ParaSudoku

SUMMARY:
We implemented a sudoku solver in C++ using openMP which combines two algorithms to achieve good performance and speedup.

BACKGROUND: Describe the algorithm, application, or system you parallelized in computer science terms. Figures would be really useful here.

There are a few key data structures here.

1. Cell_t: This is a structure to hold the cell's answer and its potential answers (candidate). If the answer for this cell is found, the answer field will be a non-zero integer. The candidate field is a boolean area which will tell which number can be put into this cell. A false in candidate[3] would mean this cell cannot be put in a number 4.
2. int sudoku[16][16] : this is a 2-d array storing the sudoku. A zero means nothing is filled in the cell.
3. fake_binary: This is a structure we used to convert an integer into an array to tell the thread how to fill its sudoku. For example, a fake binary of {4, 5, 7} would mean to fill the first, second and third blank cell with the fourth, fifth, and seventh available candidate into the cell.

The algorithm takes in a sudoku puzzle and outputs a solution if there exists one. To fill a blank in the sudoku, one must check the row, column, and the block to make sure no duplicate number exists. This is one potential parallelization point. Another parallelization point comes from the fact that "trying" is needed to solve some hard sudoku puzzles. A sudoku solver must try different combinations of answers in order to find the right one. And these different combinations could then be parallelized.
When doing updates on the rows, columns, and blocks, there are no dependencies in the program. Each thread can work on its own row. After all threads are done, they will move on to update the columns, and then blocks. Unfortunately, updating columns will result in very bad cache locality compared to updating rows. This is due to the nature of the cache line. Fortunately, this part of the program is only taking a small part of the entire computation time and does not affect speedups by a lot.

APPROACH:

First, we take the input sudoku puzzle .txt and convert it into a cell_t array. The cell_t array will be as follows. If the cell has a number to it, cell.answer will be that number. If the cell is blank, cell.answer will be 0 and cell.candidate = {True}, indicating that all numbers could be put in this cell. This is because we have just initialized the sudoku and have not updated the candidates yet.

5	3	1,2,3,...,9	1,2,3,...,9	7	1,2,3,...,9	1,2,3,...,9	1,2,3,...,9	1,2,3,...,9
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

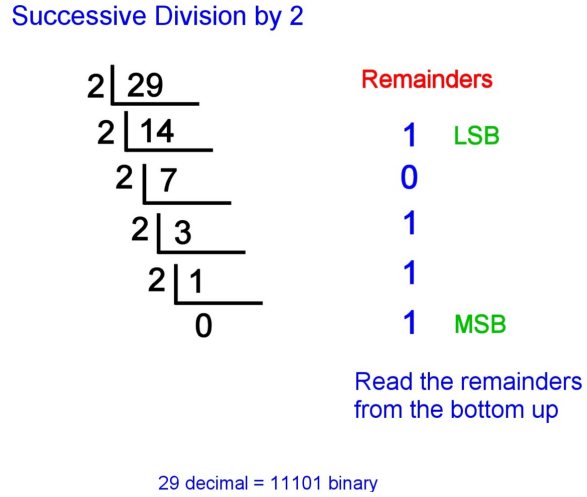
Sudoku initialization

Next, we do row updates, column updates, and box updates in sequence. For row updates, each core will look at one row and update the candidates. For example, the third cell in the first row will have a new candidate list of {T,T,F,T,F,T,F,T,T} after row update, indicating that 5, 3, and 7 cannot be put in this cell, while the rest are all available candidates. During updates, if the candidate list has only one candidate left, the answer field will be set to that number, indicating that this cell is filled.

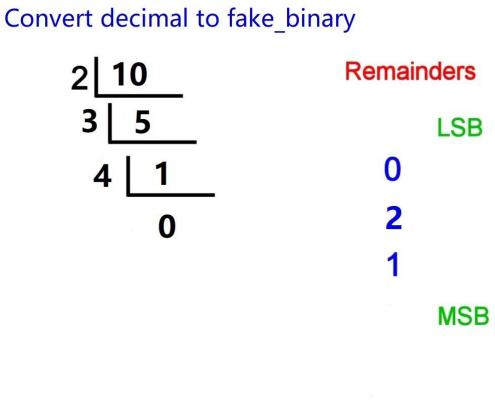
Doing the updates will fill in some blanks but cannot guarantee a solution. Thus, we need to use a backtracking algorithm after that. We store the numbers filled/not filled as a boolean array for each row, column and block. A backtracking algorithm is to fill in a cell based on the row, column and block array. The algorithm will fill in the cell from the first empty cell (front left) to the last empty cell (bottom right). For each possible solution for the cell, the algorithm will make a copy of the Sudoku and call the backtrack function with the new Sudoku. The backtracking will end when reaching the bottom right cell, which means all empty cells are filled. This tracking algorithm can guarantee a solution if there exists one.

However, it is a sequential algorithm which is very hard to parallelize. Thus, we come up with one solution to add parallelism. We take the first 8 blanks in sudoku, look at the number of remaining candidates of each blank, and calculate the total number of combinations. This gives us the total number of iterations that could be given out to threads. Each thread will then use its loop index ***i*** to get a fake_binary array. The fake_binary array comes from the idea of converting a decimal number to a binary number. As the graph (a) shows, the number will be divided by 2 multiple times until we get a quotient of 0. Instead of dividing by 2, in our case, the ***i*** is divided by the number of remaining candidates of the cell. For example, if the first three blank cells have {2, 3, 4} remaining candidates, the number will be divided by 2, then 3, then 4. The remainders will be put together and form the fake_binary array. Each digit can then be used to choose which candidate to be put in the cell. For example, a loop index ***i*** of 0 would mean that this thread will

fill the first 8 cells with the first available candidate for each cell, because 0 is {0, 0, … , 0} in fake_binary. A ***i*** of 10 would mean a {0, 2, 1} in fake_binary and the thread would fill the first blank with its first available candidate, the second blank with its third available candidate, and the third blank with its second available candidate.



(a) https://owlcation.com/stem/How-to-Convert-Decimal-to-Binary-and-Binary-to-Decimal



(b) fake_binary example

Sudo-code:

```
compute() {
    while (can be fill with doing updates) {
        row_updates;
        column_updates;
        block_updates;
    }

    int iterations = find_total_number_of_iterations();

    parallel for (int i=0; i < iterations; i++) {

        if (solution found)
            continue

        make a local sudoku copy;
        convert decimal_to_fake_binary;
        fill the first 8 blanks according to fake_binary;

        if (the pre_filled sudoku violates rules)
            continue

        do backtrack on sudoku;
```

```
        if solution found {
            #pragma omp critical
            update status to solution_found
            copy local_sudoku to sudoku_answer for output
        }
    }
}
```

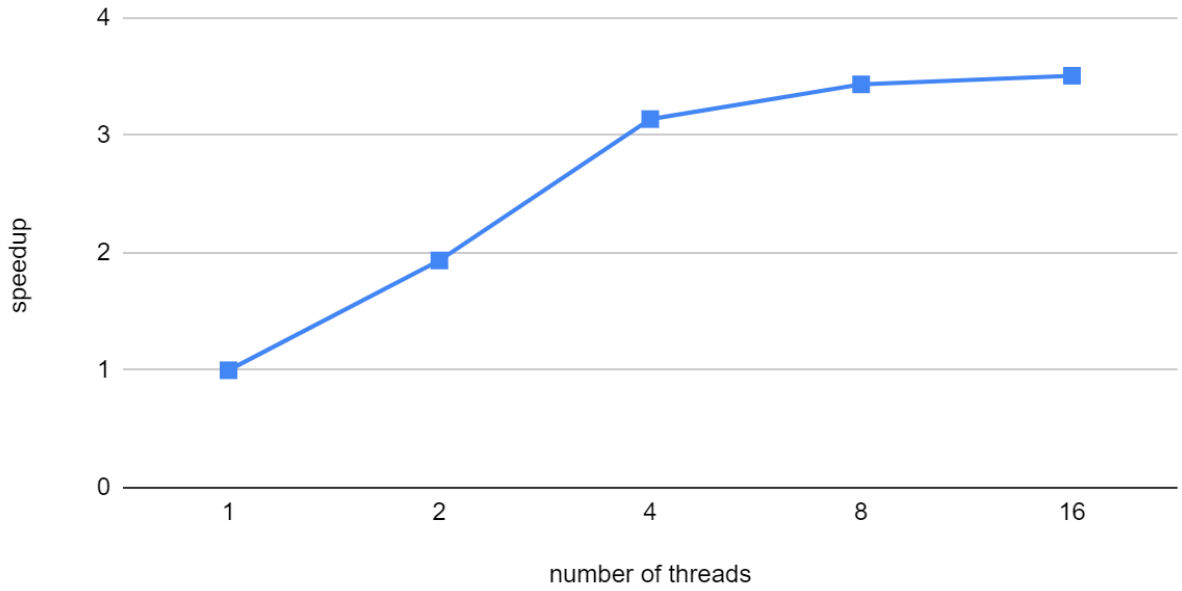
RESULTS:

We use speedup to measure the performance of our algorithm. There are two sizes of input, a difficult 16x16 Sudoku and a difficult 9x9 Sudoku. We choose the difficult ones because easier Sudokus are too quick to finish and it's difficult to calculate the speedup. We use ./sudoku -f input_filename -g grid_size -n number_of_thread to run the program, and the initial sudoku are stored in text files. The program will read the files and calculate the Sudoku using our algorithm.

The graphs of speedup are present below. Our baseline is an optimized parallel implementation for a single CPU. It's important to report results for different problem sizes for our project, since it can tell the difference of the performance when the amount of calculation is different. From the graphs below, we can see that both 9x9 and 16x16 Sudoku shows a positive relationship between number of threads and speedup. Thus, it means that our algorithm has successfully extracted parallelism from the problem. However, it is worth mentioning that speedup reaches a plateau as the number of threads is set to 16. We believe that it is because of the overhead of spawning too many threads and it brings in diminishing returns. Also, it appears that 9x9 reaches a bigger speedup than 16x16 does. We believe this is caused by the increasing amount of cache misses when the problem size is bigger. Detail explanation with supporting evidence will be in the next section.

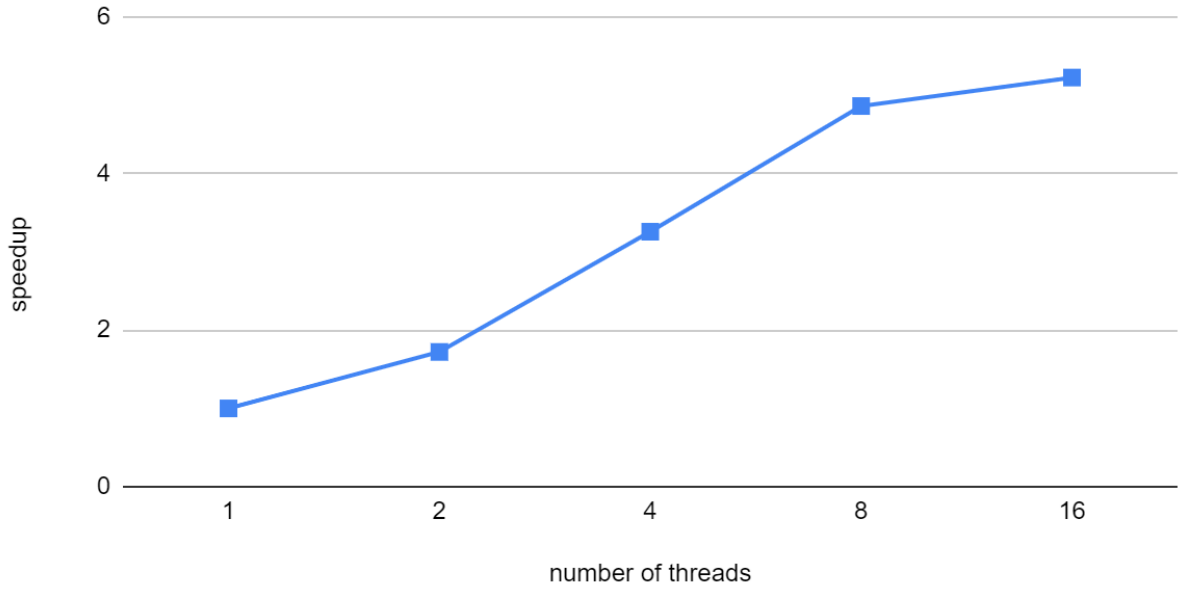
16x16 Sudoku Speedup

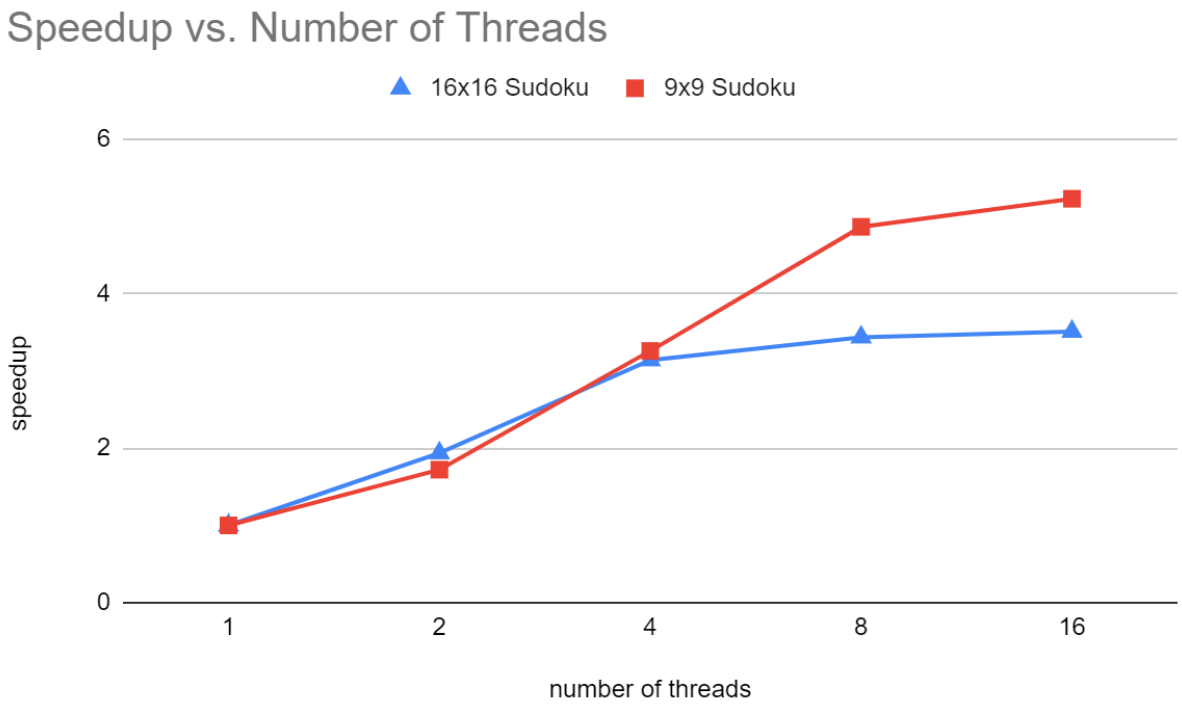
Speedup vs. Number of Threads



9x9 Sudoku Speedup

Speedup vs. Number of Threads





Our algorithm successfully achieves our goals - we reached approximately Nx speedup when the number of threads is smaller than 4. As the number of threads increases, the speedup is smaller than Nx due to the synchronization overhead.

IMPORTANT:

Cache misses rate limits the speedup of 16x16 Sudoku. According to the perf measurement of cache misses below, the percentage of cache misses increases from 0.7% to 7.8% when the input size increases from 9x9 to 16x16, causing the speedup of 16x16 Sudoku worse than 9x9 Sudoku. The increase in speedup is getting slower when the number of threads increases, and that might due to the synchronization overhead.

```
Performance counter stats for './sudoku -f inputs/hardest_9.txt -g 9 -n 8':

        1,654,223      cache-references:u
           11,989      cache-misses:u            #    0.725 % of all cache refs

      0.021861299 seconds time elapsed
```

```
Performance counter stats for './sudoku -f inputs/hard_16.txt -g 16 -n 8':

          299,070      cache-references:u
           23,408      cache-misses:u            #    7.827 % of all cache refs

      1.293864763 seconds time elapsed
```

After breaking the execution time of our algorithm into a number of distinct components, we found that most time is spent on backtracking. According to perf, 43.78% of time is spent on backtrack, 19.89% of time is spent on compute, 12.75% of time is spent on find_fake_binary, and 9.45% of time is spent on check_sudoku. There's room for improvement for the backtrack. It's possible to change backtrack to two separate functions, so that we can parallelize part of the backtracking.

```
Samples: 771  of event 'cycles:u', Event count (approx.): 636093954
Overhead  Command  Shared Object      Symbol
  43.78%  sudoku   sudoku             [.] backtrack
  19.89%  sudoku   sudoku             [.] compute
  12.75%  sudoku   sudoku             [.] find_fake_binary
   9.45%  sudoku   sudoku             [.] check_sudoku
   6.90%  sudoku   libgomp.so.1.0.0   [.] 0x000000000000a833
   6.06%  sudoku   libgomp.so.1.0.0   [.] 0x0000000000018b21
   0.30%  sudoku   libgomp.so.1.0.0   [.] GOMP_loop_ull_nonmonotonic_dynamic_next
   0.17%  sudoku   libgomp.so.1.0.0   [.] 0x000000000000a81c
   0.17%  sudoku   sudoku             [.] do_backtrack
   0.15%  sudoku   libgomp.so.1.0.0   [.] 0x000000000000a820
   0.09%  sudoku   ld-2.17.so         [.] do_lookup_x
   0.08%  sudoku   [unknown]          [k] 0xffffffffaff8c750
   0.05%  sudoku   libgomp.so.1.0.0   [.] 0x0000000000018dc3
   0.03%  sudoku   ld-2.17.so         [.] _dl_lookup_symbol_x
   0.03%  sudoku   libc-2.17.so       [.] _dl_addr
   0.03%  sudoku   ld-2.17.so         [.] _dl_relocate_object
   0.02%  sudoku   libgomp.so.1.0.0   [.] 0x0000000000018c99
   0.01%  sudoku   ld-2.17.so         [.] _dl_name_match_p
   0.01%  sudoku   ld-2.17.so         [.] memset
   0.01%  sudoku   ld-2.17.so         [.] check_match.9525
   0.01%  sudoku   libc-2.17.so       [.] __sigsetjmp
   0.00%  sudoku   libc-2.17.so       [.] __ctype_init
   0.00%  sudoku   libgomp.so.1.0.0   [.] 0x00000000000162b0
   0.00%  sudoku   libc-2.17.so       [.] __clone
   0.00%  sudoku   ld-2.17.so         [.] _start
```

Our choice of machine target is CPU, and GPU won't be a better target for our algorithm. Since backtracking is required in our algorithm, the task of each thread is not the same, and the amount of recursion is different in each thread. Because of that, the GPU is not a good machine target, and we need to run our program on CPU.

REFERENCES:

Wikimedia Foundation. (2022, April 27). Sudoku. Wikipedia. Retrieved May 5, 2022, from https://en.wikipedia.org/wiki/Sudoku

Ibm.com. 2022. IBM Docs. [online] Available at: <https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.1?topic=new-openmp-support> [Accessed 5 May 2022].

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT:

Total credit distributed amongst the participants: 50% - 50%

Backtracking algorithm, initialization and output: Yuxin

Fake_binary algorithm, update and fill sudoku: Alex
Sudoku validation and compute function: Alex, Yuxin
Debugging: Alex
Results analysis: Yuxin

Video Link:
https://cmu.zoom.us/rec/share/9RdSnN5QBf_g1z3UIka4sO1JFR72RKpdkySJodlux_IozvoPKNF23YrPFYPo2ov_.MFfv82y7wpBVppY6
Passcode: +!^C+v=0