Final project: Sudoku solver

Student name: Pham Tran Huong Giang

Matricola: 606579

1. C++ Thread

1.1. Implement

A thread pool with predefined number of workers is used to parallelly solve the sudoku board.
Each thread in the pool has its own array of nodes and a lock for this array. Thread explores new states of its nodes upto 3 levels and then puts all of these new nodes into its array. When a thread runs out of nodes, it will steal some nodes from its neighbor.
SudokuPar: Using C++ mutex to synchronize the queue of each thread.
SudokuJS: Using self-developed stealingQueue to synchronize the queue of each thread. In which, the owner thread only pushes and pops new nodes from the tail of the queue meanwhile the stealer will steal thread by thread from the head of the queue. An atomic variable is used to synchronize the stealidx - the index of the stolen node (to make sure the owner thread does pop the same node that the stealer will take). Another atomic variable is employed to synchronize the current number of nods of the queue.

1.2. Result

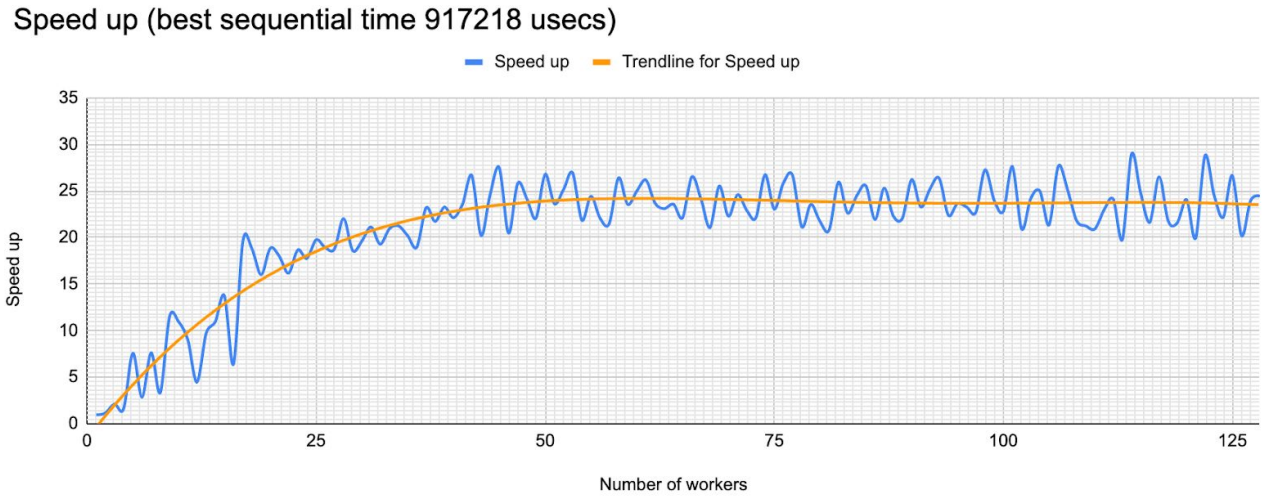
The program is tested with 3 sudoku boards as follow:

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

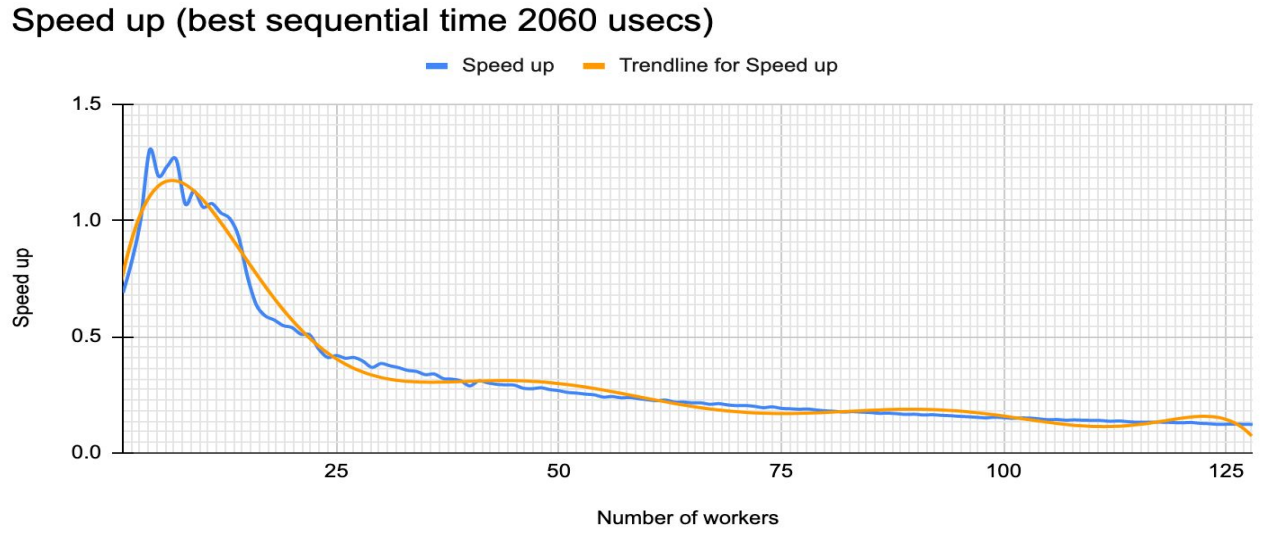
1	5	8	7	6	3	4	2	9
9	2	6	4	1	5	3	7	8
3	4	7	9	2	8	5	6	1
5	1	3	6	7	2	9	8	4
6	9	4	8	3	1	2	5	7
8	7	2	5	4	9	6	1	3
2	8	1	3	9	6	7	4	5
4	6	9	1	5	7	8	3	2
7	3	5	2	8	4	1	9	6

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

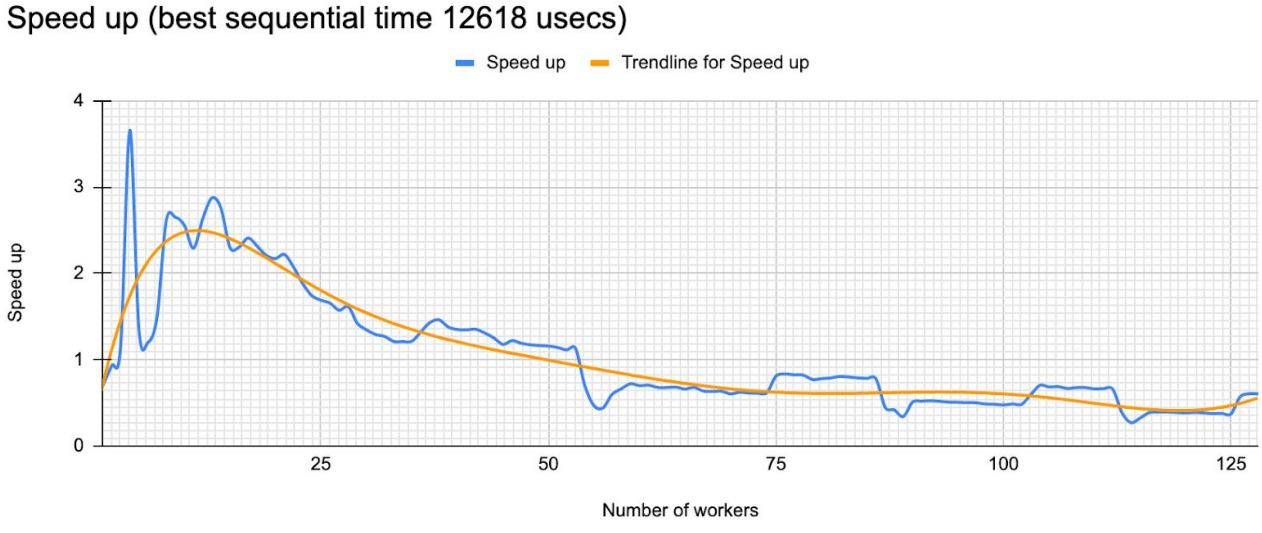
To evaluate the performance of SudokuPar, for each board, the program is executed with numbers of workers from 1 to 128 respectively.
With each number of workers, the program is run ten times to get the average running time.
Speed up is calculated by dividing the best sequential running time with the average running calculated above for each value of number of workers.
The first board costs 917218 usecs to run in sequential (best sequential result). The speed up as bellow:



The second board costs 2060 usecs for the best sequential time and the speed up as below:

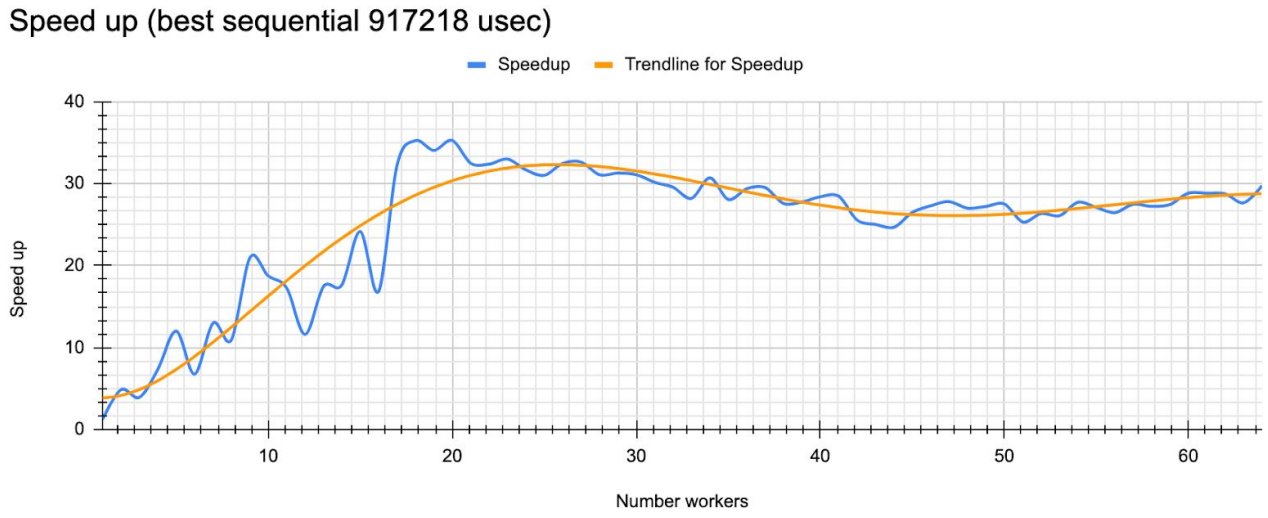


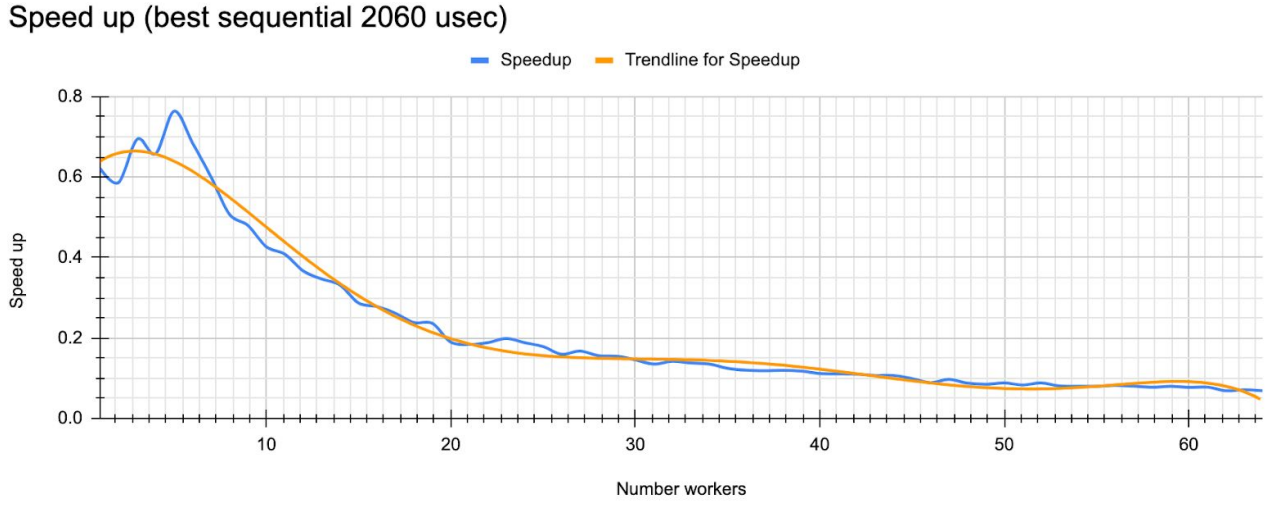
For the third board, it costs 12618 usecs for the best sequential run and the speed up as below:

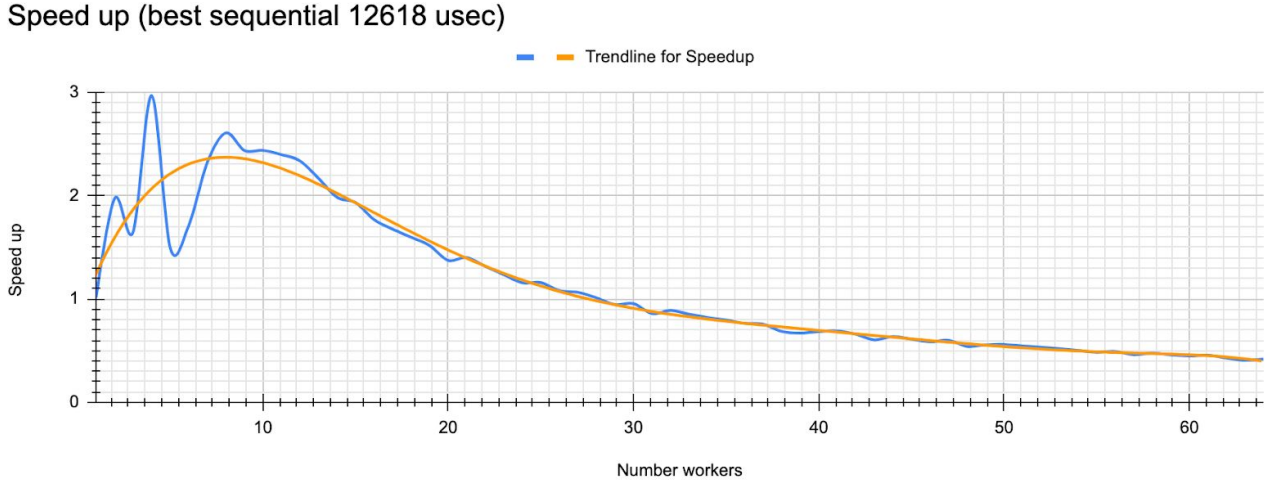


SudokuJS is tested with the same three boards with the number of workers varying from 1 to 64.

The performance is like below:







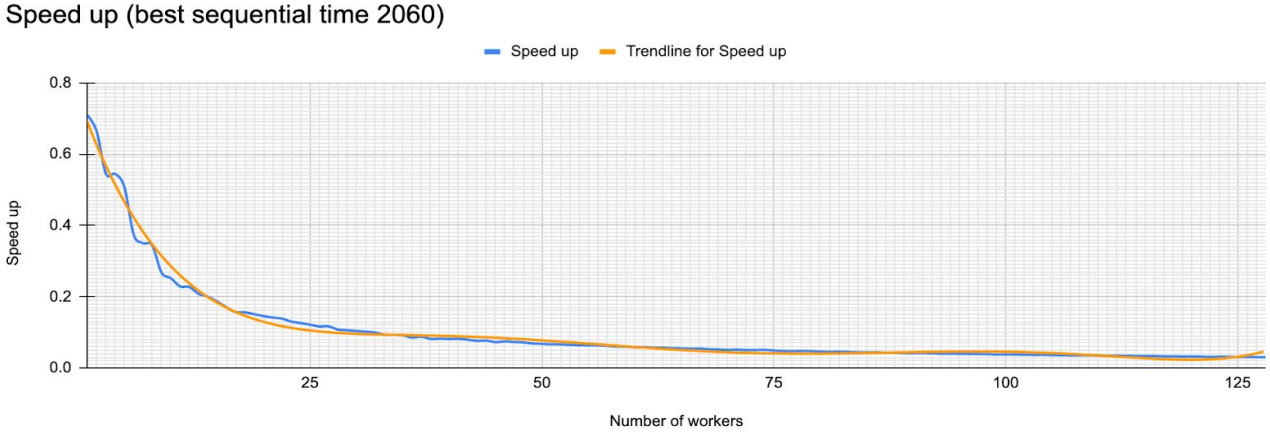
2. Fastflow

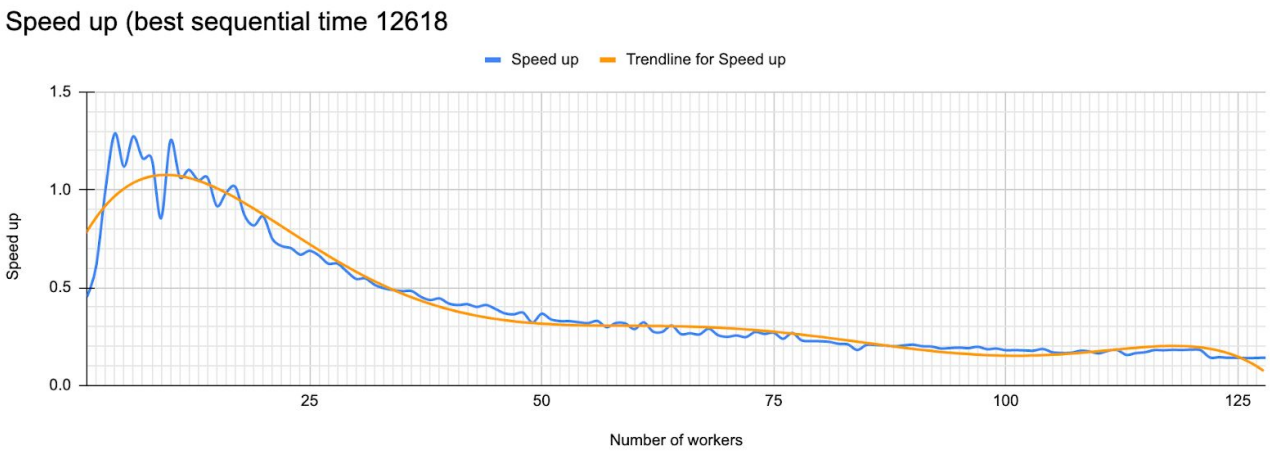
2.1. Implement

SudokuFf employs the fastflow master-worker model to solve sudoku boards.
A master works like a scheduler that respectively gives all the workers tasks to do.
A worker receives tasks (nodes) from master and explores the subtree from this node and then sends back to the master all the subsequence nodes.
The master, after receiving the vector of nodes that the workers sent, again, schedules these tasks to all the workers.

2.2. Result

SudokuFf is also tested with three boards above with the number of workers (excluding the master) from 1 to 128, the performance for each board is below:





3. Observation

It is easy to see that for the second board (with the best sequential time is 2060), it is hard to gain any speed up. Only with the implementation with plain queues and mutexes to synchronize the queues, the program can get a little bit faster (speed up = 1.3 with 4 workers).
For the board that takes the longest sequential time (the first board), the implementation with self-developed stealing queues gives the best speed up for the first 30 workers. However after that the implementation with plain queues seems to be more stable. The fastflow model also gets better speed up for the first few workers in comparison with the plain queues implementation but when keep increasing the number of workers, the speed up starts to decrease.
For the third board, which needs a medium amount of sequential time, the best speed up can get is from the implementation with plain queues with 4 workers for 3.6 speed up, nearly the ideal value. However, the implementation with stealing queues can get a speed up upto 1.9 with only 2 workers (meanwhile with 2 workers, the implementation with plain queues can get only 0.92 for speed up). The implementation with fastflow seems to be below the expectation in this case. Fastflow version can get only 1.28 with 4 workers (excluding the master).