CPD 17/18
KUDURO: SOLVING SUDOKU USING OPENMP

Authors: Mariana Martins, Miguel Pinho & João Guerreiro

IST ID: 80856, 80826 & 81248

1. INTRODUCTION & METHODOLOGY

The purpose of this project is to create an efficient parallel solver for the famous sudoku puzzle using OpenMP and study the speedup relatively to a serial solver. As a grid size, we generalize from 4×4 to 81×81.

The sudoku problem is widely studied, among the most known algorithms developed are, simulated annealing and a sparse linear system of equations as an expression of the puzzle.

According to Wikipedia, kuduro is characterized as uptempo, energetic, and danceable, which we take as an inspiration for our proposed method.

Our proposed method is a simple iterative search, which splits itself as needed, to generate a balanced workload. On a first approach we developed two serial solutions and then we began testing various OpenMP and data storage solutions, which we later used for comparison to the final one. The implementation of those can be accessed here [1].

2. SERIAL

We decided to implement two kinds of serial methods as we were not sure at the begginig which would prove easier to parallelize than the other.

Our first proposed method was the most straightforward use of recursion in order to try every combination.

Our second method was an iterative backpropagation and used an array to keep track of previous plays. This was the solution which we ended up using as a start point for the parallel version.

3. PARALLELIZATION CHALLENGES

The tree search for a sudoku solution is an inherently parallel problem. In any given node of the tree each sub-tree can be solved as independent problems. The natural decomposition is for each task to be a search in one sub-tree, given the previous state and starting point.

The challenging part in this problem is generating tasks with similar sizes, as we do not know beforehand how balanced the work will be in each sub-tree. A static partition, where we start by splitting the search into several sub-trees and assign their complete solving to one thread, is prone to be very unbalanced.

4. PROPOSED SOLUTION

Our proposed solution is to dynamically split and redistribute the tasks as needed. When a thread searching one sub-tree notices other threads are idle, it splits its tree and passes part to another. The splitting is always done at the top of that sub-tree, so as to try to divide the problem evenly and with the most independence.

A threshold (at roughly 90% of the tree height) was added to guarantee there is not over splitting in the final part of a search. Some extra splitting was also forced in the beggining both to guarantee there are always enough tasks and to try more branchs in the beginning.

We found that, in practise, this method needs very little splitting to keep all the threads busy.

A shared counter was used for the threads to know when others were idle, which is increased whenever a thread finishes its task. When a thread notices the flag is non-zero it splits and decreases that counter. A similar shared flag exits for one thread to signal all the others when it has found a solution. The task splitting itself could be done with a small shared queue, but is instead implemented with the omp task construct, for simplicity.

5. SYNCHRONISATION

The shared flag is protected when writing, but not when reading, since a bad read will not affect the overall execution. The same happens with the counter flag.

When possible, we opted for using omp atomic instead of omp critical, since the overhead is lower and it is non-blocking when regarding other atomic operations.

6. TESTED SOLUTIONS

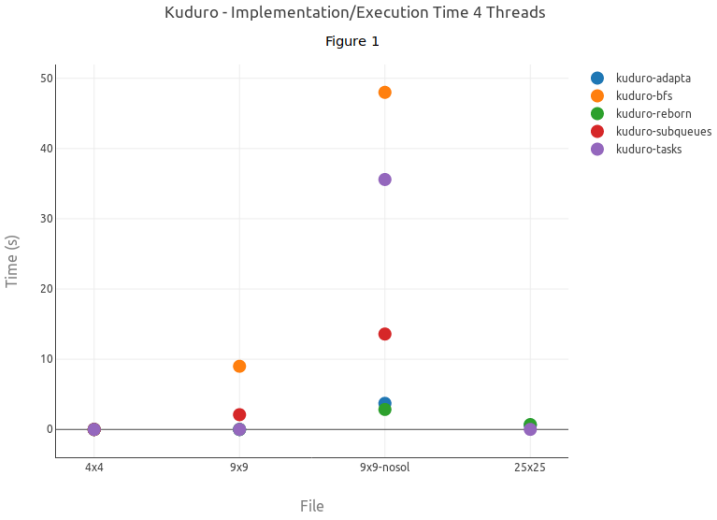
Our tested solutions were:

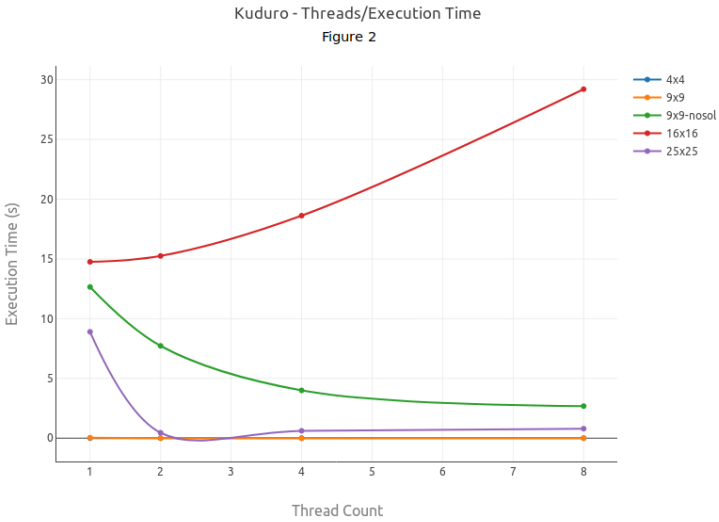
[1] Dropbox folder

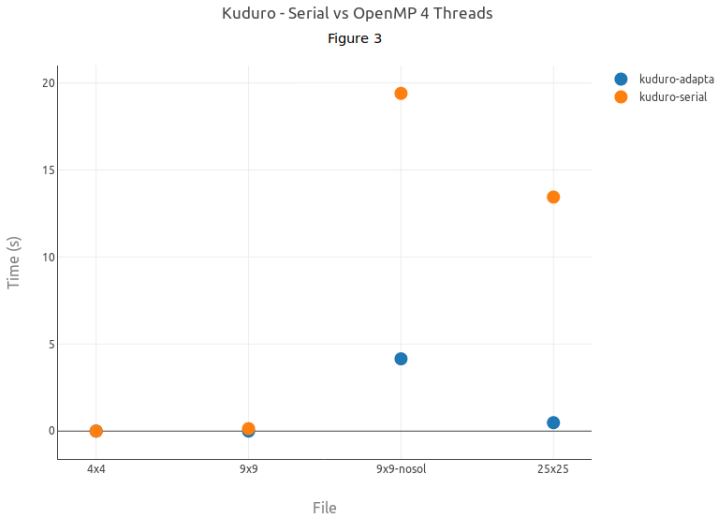
1. kuduro_bfs - breadth first search using a queue
2. kuduro_subqueue - iterative search of fixed depth, within a subqueue, later flushed to the main queue
3. kuduro_tasks - recursive search using omp task
4. kuduro_reborn - hybrid of 3, where each task is an iterative search of fixed depth (different per tree level)
5. kuduro_adapta (final) - iterative search with dynamic splitting

7. BENCHMARKS

In figure 1, we benchmarked our implementations to determine the best. Kuduro BFS and Subqueues were unable to complete the bigger maps due to excessive memory consumption. Kuduro Tasks was very effecient in maps with solution but had a very bad performance on the remaining maps.







Kuduro Reborn was the main contender to Adapta, but too many tasks were being created and it relied too heavily on the omp task dynamic management. Its main advantage was that different paths were tried sooner and randomly, which usually gave it adavantage in big sudokus with solution. The solution sometimes could be found sooner, but those results were hardly consistent. Finally we decided to go with Kuduro Adapta which showed on average the best results.

We then, in figure 2, benchmarked our proposed implementation with 2,4,6 and 8 threads which showed stagnating returns as predicted and an abnormality in the solving of the 16x16 map, probably due to the redistribution of tasks leading to the solution being found sooner. We concluded the 9x9-nosol map gives us the best overview of our implementation.

Therefore in figure 3, we benchmarked our serial against our OpenMP implementation which revealed significant speedups in larger files. Our most relevant benchmark (9x9-nosol) reveals a speedup of 4.67.

In terms of memory, we measured the peak heap size of our program, using valgrind, which was $291.1KiB$ versus $92.3KiB$ of the serial implementation on 9x9-nosol.

8. RESULT DISCUSSION

From the results obtained we conclude that our openMP delivers a solid speedup compared with the serial implementation while still retaining a small memory footprint. As for scalability we notice very diminishing returns beginning at 4 threads in most small maps, we theorize on larger maps we will notice the same diminishing returns starting in a larger number of threads.