Kevin Zhu (kzhu2)
Hao Zhang (haozhan2)

Parallel Sudoku Solver

URL: https://github.com/Hao2747/sudoku-parallel-solver.github.io

SUMMARY:

We plan on parallelizing a sudoku solver using multiple algorithms and then comparing overall runtime performance for each approach. Approaches include breadth-first search, distributed work queue, work-stealing distributed queue (hopefully), and a constraints-solver approach (hopefully), and if time permits, a DFS approach. We plan to be coding in OpenMP and if time permits, we also will try an approach in MPI for large sudoku boards.

BACKGROUND:

Sudoku varies on the compute intensity based on the approach taken to solve the board. At a classical level, it is a graph based problem, where each node is a possible sudoku board and each connecting vertex indicates which square was filled to move from one possible board to the next. Proper sudoku boards have one solution.

Initially, we need to find the possible numbers for each empty square. This can be done in parallel because we need to traverse the entire board to find the missing numbers in each row, column, and square, which can be done in parallel. In the next step, the cpus can choose which empty square they want to work on.

They can all choose to work on one square or try a divide and conquer approach to work on different squares. Another approach is a CP or trying to solve a constraints problem. While the aforementioned possibilities are more ‘trial-and-error’ focused, the sudoku wiki has a series of strategies ranging from ‘tough’ to extreme strategies to solve the board. Each of these strategies has varying amounts of parallelism. While it may take extra work to check for these strategies, they may also greatly reduce and prune the search space.

THE CHALLENGE:
The problem is inherently challenging because DFS is not easy to parallelize. The naive method for solving Sudoku involves using backtracking to solve one square at a time and recursively, but it’s not so intuitive how to make threads work together using OpenMP. This doesn’t mean that it is impossible to do (check references, and we will at least try to make an attempt on this).

Parallelism is not exactly as intuitive as in Lab 3 with Nbody. Blindly spawning tasks for every empty square leads to work redundancy. For example, in a trial error approach, where each CPU gets its own square, because every square must be filled, a processor must assign a value to the square another CPU is working on, but how does it communicate what values have been tried. Additionally, if one square is solved, should it notify all other working threads to update their board.

From our project, we hope to have a more concrete understanding of the parallel algorithms, communication overheads, and would like to create a distributed work queue and measure its performance to other approaches.

As for workload, because of a tree-like structure, the child is dependent on the parent. For parallelization, as described, the children need to communicate or understand what is not being done. Generally, the memory access can be quite random within the grid and there is a lot of divergent execution. Because the Sudoku problem could technically be represented as a graph, we also have to make sure that there is no workload imbalance as some parts of the tree might be deeper than other sections.

What makes this challenging to implement as a work queue is that in OpenMp there is a performance penalty in using critical and having high contention regions. We need to structure and assign tasks in a way that can circumvent this issue.

RESOURCES:
We plan on using both the GHC and PSC machines, and may get benchmarks from either machine. The PSC machine will be important for the MPI approach as well as testing the scaling of our algorithm. As of now, we plan to code the entire thing ourselves starting from scratch.

There are no other resources that we need. We do not believe a GPU will be good for our workload.

Parallel DFS Paper:
https://research.nvidia.com/sites/default/files/publications/nvr-2017-001.pdf

GOALS AND DELIVERABLES:

PLAN TO ACHIEVE

- Sequential/Parallel Breadth-first search solver
- Implemented distributed task queue strategy
- Sequential/Parallel some constraints

- Good testcases, working sudoku validator, good data structure design

HOPE TO ACHIEVE

- Sequential/ Parallel Depth-first search solver
- Sequential and Parallel extension for MPI for large workloads
- High performance for MPI workload

In our project, we plan to have separate sequential and parallel implementations of various algorithms. We will also be computing the overall speedup for the workload in a separate measurement as well. This way, we think the speedup comparison is more fair for different workloads and show the true achieved speedup for each task.

For example, a depth and breadth first search may not remotely have the same runtime because it may be the farthest depth (favoring depth first) or the 2nd node (favoring breadth). We also plan to time and measure different sections of the code. For example, the initial phase of finding the possible digits may be parallelized. We will try to see our MPI performance for that section. For scaling, we plan to use the PSC machines to test the parallelization on large numbers of cores.

We plan to analyze the performance differences and characteristics between each of the parallel models we create and find out where the bottlenecks are in each approach (ex was it communication or work redundancy) and try to further tune our algorithm. For a given workload, we should be able to see what strategies made a positive difference in solving problems.

PLATFORM CHOICE:

We have been coding in C++ for MPI and OpenMP tasks and we plan to continue to use them for our project. The Gates and PSC machines, along with the OpenMP and MPI versions, are resources that we have already been using and are familiar with.

Sudoku boards are relatively small so using a shared memory approach makes sense. The naive intuition for creating a sequential algorithm involves backsolving which is recursive by nature. As can be seen in Lab 3, recursive algorithms with little computational complexity are better done on OpenMP as the communication overhead is not worth it.

If the grid is large or if there are many critical sections in the OpenMP code, then using MPI is probably better suited.

SCHEDULE:

	Personal Deadlines (End of Week Goals)
Week of March 27	Finish Writing Proposal Initial Research and Outline Done (What is possible, what we want to accomplish, and goals)

Week of April 3	Any additional Research and Consultation necessary A finished sequential version with a bootstrapped code base and data structures - Agreed Code Style - Ways to import / Validate Designs - Few Sample Test Cases - Backtracking/Recursive Implementation - Start Initial OpenMP Version
Week of April 10	- Adding More Testcases - Completing and finishing an initial openMP implementation and gauging/documenting performance We may also want to start the work queue version and DFS version at this time.
Week of April 17	Finish work queue version and continue to work on DFS version Start adding strategic version ideas in parallel with fall back to regular work Queue Version Start Distributed Que Version
Milestone April 19th	Finish Initial Code Base + validation Have performance checker Have quite a few testcases ,but a system that can easily be expanded on Finish all sequential Versions Finished Naive OpenMP version (BFS) Working on/nearing completion of WorkQueue Version
Week of April 24	Hopefully complete DFS version Finish Distributed Queue Version
Week of May 1	Final report, summary and presentation. Wrap up any loose ends