CHARLES E. LEISERSON: OK, let's get started. On Thursday, we are really privileged to have Jon Bentley, who is one of the masters of performance engineering, come and give us a guest lecture. In 1982, he wrote this wonderful little book from which we adapted the Bentley rules that you folks have seen.

And he's also famous for things like kd-trees. Who's seen kd-trees. trees? Yeah, so he invented kd-trees. And who's ever used the master method of recurrence solving? He invented the master method of recurrence solving. And so he's just a wonderful, wonderful fellow, with lots and lots of insights, and just a superior performance engineer.

And he's going to give a guest lecture on Thursday. And so I would encourage you to bring your friends. Bring friends maybe who dropped the class, and others who if any of you have UROPS, other people in your research group, whether they're graduate students, they're all welcome to come. Because this is really one of the opportunities you get to actually meet one of the legends of the field, if you will. And you'll see he's very approachable. He's very approachable. So anyway, my advertisement for Jon Bentley.

Good. So let's start. I want to talk about speculative parallelism for a little bit, because that's kind of what you need to make the code go fast. So if you look at code like alpha beta, it turns out that it's inherently serial. That is, if you try to do things in parallel, then you missed the cutoffs. And if you missed the cutoffs, then you start doing work that you wouldn't necessarily need to do.

And so the only way to sort of make this type of code run fast is to use speculation. Which is to say, that you're going to guess that you can do stuff in parallel and it'll be worthwhile, and then, occasionally, you're going to be wrong. But the trick is, well, how do you minimize the wasted effort when things go wrong?

So let's just take a look at a very simple example of speculative parallelism, which is thresholding a sum. So here we have a little program that is returning a boolean. It takes as input an array of size n. So thresholding a sum-- we have an array of length n and a limit. And

these are all unsigned integers, say. And I start out the sum at 0, and what I'm going to do is add up all the elements of the array. And if it exceeds the limit, then I'm going to return.

And so how might I optimize this code a little bit? Yeah.

AUDIENCE: Split up [INAUDIBLE] four processes, [INAUDIBLE] split up an array [INAUDIBLE].

CHARLES E. LEISERSON: No, let's say on processor. On one processor, what might you do?

AUDIENCE: In the middle of the loop, you can check [INAUDIBLE].

CHARLES E. LEISERSON: Yeah, once you check so that once you would exceed it, why keep adding the other numbers? So here's a code that does that-- quit early if the partial product ever exceeds the threshold. This isn't necessarily an optimization. Because notice that now in the optimization, not only do I have a memory reference and an addition, I also now have an unpredictable branch.

Actually, it's predictable branch-- predictable branch. But it's still one more step, because it's always going to predict that it's not exceeding until it actually does exceed. So that'll be pretty predictable. But still, I've added something into the loop so that maybe slower. How might I mitigate that problem?

So I don't want to add too much into the inner loop. Yeah.

AUDIENCE: It could have an inner loop that goes [INAUDIBLE] them.

CHARLES E. LEISERSON: Yeah. So basically, I can do what's called strip mining. I replace the loop of n iterations with a loop of, let's say, n over four iterations, with an inner loop of four iterations. And every fourth iteration, I check to see whether or not I've exceeded, so that the cost of the check is going to be relatively small at that point. So are there ways of making this sort of go fast.

So now, we want to make this operate in parallel. And the problem when I operate in parallel is that as I'm adding stuff up, I want to do it. So I'm going to do this here with a reducer. And so, basically, I'm going to add up the values in the reducer. But now, I'd like to do the same thing of being able to quit early.

And so the question is, well, how can we parallelize a short-circuited loop? And so, the way I'm going to do this-- and this is sort of by hand here, but it's going to give you-- so actually, as you know, underneath the loop is really a divide and conquer loop. And so I could write it as a

parallel loop. And now, it becomes, I think, a little bit clearer how I could, in this case, what I'm going to do is return the value of the sum. And now the question is, well, how can I quit early and save the work if I exceed the threshold?

Understanding that when I'm executing this and something like Cilk, it's going to tend to go down one path. And often, it's going to be a serial execution. So I'd like if it's possible.

So here's one way of doing it, which is that I add an abort flag to the code, which is going to say whether I should keep going or whether I've actually exceeded at that point. And so I'm going to use that recursively. And now, I take a look, for each time through the loop-- or through the divide and conquer-- to see whether the sum is greater than a limit. And I haven't you know already aborted the flag, then I set the abort flag to be true.

Why do I bother testing the abort flag before I set it to true? So notice that setting the abort flag is a race, isn't it-- a determinancy race. Because-- great. Thank you. It's because I have the stuff operating in parallel is all trying to set the abort flag whenever something aborts. I can have two guys who are in parallel setting the abort flag. But if they're setting it, they're setting it to the same value. So it's a benign race, assuming your memory model is such that you can actually set the values.

So what happens here when-- why do I why do you bother checking this? So what happens when several guys in parallel want to write to the same variable? This is quiz 1 stuff. Yeah.

AUDIENCE: Cache bouncing.

CHARLES E. LEISERSON: Yeah. You can have it bouncing along the cache. It will serialize it. Because if they're all trying to write, then they have to get exclusive access for it to write and modify it. And so that happens-- boom, boom, boom-- one at a time. And so by checking it first, it can be in a shared state, and then one guy clobbers it, and then it will update all the other ones. So it makes it so we don't continually have a true sharing race.

And then in checking to see if it exceeds, we basically just check to see-- we basically call this function. we set the abort flag to false, and then we return the sum of all the values. And if it aborts, then it just returns early. So it doesn't have to keep going through the computation. And, of course, once again, you can coarsen the leaves and so forth.

So this is nondeterministic code, which we should never write unless we have to. Because

that's the only way of getting performance. And you have to make sure you, of course, reset the abort flag after the use. And then, actually, do you need a memory fence here on the abort flag? Do you need to set-- where would you put an abort flag to make sure that the value was-- yeah.

AUDIENCE: Maybe do it by setting default [INAUDIBLE].

CHARLES E. LEISERSON: Yeah. So what would you have to do?

AUDIENCE: Put a [INAUDIBLE].

CHARLES E. LEISERSON: OK. So indeed, it turns out you don't need a memory fence here. And the reason is because the code is correct, whether it's the initial value false or whether it's become true. So it doesn't matter when it becomes true. And so there's an issue of, if you put in the fence, then you know that it becomes true earlier, rather than waiting for the computation. But that may actually slow things down, because memory fences are expensive.

But because the transition is always from false to true during the execution, you don't actually need a memory fence there in order to make sure that you've got the correct value. Does that makes sense? So you don't need a memory fence.

So that's a classic instance of speculative parallelism. It occurs when our program spawns some parallel work that might not be performed in a serial execution. So you're performing it, anticipating that you're probably going to need to do it. And then, typically, what you want to do is have some way of backing out if you discover that you didn't need to do it that way. Have some way of making sure that you don't do any more.

So basic rule of speculation is, don't spawn speculative work unless there's little other opportunity for parallelism and there's a good chance it will be needed. So one of the things I've seen, in research papers, no less, is people who say, oh, I'm going to have a calculation where I'm trying to find, say, the minimum of a bunch of values. And so I'm going to spawn off n things, each of which is looking for the minimum. As soon as the minimum one comes back, I'm going to retract all the other computations.

And I'm going to get super linear speed up that way. Because in the serial execution, I might have done the longer one first. And maybe the minimum is not the first one or whatever. And so they claim super linear speedup by doing speculative parallelism. But that's not really a

good example, because there was a better serial code. If that was really a good way of doing it, they should have been doing what's called dovetailing, which is doing a little bit of this computation, a little bit of this, a little bit of this, a little bit of this, et cetera, and then going back. That would have been a better algorithm for which you would then use it.

And the risk they have is that they're spawning off n things, of which most of them may not be needed, and now they don't get any speed up, even though they've just used a lot more work. And that often is a bad choice. So usually, you don't want to speculative work unless there's little other opportunity and there's a good chance it'll be needed.

My experience is, what kind of chance do you need? You need to have certainly less than-- for parallelism, if the chance that you're going to need the work-- if you have p processors, if the chance that it could be not needed is less than-- actually, I have a theorem at the end which all which I've I'll refer you to, because it's a little bit hard to say, because I didn't put on the slide. But there's a final slide for this, which gives a nice little theorem about when you should do this.

So now, let's talk about parallel alpha-beta search, because that's kind of what you're doing with your principal variation search. So if you remember the analysis done by Knuth and Morris, they basically showed that for a game tree with the branching factor b and depth d, and alpha-beta search with moves searched in best-first order examines it exactly b to the ceiling of d over 2 plus b to the floor of d over 2 minus 1 nodes at ply d. So that's basically square rooting the amount of work that you would need if you did a full depth ply. That's with alpha-beta.

So naive algorithm looks at b to the d nodes at depth d. And so for the same work, the search depth has effectively doubled, or the work is square rooted. So that we solved last time in Helen's lecture.

So the key optimization here is pruning the game tree. And as I mentioned at the beginning of this, when you prune the game tree, you've got what's effectively a serial thing. And if you let something go ahead, you may be working on something that would be pruned.

So then how does the parallelism really help you there? You're just wasting processors that could be better spent perhaps elsewhere. Except, where else can you spend it? Because there's no other parallelism in the code. So we want to find some solution.

So the solution comes from an idea from a very nice paper by Burkhard Monien and some of his students in Germany. And they made the observation that in a best-ordered tree, the degree of every node is either 1 or maximal. This is not their observation, this is actually the observation in the Knuth paper. So when you have a best-ordered tree, if you can get all the moves ordered in their true order, then it turns out that either you're exploring all the children or you are refuting something and you get a cutoff from the very first one that you look at.

And so in this case, for example, it turns out, on the principal variation, those are all full-width-- sorry, you have to explore all the children. And then from there on it alternates. One level, you have just a single child that needs to be explored. And when you explore it, if it's best-ordered, you will get a beta cutoff for the others. And then it alternates, and then it's full-width.

And so their idea was to say, well, if the first child fails to generate a beta cutoff, speculate that in fact you have the best one, and the remaining children can be searched in parallel without wasting any work. So if it fails, you're going to say, oh, I'm going to speculate that this is in fact a full-width thing. Now, in practice, you don't necessarily get things best-ordered, but there are a bunch of heuristics in the code we've given you in chess code to make it so that you tend to do things in the proper order.

The most important of those is, if you've seen the movie before and it's in the transposition table, you use the move that you've seen before, even if it's to a shallower depth. You guess that that's going to be their best move still, even if you search deeper. And that's pretty good.

And so they call that the young siblings weight algorithm. They actually called it the young brothers weight, but in the modern era, we call it young siblings weight. And we abort subcomputations that proved to be unnecessary. So you're going to start out searching, but then you want to get rid of things that you discover, oops, I did get a cutoff from searching from one of these things. I don't need to do this. Let's not have it keep spawning and generating work and some other thing, let's abort it.

And here's the idea for the abort mechanism. So what you do is-- the abort can occur at any given node. You get a beta cutoff-- you want to abort all the children that wouldn't have been searched anyway. But they may have already been spawned off.

So they have a record in each node that tells you whether or not you've aborted. And what you do is you just simply climb up the tree to see, periodically, whether or not you have an ancestor who is aborted. If you have an ancestor that's aborted, it says I'm aborting the side

computations, then you say, oh, I'm done, and I just have to return. And so you check that on a regular basis.

So do people follow what that mechanism is? And so, that's basically what you're going to be implementing for the parallelization, is this thing of climbing up. You're going to guess, after you've searched one child, that it's good. Hey, actually some people say, why don't you check two before you search the others, so that you're even more sure that you don't have a cutoff, because neither of the first two aborted. Because, in practice, of course, the game tree is not best-ordered. And so you're going to waste some work.

But the idea is, you're going to pull up the tree to see whether or not-- and, of course, you don't want to pull on every evaluation that you do, you want to just pull frequently. So you may have a counter in there that says, OK, every 10th time, I'm going to pull up the tree. You put a voodoo parameter there that says how often it makes sense to actually check, because there's cost to going up and checking versus exploring more of the tree.

And so, I have an example here. And I think this is where I'm going to cut this short. So there's an example, which I suggest you take a look at. I want to, as I say, spend time doing Q&A, and talking about the other parts of the program, and give you folks some good ideas for the thing.

So this is sort of the theory thing, and then I have some slides that will explain this with examples in more depth. Because I don't expect that everybody got every detail of this. So I have some examples and so forth in here that I'm going to let you guys look at on your own.

Now, if you parallelization the spawning off loop of the younger siblings, you will get a code with races. And the reason you get races is because there are several-- the are three data structures-- that the search is employing that are kind of global. The first is the transposition table-- looking things up to see whether or not you've seen them before. That's a major optimization. You don't want to get rid of the transposition table.

And so you have a choice with the transposition table. What are you going to do with it? Are you going to lock it so that the races become-- at least your data-- race-free and updated atomically. Or you could replicate it. For example, you could have a worker local copy of the data structure, and each worker that is working on it only accesses their own local copies. Or you can make a copy when things are stolen or you can make just one where you maintain it locally.

Or you can decide that, well, even if there's a race there, the odds that it's going to affect how I play in the game are not that high, so I'll run with the race. Because any other solution is going to be more expensive. And then maybe you get funny answers.

So one thing for that kind of data structure, let me just recommend that you have some way of turning of the data structure so that you can actually do the parallel search and get deterministic, repeatable results. So even though it may be aborting and some things may not be aborting, you want to have some way of executing. And so, for example, you want to be able to turn off even the speculation in your code, so you can test all the other things in your code that don't depend on the speculation.

Because if you have something that's not deterministic, as I say, it's a nightmare to debug. So you're going to do the evaluation function. Hey, if you're testing it, turn off the speculation. You should be able to find out whether you've got bugs in your evaluation function. There's no point in discovering you of a bug, and it's like, well, where did it come from or whatever. So you want to have things you're turning out. And also, want to have ways of turning off, for example, access to the transposition table. Yes, I'm going to speculate, but no, I'm not going to access the transposition table in parallel, because I may get a race on the entry in the transposition table.

The hint that I will give you for that is that in the code for our program that took second prize in the world computer chess championship many years ago-- I think it was in 1999-- where we actually almost won the tournament, and we lost in the playoff. We would have won the tournament, except that our programmer suggested a rule change at the beginning of the tournament that everybody agreed to, and then we were on the short end of that. Otherwise, we would have been world champions.

And that was the last competition against computers that Deep Blue, the IBM computer that beat Kasparov, the human [INAUDIBLE], just a few months later. They performed it. And they placed third in the tournament. We tied for first. Our only loss was to Deep Blue. And we were running on an 1,824 node supercomputer at Sandia National Labs, a computer that probably is today less powerful than this.

But it was very big type of computation. They basically said, if there's a tie, they're going to base who wins on strength of opponents. So you take a look at your opponent's records, and if you had stronger opponents, you'll win against somebody. And we said, you know, if there's a

tie just between two programs, we should really have them face-off against each other as an extra round. And everybody said, yeah, that's a good idea. And then, wouldn't you know, at the end, we had the strongest opponents, and we were tied with another program called Fritz-- an excellent program.

And we were outsearching Fritz in the playoff game. And then we saw a move that afterwards, when we were able to do offline searches deeper, we were outsearching them by like 2 ply. But then there's a move that looks like it's a good move for the depth we were looking at it-- doesn't look like a good move if you search much deeper, or frankly, if you search much shallower. It was one of these things where it just looked like a good move for the ply we happened to be searching.

Because there's no guarantee-- because you can't see the end of the game-- that if you search deeper you're actually going to beat somebody else. Because there's the horizon effect of you just simply can't see what's coming up in the future. You can only see sort of on average.

So we made this bad move. We almost recovered. We almost tied. Had we tied, we would have been world champions, because we had the stronger-- we won the tiebreaker. And we weren't able to recover from the error. And so we took second prize. It was like--

[LAUGHTER]

Anyway, in that program, we decided that we were just going to let there be a race on the transposition table. And the reason was, we calculated, what are the odds that the race actually affects the value that you would actually pick that value? And if it affected the value, what are the odds that it affects the move that you're going to make? And if it affects the move you're going to make, what are the odds it affects the game? And if it affects the game, what are the odds that affects your result in the tournament?

And when you've figured all these things out, it was like, eh, we would slow the program down more by putting in, for example, locking-- because that would slow down every access-- than we would if we just ran the thing. Because one of the things that happens in alpha-beta is if you get an extreme value, usually those are lopped off. Because if it's such a good move for you, your opponent's not going to let you play it. So if you ended up with a score that was extreme because it was based on a value that you were racing on, usually it's going to not even propagate up to the root of the tree. So anyway, we just ran naked, so to speak.

So there are two other data structures you going to have to worry about to make decision about. Another one is the killer data structure. So the killer data structure is a heuristic that says, at a given depth of code, you tend to see the same moves that are good. And a move that is good for one thing at a given depth, tends to be good for something else.

So for example, it may be that you play bishop takes queen. So you've won the other players queen. And then their response is to do something irrelevant. Well, now you mate them. Well, there's a lot of other moves where you could make them on that move, for which they're playing things on the board that are irrelevant. and. So the same type of move tends to be a killer-- always, you're able to, at that depth in that position, make the same move. And if they don't address the issue, that's always a good move to check.

And so that ends up being ordered near the front. So the killer table keeps track of that. And I think, in our code, we have two killers. Is that right, Helen? I think it's set up to allow you to do for up to four killers, but we only do two in the code that we gave you. And you can enable it and see whether four killers-- but that's a shared data structure. And so one of the questions is, should you be keeping a local copy of that or should you be you using a global one that they all share and updating it?

The third one is the best move table, which is used to order all the low order thing. So the first thing that's most important is, did you get a move out of the transposition table? That tends to be quite accurate. And the second is, did you get a move out of the killer table? And finally, it's, statistically, how good are these moves?

Have you seen these a lot before? If you've seen them a lot before, that's how the other moves get ordered. And that's kept in the best move table. And once again, that's a shared table. In the search, you're going to have to figure out, do you want to do a thread worker local copy, or do you want to synchronize it, or how are you going to manage that data structure?

And the answer, I would say, is different compared to what you do with a transposition table. The transposition table, generally, it's not worth locking or whatever. And it is good to share. Because if you have seen that move before, that saved you a huge amount of computation to be able to look it up and not do it. So those are some of the tips for parallelization, which we'll get to in the beta 2.

But now, I want to talk about, for most of the rest of the time, some of the other things that you

can do with the code you've currently got. And let me take these in some order, and then we can ask questions. So opening-- who's contemplating doing an opening book? Anybody? For beta 1? Are you going to do an opening book for beta 1 or beta 2?

For beta 1, let me see. OK. Beta 2? Who wants to do an opening book better 2? Final? OK. Yeah, OK.

So the idea here is to precompute best moves at the beginning of the game. So, well, we know what the starting position is. So why don't I spend $1,000 or whatever on Amazon, and compute things to some ungodly ply, and see what the best moves are at the beginning of the game, and put that into a book so I don't have to search those? So that's the idea.

I think, to begin with, there are lower hanging fruit than the opening book. If you look at it, you're going to be an opening book, if you're lucky, for six or eight moves. And the games tend to last-- have you looked to see how long games tend to last? What is it?

AUDIENCE: [INAUDIBLE]

CHARLES E. LEISERSON: No, most games don't go 4,096. We don't let them go that long anyway. So did anybody take a look, statistically, to see how long games are? I think they tend to be like 40 or 50 moves. I mean, this year is different from other years, because we have different rules. But I think it's like 40 or 50 moves.

So you're not spending-- you're doing something that will help you in 10% of the game. Whereas there are other places you could do it which are going to help you in the whole game. Nevertheless, we've had teams that did a great job on opening books and clobbered people by having a fantastic opening book.

It's actually cheaper to keep separate opening books for each side than to keep one opening book for both sides. And in this game, it's actually fairly easy to keep a symmetry thing and basically have one opening book that works for the side on move. And that allows you to store. You don't have to store-- if I make a given move, if I have a given position, I only need, in principle, to store one answer for that position. Whereas, then, my opponent may make any number of moves for which then I only need to know one move-- what's my best move in that position.

So you can see that you have, once again, this property, that on my move, I only need to have one move stored. My opponent's moves, I need to have all the moves stored. And then my

move, I have one move store. And that basically means that you're effectively going twice the depth, as if you kept all my moves, and then all of the other opponents moves, and then all of those.

Because when I do that, it's like, well, why do I need to know the best move. So it's better to keep them separate. When you build the opening book, you'll store a lot less. You'll have a lot smaller storage. You'll be able to store a lot more moves if you do that.

But I would say, not necessarily the first thing I would go to optimize. But it's also the case that this is a place where one of the teams built a fabulous opening book one year. And one of the things about the opening book is it allows you also to sort of threshold and say, in a given move, I don't just have to have one move. Let me have three moves and pick randomly among them, so that I'm not predictable.

What the team did that used the opening book is they observed that everybody was searching from the beginning. So they were all finding exactly the same moves at the beginning of the game. And so they knew exactly what the path was that all the other programs were going to follow. And so they could make the best move on their move.

But if you have something that's got some randomness to it-- and there is a jitter in there. Who found the randomness-- the place we insert randomness in the code? So I think Zach had a comment on Piazza about that. So there's a place in the code where we dither the amount so that you will get unpredictable results. And that way, you won't always play the same move, just by changing things.

Because a hundredth of a pawn value is not very big. And who cares whether or not we have-- these heuristics aren't measuring things so closely that we care about a hundredth of a pawn. So you can dither things and have one move be better than another and not really affect the quality of the playing.

So it is a good idea to do something to not be so predictable. But to build a full opening book, that's a lot of work for beta 1. For beta 2 and the final, yeah, you'll get to that point.

The next thing is iterative deepening. Do people understand how this part of the search code works? Yeah? No? Let me go over it.

So you can imagine, say, I'm going to search to depth d. And in fact, instead of just searching

depth d, we do a depth 1 search. If you look at the code there, there's a depth 1 search. And then we do a depth 2 search. Then we do a depth 3 search. Then we do a depth 4 search. And we keep doing that until our time control expires.

Why is that a good strategy, generally? Well, first of all, the amount of work with each depth is growing exponentially. So you're only spending, in principle, a constant factor more work than you would if you searched to depth d right off the bat. But the important thing is that you can keep move ordering information as you search deeper. And the move ordering information, remember, we want our searchers to use best-first move ordering so that we get all the cutoffs we can possibly get for pruning.

And so by searching it easier, actually you end up with better move ordering. Because when you find that same position in the transposition table, you say, oh, well. I didn't search it as deep as I need to search it now. So I can't use the value that's in the transposition table for the position, because maybe I've searched something that's ply 8 in the transposition table, but I've got it searched to ply 9-- not good enough a search. But the value-- the move that you've found at 8-- that's probably a pretty good first move-- pretty good guess at best move.

So by doing iterative deepening, you actually get the move ordering for all those intermediate positions really really, really strongly. Because that transposition table is the very best information you've got for what the your best move is. And also, as I mentioned, is a good mechanism for time control. Is that clear?

So that's why you bother. It's one of these things that looks like it's redundant. Why are you searching-- if you're going to search to depth d, why search to depth d minus 1? You're going to that as part of depth d. Well, no, because you're going to get move ordering information that's going to be really valuable when you search to depth d minus 1. So when you search to depth d, you've got much better pruning than if you just went straight to depth d and had no information about the move ordering. Does that makes sense?

Endgame database-- so here's the idea. If there's a few enough pieces on the board, you can precompute the outcomes and store them in a database. So for example, the most common database to come up with is the king versus king database. So if you do King versus King, that means that it's not that you expect the game will end up as king versus king, it's that you're going to encounter that in the search. It'd be nice to know who has the win if you're King versus King.

Now, the code actually plays optimally for king versus king. The code we gave you will play the king versus king perfectly well. And, in fact, there's just two heuristics that are responsible for that. One is called k aggressive heuristic, which basically makes you move towards your opponent. And the other one is the k face heuristic, which makes you point towards your opponent. And those two things turn out to do a pretty good job of when there's one there's two kings and playing, they go through this little endgame.

Did everybody do that by hand, by the way? Did you play king versus king? A really good idea if you want to understand the game is just go and do a king versus king version of the game. Get an eight by eight chessboard and just put down two tokens kings that have orientations, and then just you and a friend, just try to see if one can mate the other. And you'll see that it goes through a very ritualistic type of behavior which ends up with one of the Kings being mated. So it's nice to know which one it's going to be so if you encounter that.

But here's the question, how many games actually make it to an endgame? Once again, you have to look at what the odds are there. Because if games are ending in the middle game, there may not be any need to search all the way to the endgame. So that's something that you need to weigh.

Now, if you do an endgame database, it doesn't suffice to just store win, loss, or draw for a position. And the reason is because you can get into a situation where you've got a winning position, and then you say, OK, well, let me move to another winning position, and another winning position, and another winning position. Oh, and I'm back at my original winning position.

And I just keep going around in a circle, always in a winning position, never moving towards the win. And that, of course, will end up in a draw. So you need to know which direction do you go for the winning position.

So the typical thing you do there is you keep the distance to mate to avoid cycling. So instead of keeping just a boolean value, you say, here's my distance to winning. My distance to winning is 6. Now, when I'm searching, I don't want to go a distance to winning that 7 or 6, I want to go to distance to winning that's 5 when I make my move. So it's very important to make sure that you maintain the distance in order to avoid cycling.

Quiescence search-- we talk about quiescence, which is to make sure that when you're doing the evaluation you're not in the middle of an exchange. So you zap their pawn, they're going to

zap your pawn, or maybe even zap your King. And you don't want evaluate in the middle there. And so the idea of quiescence is that when you're done with the regular search, you just play off moves that involve captures, and then you evaluate the position. So that's quieting the position. Make sense?

Another heuristic there is null-move pruning. In most positions, there's always something better to do than nothing. So sitting still is usually not as good as actually making a move. And so, the idea is, suppose that I can forfeit my move, and search a shallower depth, and I still have the best position-- then I still am winning. And it generates a cutoff. Then why bother exploring any other moves? I probably am in a really good position here. OK And if I don't manage to get a cutoff from the null-move, then I do the full depth search.

So the place this is dangerous is in Zugzwang. In chess, it's called Zugzwang situations. So Zugzwang is a situation where everything is fine, but if you have to make a move, you lose. So usually, having the advantage of a move, that's called the initiative in chess, and that's a good thing. You want the initiative. You want to have the extra move over your opponent.

But there are some situations where if you move, you lose. And so you want to make sure you don't have a Zugzwang situation. I don't actually know if there's a Zugzwang situation in Leiserchess. So if somebody comes up with one, I'd be interested to see that, where if I had to move, I lose-- but if I sit there. So in chess, what they do, is in the end game-- that's when Zugzwangs come up-- they turn off the null-move pruning.

So is this clear what null-move does for you? What's going on there? I see some people who are quizzical. Anybody want to ask a question? I see some people with sort of quizzical looks on their faces.

So the idea is for null-move, my position is so good, that even if I do nothing, I still win. So I don't want to search anymore in this part of it, because I get a beta cutoff, you're not going to let me make this move-- the move that got me here. Yeah, question?

AUDIENCE: That was it.

CHARLES E. LEISERSON: OK. That was it. OK, good. My move is so good, I can do nothing, and I still win. And so why bother? Let me verify that without necessarily doing as deep a search. So null-move is fairly cheap to do, and it often results, in a lot of positions, for cutoff. And if I don't get a cutoff, then I just do the normal search.

This is a pretty complicated piece of code, isn't it? Pretty complicated.

There are other situations. We mentioned killers. There's also move extensions. So typical move extensions that people look at is you grant an extra ply in chess if the king is in check, or for certain captures, or if you have a forced move. So suppose you have a move and it's the only move you can make, then don't count that as ply. So you can search deeper along lines where there are forced moves.

That's what they do in chess. In Leiserchess, not quite so clear what you do-- which ones you would do extensions for. Because it's rare that you have just one move in Leiserchess, compared to an in chess. By force move, it's like, if you don't do this, you're going to get captured, or mated, or what have you. So that may come up in Leiserchess. But anyway, it's something to think about.

So the transposition table, we talk about this as a search tree, but it's really a dag, because I can get to the same position by transposing moves. This guy does a, this guy does b, this guy just c, this guy does d. It's the same thing by doing a, d, c, b. I transpose those two moves, and I get to exactly the same position. And so I'd like not to search that position again if I've seen it. And that's what the transposition does.

There's a quality score that is in the transposition table that tells you how good to move you've made. And the quality is essentially the depth that you had to search in order to establish the value that stored in the transposition table. And so you don't want to use something of too low quality when you're searching. If you have to search the depth d, something of quality d minus 1 is not good enough. Because, otherwise, you'll not be searching the full tree.

But something typically that is deeper-- if I'm looking at depth d and I find something in the transposition table that's d plus 1 or d plus 2, that's great to use. That just gave me an even deeper view of what's behind that move. And so if you look at the logic there, you'll see that that's how they do it.

It's very tricky. One of the things that's tricky is when you find a mate, how you store that in the transposition table. And you'll see, there's special code for handling mate positions. And the reason is because what you're interested in doing when you find a mate is knowing your distance to mate. But not from that position-- the distance to mate from the root of your search.

So, for example, let's say this is the root of my search, and I search down to a given point. And now, I do a lookup in the transposition table, and I discover that there's a mate in 5 here. And this, let's say I've searched 9 ply or something. Well, if I store that this is a mate in 5, and I store the value for mate in 5, then if it makes it up to the top here, it thinks there's a mate and 5. But there isn't a mate in 5, there's a mate in 14.

So when you look it up and you discover there's a mate and 5 in the table, you have to do a little bit of a calculation to translate that into being a mate in 14 as the value that's going to be used here, rather than a mate in 5. Does that makes sense? So you'll see, that's the logic that's in there for dealing with mates. So a mate basically takes a very large number-- way larger than all the material on the board-- and then it subtracts what your number of ply is to get to mate.

So some big number-- I don't know, 32,000 or something-- minus the depth to mate is how you represent a mate. So it's some very, very big number and then minus the depth. And that way, you can make sure that you're going for a mate in 13 instead of a mate in 14, for example, preferring that you take the shorter path. Makes sense?

So that's one of the things to look at in there. And there's also a lot of stuff in there. The transposition table has-- we talked about caching-- it's actually a k-way associative cache. And so you can vary k to see what's the best choice of how many entries you should have. The more entries you have, the longer it'll take you to search them. On the other hand, the more likely it is that good move stay in the cache. So you can decide what's the right trade-off there. So there's a good tuning optimization there.

Questions? Any questions about transposition table? Transposition table is a major optimization.

So Zobrist hashing-- so most of these, Helen talked about at some level. But I wanted to give a chance to have people ask questions. Nobody's asking questions. I came here for Q&A. All you're getting is A.

[LAUGHTER]

So let me explain Zobrist hashing again a little bit. So Zobrist hashing-- very clever idea. So we have our board with a bunch of pieces on it. That's probably the wrong number of squares, right? That's seven by six. OK, who cares.

So the idea is that what I'm going to do is have a table of random numbers that I'm going to compute in advance. And this is going to be indexed by my row, my column, my piece type, and my orientation. And every different combination of row, column, type, and orientation corresponds to a different random number in the table. So we have some sort of random number there.

And what my hash function is is it's the XOR of all of the values of all the pieces that are on this table with their orientations. OK so if I have a king here, that's a white king-- I guess I need to know not just what the type is, I also need to know whether it's white or black-- so the side. I think, actually, that gets encoded somehow.

But in any case, I think maybe in the type we actually keep whether it's a white pawn or black pawn. Yeah, it's in the type, I think, is the way we actually implemented that. So there's white king, black king, white pawn, black pawn, or space.

So if I have a king there, that corresponds to a certain value. If I end up with a pawn here-- let's say, a white pawn-- then that will be another one, and I XOR those together. And I take the black king and I XOR his position is, and the black pawn, XOR that in. So I XOR all these four values. That's the hash function that I use to do things like look up things in the transposition table.

Now, if I had to compute this every time I changed the position, that's if I have a lot of pieces-- how many pieces I've got, 7, 8, 16 pieces? Each side has seven pawns and a King. So 16 pieces-- I have to do 16 XORs in order to compute my hash function.

So Zobrist hashing is really clever. It takes advantage of that XOR trick that I taught you in the first-- no, it wasn't the first, it was in the second lecture? Yeah, the bit tricks lecture. And that is that XOR is its own inverse. So if I want to remove a piece-- let's say I'm going to move this pawn from here to here.

So I remove this piece. What do I do to my hash function to remove a piece? I look up the value for that pawn in the hash table and I XOR that into my hash for the table. And I now have a hash for those three pieces left. Does that make sense?

So I have my hash function, which is a hash of the four things. I'm not sure you guys can see very well there. And I simply XOR it with the hash of the pawn that I removed in this case. And

now, I moved it to here. So what do I do? I look up that one and I XOR that value in here for the new position-- the new pawn position-- and that gives me, now, the hash for the new table with the pawn in that position.

So any move that I make, I can basically update my hash function with only two XORs. And XORs are very fast. They're one instruction. And they can do lots of XORs and stuff. So this is actually a very, very cheap thing to do-- a very cheap thing.

So that Zobrist hashing, that you can keep these things up to date. And that's something we implemented for you. That's an optimization that was a freebie. We could have given you the hash function like this and had you implement that, but this is one of the ones we gave you as a freebie for optimization. You're not all saying, thank you very much, Professor Leiserson?

[LAUGHTER]

No, in some sense I took away an opportunity for optimization, didn't I, there?

And so in the transposition table, there are records for the Zobrist key, the score, the move, the quality, also a bound type, whether it's upper, lower, or exact. Because when I return something from alpha-beta, I only know a bound on it, if it's greater than alpha or less than beta. And in some sense, the age of how old is this move. Because as things get older, I also want to age them out of the table. There are several aging things there. And you'll see the best move table also has an aging process, whereas every time it updates the values and gives a new value, it ages all the other values so that they gradually disappear and aren't relevant.

One of the ones that people get confused about is the Late Move Reductions-- so-called LMR. This is the situation where I'm going to do, let's say, my parallel search of all my moves. And the question is-- once again, you're trying to prune everything you can. And so the idea is, which are the moves that are more important to search deeper? The ones near the beginning of your move list, if it's sorted in best-first order? Or the ones towards the end of your move list?

So where is it most important to search deeply? For things that you think are possibly the best move or the ones that you think are the worst moves? Yeah.

AUDIENCE: Search for the best move.

CHARLES E. LEISERSON: Yeah, it makes sense to search the best moves, right? So the idea is, well, if something is way down on the move ordering list, why search it as deeply? It's probably not as good a move. And so, let me search it more shallowly? I probably don't lose much of an opportunity to discover that that's actually is indeed a bad move. There's a reason that got it ordered down there. And so that's a late move reduction.

So with a good move ordering, a beta cutoff will either occur right away or not at all. So you search the first few moves normally, and then you start reducing the depth for moves. I believe, in our code, we have two numbers where we reduce by depth 1 after a certain number of moves and reduce by depth 2 after a certain number of other moves.

Those are things that you can tune. I wouldn't tune them very much. I don't think. And once again, I could probably be wrong here, and someone will discover, oh, if you tune it like this, it's way better. But that's the idea of the late move reductions.

Probably one of the most important things to think about is the representation of the board. Right now, we represent the board as it is. That's a terrible representation. It's very time-consuming.

There's sort of two major ways you can do it. Oh, I didn't put the other one here. Well, anyway, one of them is bitboards. Here, you use a 64-bit to represent, for example, where all the pawns are on the 64 squares of the board. And then you can use POPCOUNT and other bit tricks to do move generation and to implement other chess concepts.

So if you're looking to see what are the possible places a pawn can move, and you want to say, can they move right? And let's say it's stored in row major order, you can just do a right shift by 1, and that tells you where all the places that those pawns could move. And now, you can just pick them off one bit at a time to generate your move list.

And then you can do it that way. If you're going to move up a thing, well, then you're actually doing a shift by or down. You're doing shift by how much? By eight, to move up or down, if you're storing things in row major. That makes sense, right?

So if it's 8 by 8, and you're keeping a bit for each thing, then if I want to generate where is this one? If I shift this whole thing stored in row major order by 8, if I shift it right, it basically puts it there. So I'm moving by 7, 8, and 9, that gives you-- and then shifting it by 1 or minus 1 gives you this, or left by 1. And then, similarly, you can do by shifting by 7, 8, or 9 that way. And I

can generate all the possible moves. So that's one way of doing move generation is using bitboard.

And there are a lot of things, for example, that you can do with bitboards in parallel. Because you can say, did I make a capture here? Or let me use a bitboard to represent where the laser goes. Did I make a move that is going to affect the path of laser?

Because one of the major optimizations you can do is in the evaluations in dealing with the laser. Because you're spending a lot of time stroking out laser positions. What's the point of doing that-- if you made a move of something and it didn't affect where the laser goes why bother doing it out? You can just cache what the value of the laser is. And there are a lot more good stuff on the chess programming wiki. So, yeah, question?

AUDIENCE: How do you [INAUDIBLE] a right shift when the [INAUDIBLE] if the shift by 8, and it's clear when it falls off [INAUDIBLE].

CHARLES E. LEISERSON: Yeah. You got be careful there, right? Because if I do a shift to the right, for example, what'll I do? I'll just do a mask to eliminate all the things that got wrapped around. So two instructions or whatever. Yeah, details. Yes. Details are good though. Good question. Good question.

Whose timed their program? Where are the opportunities that you see for performance engineering for a first pass? What are the expensive things? What do you have as an expensive thing?

AUDIENCE: Add laser paths-- [INAUDIBLE]

CHARLES E. LEISERSON: Sorry the?

AUDIENCE: Marking the laser path.

CHARLES E. LEISERSON: Marking laser path, yep. Good. What else is time expensive?

AUDIENCE: Laser coverage. [INAUDIBLE]

CHARLES E. LEISERSON: Yeah, laser coverage. Boy, that is really expensive. That is really expensive. So where else is expensive? What else is expensive? So I would definitely go after l cover. That's a huge one to go after.

One of the things, by the way, if you are making your changes to the code and you're going to change the representation, leave the old representation there. Take it out at the end. Add the new representation and put in assertions that say that things are equivalent or whatever. But don't get rid of the old stuff, because you'll end up with broken code. And definitely, used things like perft to tell you whether or not you made any change If you touch anything with move generation.

So where else is expensive? Yeah.

AUDIENCE: Can you explain the laser [INAUDIBLE]?

CHARLES E. LEISERSON: Sure. How much detail do you want? How it actually works?

AUDIENCE: [INAUDIBLE]

CHARLES E. LEISERSON: OK. So what is supposed to do is figure out how safe-- the idea is, I want my laser to get closer to your king, and I want your laser not to be close to my king. And if I can move into positions where my laser is closer to your king but your laser doesn't get closer to my King, that's a good sort of thing.

But when we say get the laser close, what happens when I've got, say-- let me do it this way-- a position like this. So here's the-- OK. Suppose I look at this position. How close does the laser get?

Let's say I'm black here, and I look at the laser. Well, the path of laser is it bounces there and goes across. So I didn't get very close to the king there, but I'm one move away from getting it pretty close. Because if I just move this guy out of the way, now I've got it really quite close.

So if I compare that to, let's say, a situation where instead of the pawns are there, let's say a pawn is here. Now, the laser is actually closer to the King than it was in the first position, but it's a much worse position. The first one was much better when I had the pawns here and here, because I was simply one move away from getting it really close. And so if you use just the direct laser thing-- and we did some tests on this-- this turns out to be not very-- it doesn't guide the program very well on getting into situations where my laser is getting close to your king. Does that make sense?

So then we said, well, how should we measure how close the laser gets? So what we said is, well, let's take a look at all the possible moves from here of one move and then look to see how close we get things. And the way we did that is we said-- actually, Helen and I worked on this really hard, because this is a new heuristic that we have not used in previous years. And it works great, it's just slow as anything.

But it works well, so you have to evaluate, is it worth doing something if it's really slow? It may be that you'd do better to use a simpler heuristic and get deeper search than it is spending a lot of time evaluating. But anyway, we gave you one, that if you can make it go fast, should be really good.

So the idea is that what we do is we look at all the different paths from moving any one piece and look at the paths of the laser. And what we do is we go we count 1 for every position that we go away-- 2, 3, 4, 5, 6, 7, et cetera. We actually add an extra value if we bounce off an opponent's-- how much do we add, Helen, do we add 1 or 2 if we bounce off an opponent?

AUDIENCE: 2.

CHARLES E. LEISERSON: I think, 2. Yeah. So if this is an opponent's pawn, we would basically go from 3 to 5, 6, 7, 8, 9. And we basically do that for all the moves. And the way we combine them is we take the minimum number that I can get there. What's the what's the cheapest way I can get to every square that the laser goes?

So we take all the different moves, we stroke them out, and we take the minimum. So if I have another path, let's say by turning the king it will go this way, and there's a you know there's another one here that's my pawn, then I may get there in 1, 2, 3, 4, 5, 6 7. And so then this would become a value of 7 here. so that's the first thing is we basically get a map of how close are things.

And the second thing we do is say, well, how valuable is it to have these to be near the particular king? And then what we ended up with is something where if I look at the distance that I am away-- let's say, this one, for example, is one row and one column away, this is 0 row 0 column-- we use, I think, it's 1 over the row plus 1, times 1 over the column plus 1, as a multiplier to weight how good it is that we've been close to the king. And we invert these values so that it's better to have a smaller number there, and then we add them up.

We don't quite add them up. No, I'm sorry. What we do is we look at these things as a fraction

of my shortest path distance. Sorry, that's what we did. Yes, I'm sorry. That was a previous heuristic.

So here, for example-- let's say this is the 9-- the shortest way of getting there is 1, 2, 3 4, 5, 6 7, 8. So this would actually, when we do the conversion, would give a value of 8/9 for being in that square. So we didn't get it. Whereas this one, the shortest path distance is 7 because of this path, and you can get there in 7. This would have a value of 1. So this is better.

And so we do that for all the squares. So we get a fraction of how much do we do. And then we wait it by something like this, which falls away quickly. But it's important, in heuristics like this, to have a smooth path, so that you can get things closer. If I made all these things be 0, it wouldn't know that it could move up and get a little bit better in the second or third order digit to get closer.

And so then we add it all up, and that ends up being a number, we discovered, that's sort of in the range of-- what did we figure the range was there? It was up to like 4, or so, right? Something like 4 would be the maximum value it would be. And then we said, OK, then we have this magic constant multiplier that you can play with, that says, OK, let's represent that as a fraction of a pawn. How much is that worth? And we multiplied by that value.

So that's what we're doing. So that it makes it so that we do tend to move our laser closer, we subtract how close the opponent can get from us, and then we say that's the evaluation. Now, if you have a better way of determining whether a laser is getting close than this one, that's cheaper to compute, that's good.

For first pass, I would just simply try to make this go fast. Because for many of the moves that you're going to look at, it's going to be moving this pawn to there. Nothing changed. There's no point in stroking this out, and calculating all the minimum, et cetera, because it didn't touch the laser path. Things that are going to touch a laser path are things that you move on or off the path. So why bother? And so if there's a clever way of caching it so that you can do things, that's good too.

Any questions about other things that could be done? So another place, I'll tell you, that's a good idea to look at is-- especially once you get this heuristic top rate a little bit faster-- is the sorting of moves is actually pretty expensive. Once you figure out, here all the moves-- there are a lot of moves-- now you go through and you do a sort. And if you think about it, in a best move order tree, sometimes you only explore one of those. So why'd you bother sorting all of

them? On the other hand, sometimes you do need to sort all of them. So how you optimize that sort so that you don't waste time sorting stuff that you're not going to actually ever explore, that's another opportunity for optimization.

Yeah, question?

AUDIENCE: How do you sort the moves [INAUDIBLE]?

CHARLES E. LEISERSON: The moves are sorted by these things like-- there's a sort key in there that represents a whole bunch of things, like whether it was a killer. If it's a killer or it's a transposition table value, it's a very big value. If it's a statistical thing that the history table says, historically, this has been a pretty good move, then you get another value. And then exchanges get ordered and so forth. So there's a bunch of things that are where you end up with information, and then it sorts those things.

So that's actually kind of complicated logic in terms of what the actual values that are used there. But the idea is, here all the moves, now let's figure out what's our best guess as to what they are. Most of the good moves are coming out of the transposition table. But sometimes, the transition table it says it's not a very good move, and there may be a better move. So the quality of what you get out of transposition table is good, but that doesn't mean that that's always the best move.

AUDIENCE: [INAUDIBLE]

CHARLES E. LEISERSON: Yeah. Any other questions about what should be worked on? Let me just make sure I-- OK, go ahead.

AUDIENCE: Where does that sorting happen? I've never [INAUDIBLE].

CHARLES E. LEISERSON: Where does the sorting happen? That happens, I think, at the move generation. Is that where it is? Is it stored in the move generation?

AUDIENCE: In search.

CHARLES E. LEISERSON: Or is it in search? I think it's in search. I think you're right. Search calls move generation and then sorts it. I think that's right. Yeah, question?

AUDIENCE: So one of the things we did that we were providing [INAUDIBLE], we have a very tiny board implementation.

CHARLES E. LEISERSON: A very tiny board implementation.

AUDIENCE: Yeah, [INAUDIBLE]. You can do a lot of matrix on it as well. But it's like maybe [INAUDIBLE] replace it everywhere.

CHARLES E. LEISERSON: Yes.

AUDIENCE: So what's a good strategy?

CHARLES E. LEISERSON: Well, as I say, keep the old one around.

AUDIENCE: [INAUDIBLE] still use [INAUDIBLE].

CHARLES E. LEISERSON: Well, you use the old one while you're still getting the new one right. It's also good to be able to go from old representation to new representation and having conversion functions. But, yeah, making representation changes, painful, painful. And boy, if there was one tool I would love that would automate stuff like that, that would be it, to be able to change representations of things and still have things go well.

I should have mentioned, by the way, the other representation besides bitboard. Another one is just to keep a list of the pieces and their positions. Because that's going to be smaller than keeping this great big board. And do you keep this list sorted or do you not keep it sorted? But the advantage of that is, particularly as the game goes on, that list gets shorter and shorter. And so if you're doing less in manipulating the board position, that's generally a good thing.

AUDIENCE: So for the board reputation, my recommendation is to refactor all the board axes into a function. And then you would still use the old representation in the function. Then you can validate that you refactor everything correctly. And then you can easily change the board representation to whatever you want and keep changing it without having to do it a lot of refactor after that. So find all the board axes and refactor that to a function call before you modify it.

CHARLES E. LEISERSON: Yeah. Because the compiler will inline that stuff. So putting in a function call is not necessarily a bad idea. And if it doesn't, you can put it in macro, and it'll be effectively the same thing. So

great idea. Great idea. Yeah, question?

AUDIENCE: Anything else that wasn't there that you would highly recommend for us to look at?

CHARLES E. LEISERSON: Testing, testing, testing. If you can test and know that when you make a change it's correct and that you're not-- that's probably the number one hole that people go into is they don't test adequately. So having good test infrastructure is good. Yeah?

AUDIENCE: Some questions about codes counts-- I saw on Piazza there was [INAUDIBLE] if the beta changed the [INAUDIBLE] count will stay the same.

CHARLES E. LEISERSON: As long as you're doing it deterministically, yes?

AUDIENCE: Yeah. But even the references limitation is nondeterministic serially for [INAUDIBLE].

CHARLES E. LEISERSON: That's because there's a randomness thing. There's a little variable, and you can set the variable to be not random. And then it will be.

AUDIENCE: Is that a part of the set option RNG?

CHARLES E. LEISERSON: Yes.

AUDIENCE: When that's set, it's still nondeterministic.

CHARLES E. LEISERSON: Run serially?

AUDIENCE: Yeah. I checked the-- is there anything else that should be done? Or should it just be that seed option?

CHARLES E. LEISERSON: I think that's--

AUDIENCE: [INAUDIBLE]

CHARLES E. LEISERSON: [INAUDIBLE], can give to her the mic?

AUDIENCE: Yeah.

AUDIENCE: [INAUDIBLE]. Yeah. Did you try the things that [INAUDIBLE] include [INAUDIBLE] like [INAUDIBLE].

AUDIENCE: Yeah. I got out of the opening book and I set the RNG option.

AUDIENCE: OK Let me test it again. Because when I tested it , it was deterministic. So [INAUDIBLE].

AUDIENCE: OK.

AUDIENCE: [INAUDIBLE]

CHARLES E. LEISERSON: Yeah. It shouldn't be-- if you run serially, the only things that should be doing things is if you're doing-- it should be deterministic if you turn off the random number generator. So as I say, that's put in so that it will behave nonpredictably. But that's exactly the kind of thing you want to find right at the beginning. So that's exactly the thing is find out all the ways of making it deterministic, and that would be really important for beta 2.

So Thursday, we have Jon Bentley, legend of the field, opportunity to meet a celebrity. Bring friends. He's going to give a great lecture.