CS 758: Programming Multicore Processors (Fall 2016)

Homework 3

Cilk: Parallel game AI

You should do this assignment alone. No late assignments.

***Special thanks to Dr. John Mellor-Crummey at Rice University for the basis of this assignment found** here*

Filelist for the assignment:

- Template Files
- Cilk documentation
- Cilk-5 Paper
- Cilk hyper-objects (reducers, etc) paper
- Cilkscreen paper
- Cilk++ paper (Cilk++ was the version after Cilk-5, but before Cilk Plus)
- Cilk Plus Website (The website for the quasi-open source Cilk Plus project)

The purpose of this assignment is to give you experience writing programs with Cilk. You will also get practice debugging with the Cilk race detector, `cilkscreen`, and understanding performance using the `cilkview` tool.

You will do this assignment first on Ale-01 and Ale-02 - each containing two Xeon x5550 processors which have 4 cores each for a total of 8 cores. We have given you individual accounts on this machine. Use them only for 758 homework assignments and, with instructor's permission, for your course project. The accounts and storage will be deleted at the end of the semester unless you obtain permission from the instructor for extending them.

Then, you will again use godel to see if your application is scalable to 64 cores (plus however many threads). You will use the same run.sh interface to godel.

Cilk

Cilk is a parallel extension to the C/C++ programming language. For this assignment you will be using "Cilk Plus" included with the Intel compiler suite. Cilk adds a few keywords to C (cilk_spawn, cilk_sync, cilk_for, etc). From Wikipedia: The biggest principle behind the design of the Cilk language is that the programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel; it should then be left to the run-time environment, particularly the scheduler, to decide during execution how to actually divide the work between processors. It is because these responsibilities are separated that a Cilk program can run without rewriting on any number of processors, including one.

Game AI

For this assignment you will be parallizing an AI for the game Othello. Othello (also known as Reversi) is a strategy game played on an 8 x 8 gameboard. The rules to the game and a brief explanation of strategy issues are available on the Wikipedia page for Reversi. A website with an interactive program to play the game is http://www.web-games-online.com/reversi/. A sequential program that enables two human players to play Othello is included in the template files here. You may use this program as you see fit to get a jump start on your assignment. Feel free to use the code directly as the basis for your parallel solution. Some of you may be uncomfortable with the board representation that was chosen using bits of a 64-bit integer. If so, you may prefer to rewrite the code using a 64-element array of

characters as the board representation. While this is less compact, it is equally acceptable. If you would find it more intuitive to develop your own solution from scratch rather than building upon the code that I provided, that is fine but not necessary. The template provided is in pure (C99) C. Cilk has extensions for both C and C++ so you may choose either if you choose to write your solution from scratch.

You will write a shared-memory parallel program in Cilk that enables the computer to play the game. Implement the computer player as a parallel function that plays Othello/Reversi by searching n moves ahead to select the "best board position" for its move. For example, searching 1 move ahead for Player 1 means selecting best legal move for Player 1 based only on comparing the board states that would result from any of the possible legal moves for Player 1. Searching 2 moves ahead for Player 1 means selecting the move that would result in the best board position after Player 1's move followed by Player 2's best move. This process of considering alternating moves generalizes naturally to consider lookaheads of n moves. Note that if one player cannot move, his opponent can move again if any legal moves remain; your search should account for this accordingly. Note: Constructing a sophisticated board evaluator to compute the best strategic move is beyond the scope of the assignment. A simple board evaluator that computes the best move by maximizing the difference between the number of your disks and the number of the opponents disks on the board will suffice. However, if you want to implement a more complicated evaluations function feel free.

Algorithm

To implement an AI for Othello you will us the minimax algorithm, detailed on Wikipedia. The minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. Information on minimax can also be found in Russel, Norvig: *Artificial Inteligence: A Modern Approach*. From Wikipedia, the psuedocode for the algorith is

```
function integer minimax(node, depth)
    if node is a terminal node or depth <= 0:
        return the heuristic value of node
    α = -∞
    for child in node:                          # evaluation is identical for both players
        α = max(α, -minimax(child, depth-1))
    return α
```

The minimax algorithm turns out to naturally fit into Cilk's task parallel programming model due to its recursive nature.

Template files

You are provided with some template files available here. These files include the main program (othello.c) and two players, a human player, and a simple AI player. The simple AI chooses a random move from all of the available moves and is included to test your "good" AI against. The Simple AI can give a unique game every time. However, do NOT run experiments with a completely random AI. The seed chosen in the template gives an "interesting" game and should be used when running experiments. You can uncomment the random seed in `main` while debugging to produce more than one game scenario.

Also included in the template is a timing library. It times the second player of the game, giving both total runtime and per-turn runtime.

Problem 1

Implement a sequential version of GoodAITurn. This is where you will implement the minimax algorithm with a depth of **DEPTH**(the depth parameter found in othello-good-ai.c).

A)

Provide a couple of sentences in the final writeup describing how you convinced yourself your minimax algorithm was implemented correctly.

Problem 2

Implement the parallel version of GoodAITurn. You may choose to implement a new version of GoodAITurn, or keep a single version of GoodAITurn which is parallel and add "-cilk-serialize" to the CFLAGS for the sequential version of Othello. Hint: You may want to use reducer objects. How reducers and other hyper-objects are implemented can be found here and how to use reducers can be found here. (The link to the hyper-object paper is included only for reference, don't feel that you have to understand all of the details in the paper.)

Your submitted program should be free of data races. Cilk's cilkscreen tool uses binary rewriting to instrument your executable to check itself for data races as it runs. Running your program with cilkscreen at the front of the command line will check an execution for data races. If cilkscreen reports races, make sure that you compile your program with the -g flag (make DEBUG=1) and run it again. (Executables compiled with -g have more detailed race reports, which will help you identify the references involved in the data races.)

A)

In a couple of sentences, describe your experience with using cilkscreen to find and debug data races. How does this compare to finding data races with OpenMP and TBB? In other words, would you rather use Cilk, OpenMP, or TBB for this assignment? Why?

B)

In a couple of sentences, how does parallelizing a program with Cilk compare to using OpenMP and TBB? For this particular problem, how do you think your Cilk implementation would compare to a hypothetical OpenMP and TBB implementation?

Problem 3

Cilk's cilkview tool uses binary rewriting to instrument your program to profile its parallelism. It will report the total amount of work in your program, the critical path length, the average parallelism, as well other measures such as the total number of stack frames, spawns, and syncs. Compile variants of othello-parallel to have the good AI player use lookahead depths 1, 2, 3, 4, and 5. For each lookahead depth, use cilkview to profile your program.

A)

Graph your measurements of the parallelism found by cilkview with respect to the lookahead depth. Give a couple sentences of why you think the graph looks like it does. Did it look like what you expected?

B)

What is "burdened parallelism" (as reported by cilkview)? How does the burdened parallelism scale with the lookahead depth and with the work and span reported by cilkview?

C)

Given the output from cilkview, how do you think this program will perform on 8 cores at each lookahead depth?

D)

Again, given the output from cilkview, how do you think this program will perform on 64 cores on godel at each lookahead depth?

Problem 4 on ALE

Run both your parallel and sequential versions of othello on one of the ale nodes for lookahead depths of 1,2,3,4,5. For a depth of 5, run your parallel version with 1,2,4,8,16 threads. You can specify the

number of threads for Cilk to use similarly to how you did for OpenMP by `CILK_NWORKERS=N othello-parallel`.

A)

How does the speedup of your parallel implementation of othello scale with the lookahead depth? Explain why you think this is. Does it look like you expected?

B)

How does your parallel version of othello scale with the number of threads? Remembering back to 3c, is this what you predicted? In this particular case, how well did cilkview predict the speedup? Provide a couple of sentences to explain why or why not cilkview predicted the performance.

Problem 5 on GODEL

Run both your parallel and sequential versions of othello on godel for lookahead depths of 1,2,3,4,5. For a depth of 5, run your parallel version with 1,8,16,32,64,128,256 threads. You can specify the number of threads for Cilk to use similarly to how you did for OpenMP by `CILK_NWORKERS=N othello-parallel`.

A)

How does the speedup of your parallel implementation of othello scale with the lookahead depth? Explain why you think this is. Does it look like you expected? How is this similar or different from the ale machine? Why?

B)

How does your parallel version of othello scale with the number of threads? Remembering back to 3c, is this what you predicted? In this particular case, how well did cilkview predict the speedup? Provide a couple of sentences to explain why or why not cilkview predicted the performance.

Tips and Tricks

You will be using the Intel compiler from Intel ComposerXE 2015. It is installed in /s/intelcompilers-2015. Before attempting to compile anything with Cilk, be sure to source the following files like below:

- `source /s/intelcompilers-2015/bin/iccvars.sh intel64`
- cilkview/cilkscreen can be found in /p/course/cs758-david/public/cilkutil/bin/. You can add that to your path or run them by specifying the complete path. These will only run on the ale machines. These binaries fail on godel and on the lab machines with Ubuntu.
- **Start early.**
- Run multiple experiments to reduce the chances that your performance numbers are wrong due to contention.
- Run `top` on ale-01/02 and see if anyone else in your class is currently running experiments. If so, wait a few minutes. This isn't needed for godel ;).

What to Hand In

Please turn this homework in on **paper** at the beginning of lecture. You must include:

- A printout of your serial version of GoodAITurn and any new functions you added to support it.
- A printout of any changes to the serial version you made when parallelizing it.
- Answers to the questions in problems 1-4.

Important: Include your name on EVERY page.

Page last modified on September 25, 2016, at 09:28 PM, visited times