Bio

I am a fourth-year EECS undergraduate, focusing on CS side of the major. I am planning to go into industry after this semester, so now is my last chance to take graduate courses. My inability to choose which topics within CS interest me most has lead me to take a large and eclectic set of classes.

I chose to take this class for two reasons: on the practical side, everything is multicore these days and it would be prudent to know how to best take advantage of that; perhaps more importantly, it sounded interesting and I like to learn.

Application: Computer Go

The game of Go is both one of the oldest surviving strategy games in the world and probably the single game where the most work is being done to try to increase the performance of computer algorithms. This is because Go, though it has a very simple set of rules, has complex strategy and a multitude of options at every move. The high branching factor renders simplistic algorithms such as alpha-beta search useless, and the need to prioritize between various battles going on in various parts of the board has so far prevented computers from rising above the level of competent amateurs. I myself cannot claim to be an expert in computer Go, but my father is, and I’ve picked up some things from him.

For a long time, computer Go was all about strongly focused searching and sophisticated evaluation functions, both guided by the programmer’s strategic knowledge, to the extent it could be put into code. But more recently the field has been dominated by Monte Carlo methods: pick a few moves based on what’s looking good, and then play out a random game from there as quickly as possible to see who wins. More specifically (and ignoring a certain amount of algorithmic variation between programs), it starts with a tree of moves that is one layer deep, with a leaf for each move possible from the current board state. For each move, that move is made, and then the game is played out randomly from there until there is a clear winner (usually until there are no remaining legal moves), which is counted as a reward of 1 if the player who made the nonrandom move won, and a reward of 0 otherwise. After that, branches are chosen by computing for each possible move $j$ the
sum $\bar{r}_j + \sqrt{\frac{C \ln n}{n_j}}$, where $\bar{r}_j$ is the average reward for that move, $n_j$ is the number of times that move has been tried, $n$ is the number of times that move or any of its siblings have been tried, and $C$ is a tunable constant which is usually close to 2. Whichever possible move has the highest score is the one chosen. This allows exploration while providing regret which is provably optimal to within a constant factor. When a leaf has been visited enough times, all the moves that could follow it are added to the tree as leaves below it, and then visits to their parent will choose a different one of them each time until they’ve all been tried once, then it will proceed by the same formula. Once a sufficient amount of computation is deemed to have occurred (which is usually determined by how much time the program has budgeted for choosing the next move), the child of the root which has been visited the most times is chosen as the move to make. There are also various techniques that may be used to guide the search to some extent, or to take advantage of the fact that a good move on one particular turn will often be a good move on many turns, but what I have described is the basic idea.

While the algorithm as I have described it sounds serial, it is actually fairly parallelizable. Since it is a probabilistic algorithm, it does not matter so much if threads are working on information that is somewhat outdated. And since it is a probabilistic algorithm, its ability to find good moves is heavily dependent on how many trials it is able to run. The implementation I am familiar with will spawn enough threads to keep the available processors busy, each of which is building its own tree of moves, and have the threads periodically share their findings with each other. The last I heard, it stops getting stronger somewhere around 64 cores, but I was lead to believe that this is most likely a latent concurrency bug and that the limit imposed by the need for the threads to communicate lies significantly higher. Furthermore, even past that limit, running multiple completely independent copies of the algorithm and averaging the results ought to continue to provide some added benefit, though of course at a much lower rate. Thus, while this algorithm does not scale well forever, it should continue to scale to some extent almost indefinitely.

References
David Fotland (personal communication)

http://www.smart-games.com/manyfaces.html