Parallelizing Alpha Beta Search

This recitation introduces the Abort library and explains how to parallelize alpha beta search for the final project.

1 Getting started

We will be working on your final project code directly. You should have already formed groups and received an email with instructions on how to clone your repository.

Make your project by using `make`. You can run the khetengine by running `khetplayer`. Once started, it will wait for input via stdin. It accepts the commands described in the Universal Khet Engine interface document. To start a simple search, input the following commands:

1. uki
2. position classic
3. go depth 6

The engine will then begin to look from the classic starting position for the best move, while looking 6 ply deep in the game tree. You can stop the search at anytime with the stop command. Once the search ends, the engine will output the best move found.

2 The Abort Library

Right now, time control is done by having a thread monitor the time time, and set a boolean flag once the time exceeds the allotted time for search. This flag is also set if the user sends the stop command. The search functions in `ABSearch.h` periodically check if this flag is set, and return early if it is. Ideally there would be some sort of abort command that terminate the spawned search functions.

As we saw in lecture, spawned computations can be seen as forming an invocation tree, where nodes are computations, and children are spawned computations. The Abort library simplifies this by having the the flag encapsulated in a object, along with a pointer to a parent Abort. If a particular computation wants to know if it should terminate, it should check that its parent has not issued the abort command(which sets the flag), and that the parent is not itself aborted by its own parent. This amounts to checking the flag of all nodes in the path to the parent in the invocation tree.

This is the main idea behind how the Abort library in `Abort.h` works. The `abort()` function sets the boolean flag, so that spawned computations will eventually see it set and return early. The `isAborted` function recursively checks whether its own flag is set and whether its parents flag is set. Lastly, the `Abort(Abort *p)` function creates a new Abort object with p as its parent.

3 Implementing Time Control with Abort

We can now reimplement time control by replacing the `global_abort` boolean flag with an Abort object. We can replace any checks of its values with calls to `isAborted`, and use `abort` to set its values.
• **Exercise:** Use Abort to replace the boolean flag, and verify you can still stop searches with the stop command.

4 Parallelizing Alpha Beta Search

Now that we are more comfortable with the Abort library, we can see how to use this to parallelize alpha beta search. The simplest parallelization is to simply check every subtree in parallel, while ensuring that there are no races in updating the appropriate alpha and beta values. We need to able to prune away subtrees, which amounts to aborting the spawned searches of them.

To do this, we will need to associate every search with an abort object. Whenever we call search, it should take as an argument the Abort object of the computation that spawned it. If the search function periodically checks whether it is aborted and returns early, we can get the desired behavior. The questions of when, and how often to poll are still an open questions. While the polls take time proportional to the height of the invocation tree, the tree for ABSearch is as high as the depth of the search. Also, after one poll, the abort objects will all be in cache, making later polls quick. However, any time spent polling is time not spent searching. Empirically, polling before spawning any additional work seems to be reasonably effective.

Currently the results of any search are passed as arguments into a lambda function. Lambdas are a new function of C++0x, and here we are using them as nested functions. They are actually still objects of a compiler defined type, which is automatically filled by using the auto keyword. The `root_search_catch` functions use a mutex to ensure serial execution and update alpha and beta as needed. It currently returns whether any pruning occurs, so that the loop that spawns searches is exited early. We can modify this to call abort instead of returning any value.

The for loop which spawns searches bears some special mention. We have learned in the course that using a `cilk_for` loop is better than a for loop containing `cilk_spawn`. This is a special case in which the opposite is true. Ideally the list of moves would be ordered by the some heuristic as to which are likely to be better. If they are, we would want to ensure that the search of those moves are stolen first. This will ensure we get good alpha values quickly and so prune sooner.


• **Exercise:** Modify `root_search` and `search` to use Abort objects for pruning. Change the for loop which searches the subtrees to be parallel.