions of your program. Will it be based on performance or Elo or some other metric? Will it be relative to each other or a third reference point? What is your threshold for establishing significance?
10. A breakdown of how your team plans on dividing the work, using codenames for team members to preserve anonmymity. You should convince the course staff that your group members are performing approximately equal work and that everyone is doing work worthy of a final project for 6.172.

After the due date, your anonymized design document will be made public to the rest of the class, much as for past projects. **Keep anything that could identify your team or one of its members out of the design document.** For the section on division of work, invent cool codenames for different team members.

Only one copy of the design document need be submitted on Gradescope. Please add all teammates to the submission.

Apart from submitting your design document on Gradescope, you should also push it to your Git repo so that the MITPOSSE Deputies can access it. If you prepare your document using a language like LaTeX or markdown, please also push your document source to your Git repo, as that will make it easier for the MITPOSSE Deputies to give you inline feedback comments directly on the MIT GitHub pull request.

MITPOSSE review

Unlike in previous projects, your MITPOSSE Deputy will not review your beta code and writeup. They will only review your design document. Please meet with your MITPOSSE Deputy by Friday, November 19.

5.3 Beta

Your beta bot should be serial code. Parallelization is not necessary and won't be evaluated at this time. Your beta code should reflect MITPOSSE comments on your design document. In your beta submission, include both your code, which will be made public to the class as with past project betas, and a beta group write-up.

Beta write-up

Submit on Gradescope your group's write-up for your beta release. The beta write-up is due on Thursday, November 25, one day after your beta code submission. You do not need to push the write-up to GitHub, as the MITPOSSE Deputies will only be reviewing your design document, not your beta code or write-up.

At this point, you should have a strong understanding of the game engine's structure and performance characteristics. You should also have made some progress with optimizations, and you should have a plan for what optimizations you are going to implement in your final submission. There are many potential changes you could make, but you should focus on the areas that will give you the best return on time spent. Feel free to copy text and figures from the design document you submitted previously and to build upon that text to describe any new performance bottlenecks you discovered as well as your updated design plan.

Checklist for beta write-up

To summarize, your beta write-up should contain the following:

1. Executive summary (overview).
2. Identification of performance bottlenecks.
3. Discussion of performance optimizations attempted and their impact (supported by performance measurements).
4. A report on team dynamics, including how work was divided and whether/how this differed from your design document. Your updated work breakdown should convince the course staff that your group members are performing equal work and that everyone is doing work worthy of a final project for 6.172.

5. How you responded to MITPOSSE comments in your beta source code.
6. A retrospective discussion of how your beta work has gone (good and bad), including a summary of performance progress.
7. Plans for parallelization and an updated list of optimizations you expect to make (and how you prioritize them), supported by profiling data. You should also estimate the impact of parallelization and each optimization.

Similarly to the previous projects, as part of your write-up, you must submit a log indicating how you spent your time on the project. You may choose to use a spreadsheet to keep your log, a web site such as `https://clockify.me`, or any other method that produces an easy-to-interpret log.

5.4 Final

The final deliverables for this project consist of a final presentation, a final code submission, and a final write-up, which are detailed below.

Final presentation

We expect each group to deliver a 10-minute presentation on Tuesday, December 7, one day before the final code and report submission is due. Suggested topics to cover in your presentation include:

1. Your overall optimization strategies.
2. How you implemented or approached your plan.
3. What kind of performance bottlenecks you observed and solved.
4. Brief overview of the performance results.
5. Analysis, if there is any.
6. Brief description of the work breakdown.
7. Other thoughts, such as strategies that you tried but didn't work. Feel free to share some war stories with us!

Please create your presentation using Google Slides by Monday, December 6. As the deadline approaches, we will post a link to a Google form where you can submit your slides.

The presentations will run for the entire day, and each team will be assigned a 10-minute slot. All group members must contribute approximately equally to both the preparation of the presentation and the presentation itself. We will provide more information about the presentation format after the beta deadline.

Final code submission

Please turn in your code by the final turn-in deadline at 5:00 P.M. on Wednesday, December 8. We will grade your program by extensive autotesting on parallel servers, including on machines that contain more cores than the ones you have access to.

Final report

Submit a final group write-up on Gradescope. Your report should build on the beta group write-up and additionally discuss the work you performed since the beta release. Include the same information mentioned in the beta write-up guide above (with the exception of future plans for the final and the MITPOSSE meeting comments). Also, provide an updated project log documenting how you spent your time on the final part of the project. Feel free to reuse (copy-paste) material from your beta write-up where applicable.

6 Evaluation

The performance of your code will be measured as follows. We will play everyone's bots, in addition to a few staff versions which include the starter code, in a tournament. The number of games in the tournament will be huge — large enough to minimize the effect of randomness. Using the outcomes of these matches, we compute the Elo rating for each team with sufficient accuracy to determine a final performance grade.

Your performance grade will be roughly based on the Elo difference between your team and staff bots, including the starter bot. Notably, it will *not* be based on the Elo difference between your team and any other student team. Thus, although your bot will play all the other teams, you will not compete with other teams for your grade, resolving the paradox mentioned in Section 2.

Please make sure that your program neither crashes nor produces illegal moves! A crash or illegal move attempt will count as a loss.

For the beta, we will evaluate bots on a serial server. For the final submission, we will evaluate them on a serial server and on multiple parallel servers, including servers with many more cores than you will have access to for development. Consequently, it behooves you to use Cilkscale and other means to gauge the scalability of your bot. If you don't manage to get a working parallel bot for the final, your grade for parallel performance and scalability will be based on comparing your serial bot to the parallel reference bots.

Grade breakdown

Grading for Project 4 will be based on the following point distribution.

	Design	*Beta*	*Final*
Design document	15%	—	—
Serial performance	—	25%	10%
Parallel performance and scalability	—	—	30%
Write-up	—	10%	5%
Final presentation	—	—	5%
Total	15%	35%	50%

The point distribution serves as a *guideline*, not an exact formula.

7 Words of wisdom

This section contains advice to help you produce the best bot you possibly can. Read it now, but set a repeating reminder to read it every few days. An item may not sink in until sometime later, at which point you may find that it is exactly the advice you need.

Testing your bot

- If you have made only performance improvements but no behavioral changes, you can test for correctness by comparing your bot to a previous stable version. If the bots search to the same depth (not with time control), then the two bots should make the same moves from the same starting board position. Since this comparison relies on deterministic behavior of the bots, it won't generally work when you parallelize your bot unless you implement a strategy for turning off nondeterminism.
- FEN strings can facilitate debugging. If your bot exhibits a bug deep in a search, instrument your program to record the FEN string for board positions so that you can figure out what the board position was right before the bug appears. Then, rather than having to run the entire search again to elicit the bug, you may be able to run it just from that board position. To identify games where the bot crashes, as well as move lists for those games, you can look at the autotester output (`.pgn` file).
- You might be tempted to evaluate your bot's performance by running the UCI command `go depth` and looking at the Nodes Per Second (NPS) metric. NPS is generally *not* a good performance indicator, as it may vary at different points in the game depending on the complexity of the search tree. It's better to run many games with the autotester and see whether there's a statistically significant difference in Elo rating.
- Don't rely just on tools to find bugs. Watching games on the scrimmage server can allow you to spot egregious game-play errors which may signal the presence of a bug. For example, you may find yourself asking, why is my bot committing suicide in this position?

Tools and resources

The code base contains tools to assist you in performance engineering the player code, and there are many online resources to explain things you may see in the code and give you ideas for improvements.

- The autotester framework allows you to evaluate multiple versions of your player to determine whether modifications you have made indeed help the player to perform better. Use the Elo software to compare game outcomes. The top-level `README.md` file provides instructions on how to run the autotester and the Elo software.
- Use the profiling tools you've learned in this class to identify performance bottlenecks. Profiling should reveal substantial low-hanging fruit.
- The autotuner can be an invaluable resource for optimizing parameters for your bot. It can help with determining weights for heuristics as well as finding optimal sizes for tables. Useful parameters to tune can be found in the `options.h` file.
- Run your code on the scrimmage server often. You can sometimes find issues with your bot that don't arise when comparing with earlier versions of your own code.
- Use Cilksan to check for race conditions in your parallel code. Program crashes count as losses when we evaluate your program.
- The chess-programming wiki (`https://www.chessprogramming.org`) provides invaluable information. Although Leiserchess differs from chess in some ways, many concepts for building quality chess engines should apply to Leiserchess as well.

Individual work

To contribute to a team most effectively, you must make the most of your individual time. Here are some things you can do to make yourself an effective teammate and individual contributor:

- Come to meetings prepared. Be on time. Do your homework. Ask for help if you need it. Get enough sleep.
- If one of your teammates is having trouble meeting their obligations, don't complain or put them down. Ask if there is something you can do to help them get on track. Don't expect everyone to have the same talents as you. You will be more successful as a team if everyone helps everyone else contribute to their fullest, even though one person's fullest may not be the same as another person's fullest.
- Budget time to become comfortable with how a game engine works. Without this background, it will be difficult for you to think of substantial optimizations or troubleshoot.

- Study how the code is organized. Although it's probably a waste of time to read through every line of code before beginning, try to understand the big-picture components of the code and how they interact.
- Keep an open mind. When looking at preexisting code, it's easy to be lured into following the footsteps of the original authors. You should ask yourself whether the data structures and algorithms chosen are appropriate. Is there a faster (or more parallelizable) method? Is a given piece of functionality even needed, or would your bot be stronger without it?
- The size and complexity of the code base may feel daunting, and it is. But evidence from prior semesters of 6.172 indicates that anyone who has made it this far through the semester *can* master the project. Don't panic, and just set about to do the common-sense things you need to do. As the ancient philosopher Lao Tzu said, "A journey of 1000 miles begins with a single step." Forget about the grade, and focus on learning and having fun while doing the project.

Development strategies

The Leiserchess engine comprises a significant amount of code. You will need to decide how to use your team's limited time to achieve the best results. Here are some suggestions:

- When evaluating whether a modification helps, it's tempting to play just a dozen or so games between the version without the modification and the version with the modification and go with whichever version wins more. For many program improvements, you'll need to run 1000 or more games to see a significant separation, not just a dozen or even 100. Many program changes net only a couple of percentage points advantage in win rate, yet many such changes can substantially improve your bot. The Elo software provides error bars that you can use to gauge whether you have performed enough tests to separate two versions statistically.
- Autotest with many versions, not just two. You can mine more information with the same number of games by running tournaments with more players, because the performance of bots tends to be transitive — A beats B and B beats C usually implies that A beats C — even though transitivity sometimes doesn't hold. By playing A, B, and C, you tend to gain information about A versus C even when neither is playing the other.
- Use fixed-depth autotesting, rather than timed games, to compare bots with functional differences that shouldn't affect performance, such as when tuning evaluation function weights. Typically, if you can show a statistically significant Elo separation with 5-ply searches, you will see the separation in deeper searches and timed searches. Fixed-depth testing gives you the advantage of repeatability, and you don't need a quiesced server for careful time measurements, allowing you to run the tests on any machine, including your laptop.

- Make incremental changes as much as possible, and **make each change a separate commit**. Try to avoid large changes that preclude you from compiling or running your code for extended periods of time. If you must make a large change, figure out how all teammates can contribute to ensuring that the code won't stay broken for long.

- The ramifications of changing the board representation can be extensive, affecting wide swaths of the code base. Plan to make representation changes as early as possible because they will take you time to implement, but not too early because you will need time to gain a full understanding of the codebase before dealing with extensive changes to it.

- Plan several alpha (internal) releases of your bot before each of the beta and final releases. It's easier for a team to work on separate components as increments off a stable alpha rather than trying to work on a component using a teammate's component that is under development. Make each alpha a real release, with full testing and documentation, as if it was your beta or final version. When the beta or final deadline approaches, you'll have little to do but submit the latest alpha. Alpha releases can help you catch release problems well before you're under the pressure of the beta and final deadlines.

- Plan at least two alpha releases at a time, rather than just the next one. At the project start and after each alpha release, make a list of possible improvements, and assign each proposed change to one of the upcoming alpha releases. Try to make the alpha releases as incremental as possible, and have many of them. Plan a learning curve for your team, assigning small modifications of a software component to early alpha releases and more involved modifications to later releases. You would be wise to make at least one alpha release during the design phase of the project.

- Use Git branches off a stable alpha to organize concurrent work among team members. It's difficult for everyone to be using each other's experimental code when developing their own. Better is to do independent development off the same stable base and then meet to merge all the changes into a new stable version.

- Don't lose sight of the big picture. Once you find one performance bottleneck, it's easy to get sucked into optimizing it to death. Recognize when you've made sufficient progress and move on to the next one.

- The static evaluator in the code base uses well-tested heuristics. Although better heuristics certainly exist, there is plenty of low-hanging fruit from simply optimizing the program before you bother coming up with your own. Keep in mind that there is a trade-off between an accurate static evaluator and a fast one. A program with a less-accurate evaluator can be stronger than a program with a more accurate one if it is faster and can therefore search deeper in the tree in a given amount of time.

- Explore ways to enable your computing resources to do useful work even while no team member is actively writing code, for example, by batching a bunch of tests to run overnight (for some students, during the day!). Write scripts and tools to automate manual processes.

Team dynamics

A four-person team can do much more work than a two-person team, but the overhead of coordination goes up substantially, since there are six pairs instead of just one. Many teams with individually brilliant members fail because they could not adopt effective procedures for working together. Here are some ideas for avoiding common pitfalls:

- Assign roles to the team members. Most organizations are more successful with a CEO, even if the CEO rotates. The primary job of the CEO is to convene meetings and decide when a discussion has gone on long enough and a decision must be made. Assign czars to manage various aspects of the project, such as someone to assemble the deliverables, a scribe for meeting notes, etc.
- Agree on ground rules, and update the team contract as you discover more issues. (Share updated contracts with the course staff.)
- Keep a project notebook, possibly as a shared Google doc. Record decisions and action items from meetings. Put the next meeting's agenda in the project notebook, and encourage team members to contribute to it before the meeting. Discuss and update the agenda at the beginning of meetings. Set up a Slack channel, but don't let that substitute for a well-organized project notebook.
- Pair program, and exploit all six possible pairings for a team of four. Divide the project into six parts, not four, and assign pairs of teammates to work on each of the six parts. That way, every teammate becomes familiar with with every other teammate and masters about half the code. Even if one teammate can't make it to a planning meeting, every piece of code has at least one expert present, enabling the team to move on.
- Teams can lose valuable time trying to isolate old bugs that were only discovered after many commits had been made, sometimes rendering the recent work useless. To minimize the risk of "breaking the build," agree and adhere to a sensible policy for testing code before it is committed to a shared version. Write infrastructure tools and scripts to assist with your methodology.

8 Rules and fine print

- You must write your own code. You may not borrow other teams' code (e.g., from the published betas) for any significant functionality in your submission, but you may take inspiration from others and borrow minor code snippets from Piazza posts. Reference materials, borrowed snippets, and inspirations should be cited in your final report.
- If you develop a useful tool or script, please share it on Piazza. You may borrow tools of any complexity to develop your bot, as long as they don't write your code for you.

- You need not use the Cilk extensions for parallelization, although you are encouraged to. You may use TBB, Pthreads, OpenMP, or any other concurrency platform. Please let course staff know if you intend to use another platform.
- As a reminder, your bot's performance will be graded relative to the staff bots, not relative to your classmates. You will receive course-contribution credit for sharing any insights or tips with your classmates on Piazza.

When in doubt, don't hesitate to ask the course staff for clarification.

9 Exhibition tournament

Just for fun, following your final submission, we will hold a live exhibition tournament at 2:30 P.M. on Friday, December 10, with prizes for the winners. It really is for fun. Your bot's performance in the tournament will not affect your grade.

In two-person games with perfect information, like chess or Leiserchess, a weaker player can sometimes beat a stronger competitor. Moreover, tournaments generally involve too few games to properly evaluate the relative strengths of players. Thus, although stronger programs are more likely to win, any program has a chance of being crowned #1 in the Leiserchess 2021 Exhibition Tournament. Come and see how you and your peers do against each other!