I parallelized the line solvers of two different nonogram solvers. Running pbnsolve’s line solver with 24 OpenMP threads made the solver take 85% longer. Running nonogram’s line solver with 24 Cilk threads made it take 7% less wall time, in terms of wall time. The reasons for this lack of speedup are discussed.

Background

“Pic-a-Pix,” “paint-by-numbers” or “nonograms” are a type of puzzle in which users attempt to reconstruct a small, pixelated picture by using the starting clues to deduce the color of each cell. Each column and row of the puzzle’s grid are labeled with “clues”: numbers that indicate the number of contiguous cells that should be filled in with a given color. For example, the clue for the first row of the below puzzle is “2 1,” indicating that the row will contain two black cells in a row and then one black cell, in that order, and separated by at least one white cell. Each puzzle should have one unique solution that is logically deducible from the input.

Most pic-a-pic solvers involve at least two different “phases” of operation, a line solver and then some sort of tree search algorithm. I will sketch those out in broad strokes, and then will discuss each of the two existing solvers that I modified in more detail.

Line Solvers

The first stage of almost any pic-a-pic solver is to iterate through the puzzle line by line, in approximately the same fashion that a human puzzle solver does. For each line, a line solver considers the line’s clues and its current partial solution, if any. It then applies a number of nonogram solving techniques (e.g., the “left-right-overlap” algorithm) to make any possible deductions from the given information, and then updates the partial solution with anything it has deduced.

Clearly, iterating through each line once doesn’t guarantee a solution. The line solver might (a) reaches a solution, (b) reaches a contradiction (only possible during the tree search
phase), or (c) stalls with the solution still only partial. At this point, the solver shifts to a second approach, usually involving tree search.

Tree Search
Most human-solvable pic-a-pix puzzles will reach a solution through use of the line solver alone. However, some puzzles require further reasoning. For human readers, this usually takes the form of reasoning about two lines simultaneously; for example, line x may have two possible solutions, but one of them would render line x+1 unsolvable.

To solve this kind of situation, the solver pursues each of the possible solutions separately. Most of them will lead inevitably to a contradiction; it can then safely conclude that the inverse solution was the correct one. This sort of problem-solving can be represented easily by a search tree, with each node in the tree representing a different choice being examined (cell 47 is blue; cell 47 is red; cell 12 is blue; cell 12 is blank; etc.) Most nodes will lead to a contradiction relatively quickly, thus quickly pruning the search space.

I will now discuss both of the baseline solvers I used; in each case, I'll present how the existing, sequential algorithm works, the challenges it poses to parallelization, and how I parallelized it.

Sequential Baseline #1: Jan Wolter’s pbnsolve
The late Jan Wolters wrote pbnsolve, one of the fastest, most stable, and most comprehensive solvers for what he called “paint by numbers” puzzles (thus the “pbn”). I chose this solver to parallelize, because (a) it was in well-commented C, (b) its author’s website was extremely useful in my research phase, and (c) it seemed like it’d have the best chance of being able to solve some of the puzzles that are too big and difficult for most solvers.

Unfortunately, it’s also over 9000 lines of extremely tightly coupled code.

pbnsolve represents each line as (a) a line in the “puzzle” struct, a “clue,” which gives the clue info (e.g., “1 3 8”), (b) a “job,” which may or may not exist in the job heap, and (c) a line of cell structs in the solution struct. Each of these four representations references the others in some way, and of course, each of the cells in that line also exists in duplicate in another line. (i.e., each cell is represented separately in its column and in its row). This extremely tight coupling made it extremely difficult to make changes to the code, because the linesolver ends up needing to access and/or modify all of those structs every time it makes a deduction.

pbnsolve’s Line Solver
pbnsolve runs linesolve in a while loop; on each iteration, it calls a function that grabs the top job from the heap and reheapifies the heap. (Jobs are kept sorted by the amount of new
information added to them since last time they were linesolved.) This heap runs extremely efficiently; small, line-solvable puzzles are usually solved in less than .01 seconds.

This should have been a red flag that I shouldn’t bother parallelizing the line solver, but I felt committed to my project proposal. I decided to parallelize the linesolver using OpenMP, since my main plan was to parallelize execution of a loop. In order to do this, I converted the `while (next_job)` loop to a `for (job in jobs)` loop that runs over every job in the heap. This immediately slowed down the algorithm substantially.

However, it was also during this implementation that I realized that hardly data in pbnsolve’s structs can be changed without necessitating changes in the other structs related to the same cell or line. Because of this, virtually all the code has to be inside a mutex. This was therefore an extremely unsuccessful parallelization effort.

**pbnsolve’s Tree Search**

pbnsolve’s tree search is called “probing.” This phase chooses a few cells whose color is unknown; it also has a clever way of choosing cells that are likely to run into contradictions quickly if those cells’ colors are guessed. It then guesses each possible color for each of those cells, and runs a breadth-first search to find contradictions by calling the linesolver on each possible state of the puzzle.

I originally intended to parallelize this process, but again the data structures were very difficult to modify. Probing involves running the linesolver in a backtrackable mode, where changes are made to each clue and the solution, but a history is kept in the puzzle so that the deductions can be undone if the probe results in a contradiction. In other words, if a guess (e.g., “cell 50 is white” leads to a contradiction in every circumstance, then the entire state of the puzzle and solution have to be reset to the their pre-probe state, with “white” removed from the list of possible colors of cell 50. This makes parallelizing the tree search difficult, because the algorithm had to be substantially rewritten to give each instance its own data — and the memory footprint of all these structs is surprisingly large.

pbnsolve also has a number of other modes, such as a “sprint” mode where it switches to guess-based depth-first-search, and a mode in which it memoizes linesolver results.

All in all, I spent over three weeks trying to refactor pbnsolve to get better results than my tragically bad parallelized linesolver, but the results below are the best I managed.

**Sequential Baseline #2: Jakub Wilk’s Nonogram Solver**

During the last few days before the deadline, I decided to switch gears substantially and make my second parallelization attempt on an entirely different solver, Jakub Wilk’s nonogram solver, which is also written in C. I chose Wilk’s solver in large part because it’s so simply structured (and about 1700 lines of code instead of 9000!). Like Wolter’s, Wilk’s code runs a line solver followed by a backtrack-able guessing algorithm. Unlike Wolter’s, his data
structures are simple: there’s a struct called Picture that represents the entire puzzle, including input clues and the output picture cells, and a Queue of lines.

**nonogram’s Line Solver**
Wilk’s line solver is called “shake.” It works by putting every line of the puzzle into a queue, and then repeatedly calling a function called finger_line that takes the Picture and Queue, gets pops a line from the queue, partially solves it by calling a function called touch_line, and then updates the picture struct.

This line solver is much more straightforward to parallelize. Instead of using OpenMP, this time I used Cilk, so that I could keep the existing while loop (and also so that I could call it “Cilk Wilk”!). Instead of calling finger_line while !is_queue_empty, I now spin in a while (1) loop cilk_spawning finger_line and breaking when is_queue_empty. Inside finger_line, I lock the Picture and Queue with pthread mutexes to avoid race conditions, but at least some of the work can still be done in parallel.

**nonogram’s Tree Search**
Wilk’s nonogram solver has a backtrack mode, which duplicates the Picture struct before modifying it. I was excited to see this, since this removes all the problems of maintaining state that I encountered in Wolter’s solver. However, I didn’t have a chance to parallelize it.

**Extremely Underwhelming Results**

**pbnsolve**
As should be clear by now, parallelizing pbnsolve was much more difficult than anticipated, largely because of the tightly coupled data structures all storing interdependent information. I parallelized pbnsolve using OpenMP on Latedays; the thread count was thus 24. My test set in this graph is a set of 24 puzzles, most of them from Jan Wolter’s original test set. Since he was interested in testing the capabilities of different line solvers, his test set consisted of a few small line-solvable puzzles, and a wide range of large, non-line-solvable puzzles. I removed the outright unsolvable puzzles and added in a few large but line-solvable puzzles from webpbn.com, for a total test set of 42.
As can be seen, parallelizing pbnsolve only made it slower. The sequential solver solved almost all puzzles in less than two seconds, with seven very difficult puzzles taking much longer. My parallel version took longer for 36 of the 42 puzzles, or 86%. In total, the sequential solver took 48.3 seconds to solve all 42 puzzles, while the parallel one took a dismal 89.0, meaning it took 84% longer.

I discussed this outcome with Kayvon, who suggested that my algorithm change was to blame, since I now have a for loop through all pending jobs instead of a while loop through the most promising jobs. However, I tested a sequential version of my inferior algorithm, and it performed about halfway between the original sequential algorithm and my parallelized algorithm. I conclude therefore that half the slowdown comes from the inferior algorithm, and the other half from the overhead of creating threads and locking resources. Because most of the “parallel” code has to be inside mutexes because of the tightly coupled data structures, there simply isn’t enough parallelism to make up for this overhead.

**nonogram**

Wilk’s nonogram solver couldn’t be tested on the same data set, since it can’t solve puzzles as complicated (or with as many colors) as Wolter’s. However, it comes with about 150 test puzzles, so I used 100 of those to test the sequential solution vs. the parallel solution in Latedays. As can be seen below, the sequential solver solves most puzzles in around 2 microseconds or around 13; I didn’t check every puzzle, but I’m pretty sure the difference between these groups is whether or not it’s line solvable.

With very coarse-grained locks on the Picture and Queue, I was able to speed up the algorithm very slightly. As the graph shows, the smallest puzzles took slightly longer to solve, presumably because of thread overhead, but the big savings were in the higher end of the spectrum. Eight percent of puzzles took >20 seconds to solve on the sequential solver, compared to only 2% on the parallel solver.
This is somewhat surprising, since it was the line solver that I optimized and not the backtracker that handles large jobs. However, when the algorithm switches into backtracking mode, it makes a few guesses and then runs the line solver on each guess; that means that the backtracking puzzles actually make much more use of the line solver than linesolver-only puzzles do, so it does seem reasonable that the benefit more from the parallelization of the line solver.

In total, the sequential solver took 1.048 seconds to solve all 100 puzzles, while the parallel solver took only .875 (wall time).

Still, it is very clear that these are not the results I was hoping for when I set out to parallelize a line solver. In the case of Wilk’s nonogram solver, I think the main difficulty is best described by this quote from Cilk’s documentation:

One mistake that new Cilk Plus programmers sometimes make is to time a parallel loop that does not have enough work. If the loop is small enough to complete before any steals are likely to occur, then the time to finish the loop does not decrease as \( P \) increases. In this situation, the fastest running time is likely to occur with \( P=1 \); more workers should not help, and might even hurt performance.

One threshold I often use when coding for my multicore desktop machine is the “microsecond test.” If the time to execute the entire loop serially is only a “few” microseconds, then the loop might be too small to parallelize with multithreading. If the loop requires hundreds of microseconds to execute, then it may benefit from parallelization. The microsecond test is derived from a rough estimate of the cost of a steal in Cilk Plus, that a successful steal incurs an overhead that is on the order of microseconds.

Wilk’s code solves the smallest puzzles in two microseconds total, which is divided among anywhere between 10 and 100 calls to finger_line. The computation done inside finger_line is exceedingly fast; what makes Wilk’s solver slow is not that finger_line does a lot of work, but that so many calls to finger_line are made. Parallelizing that call with Cilk was therefore a very bad strategy for speeding up the algorithm as a whole. It adds the costs of thread maintenance and work-stealing to the algorithm and provides very little.
Lessons Learned
It’s fair to say that this is the most disastrous final project I’ve ever undertaken. I didn’t expect linear speedup, but I at least expected 24 threads to be more than a couple percentage points faster than single-threaded code!

However, I have learned a lot from this project.

First, working with an existing code base was a valuable experience. I am a lot more committed to commenting my code for future users now, and above all I’m more aware that structuring code and data well are crucial if you want to be able to modify the code later. I managed to get better results from Wilk’s code in one day than I got from three weeks of working on Wolter’s code, and that is largely because Wolter’s code is so simply structured.

I also learned that I should spend more time analyzing the code’s performance before making design choices. Before I parallelized Wilk’s code, I did check that nearly all of the solving time was spent in finger_line. However, it wasn’t until after I parallelized that function that I realized that the code wasn’t spending a lot of time in each finger_line call, it was simply calling finger_line a lot of times. That has important ramifications for how to parallelize the code, and I made a suboptimal choice because I didn’t have that information.

In conclusion, although I am very dissatisfied with the fruits of my labor, I think I have at least gained some important insights that I hope will make my future labor more fruitful.