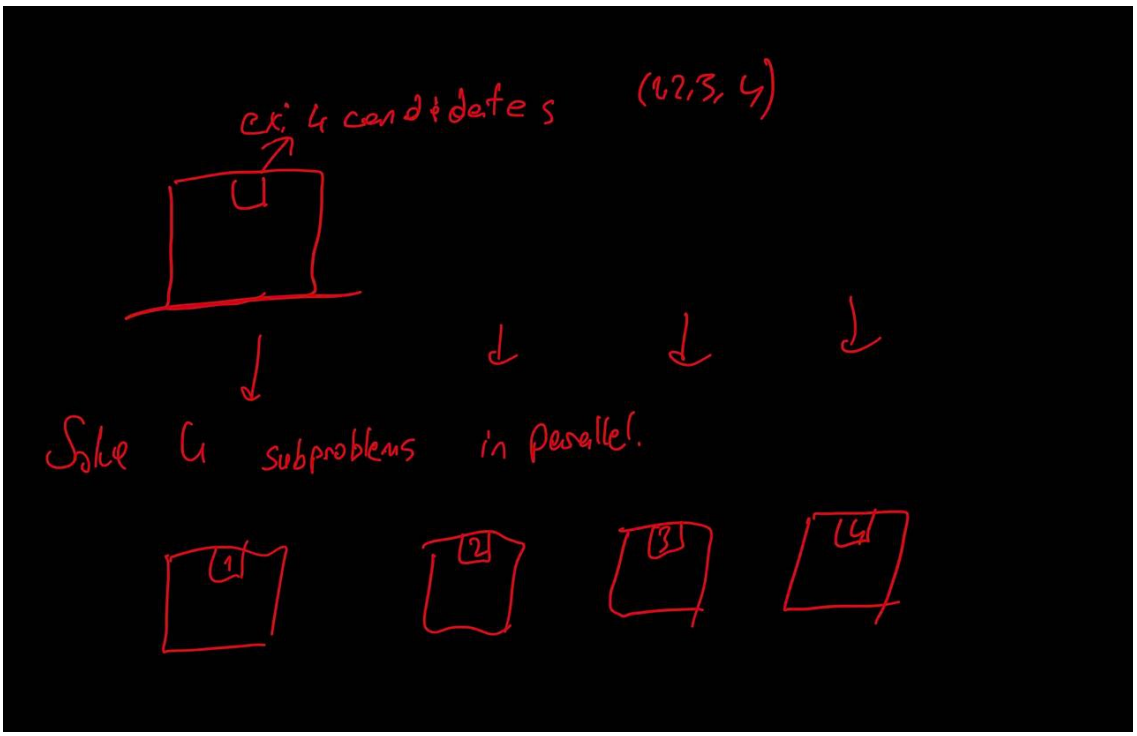
Parallel Sudoku Solver

Omer Faruk Ozdemir, ofo5108

Parallelization

1. In my implementation, I partition the task into different subproblems by selecting different candidates in the first hole, for each thread. The advantage of this approach is, the subproblems are completely independent and there is minimal synchronization(only when a solution is found). However with this approach we need to copy the map data to each thread, which creates an initial overhead.
2. To cope with this initial overhead, I tried using a global array which keeps each thread's memory in the global static memory. This increased performance in the easy benchmark, speedup was ½ instead of ¼, however it was slower with a higher number of threads. I suspect it was due to sequential manual memory copy management with memcpy.
3. I tried task spawning(limited by some number 9^2 or 9^3) and nested parallel loops for increasing scalability. However their 8 thread performance was slower, hence I followed the simple approach, only 1 hole parallelization.
4. I used spread thread affinity as it provided better performance. I expected spread to provide better performance, since we do not share data during the computation, threads can utilize the cache memory better when they are not hyperthreading.



System & Compiler

GCC 9.4.0 Fopenmp

Intel DevCloud Xeon 6128
2 sockets
12 cores
24 threads in total

L1i cache: 384 KiB
L1d cache: 384 KiB
L2 cache: 12MiB
L3 cache: 38.5 MiB

Sequential Performance

Easy: 594838 board/sec
Hard: 252 board/sec
Benchmark: 601 board/sec

8 Thread OMP Parallel Performance

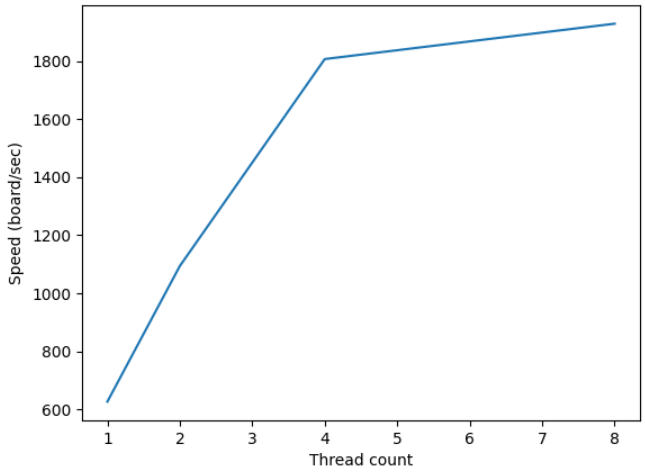
Easy: 151727 board/sec , speedup: 1/4
Hard: 808 board/sec, speedup: 3.21
Benchmark 1930 board/sec, 3.21

Best performance in any machine: 8 threads Devcloud, numbers above.

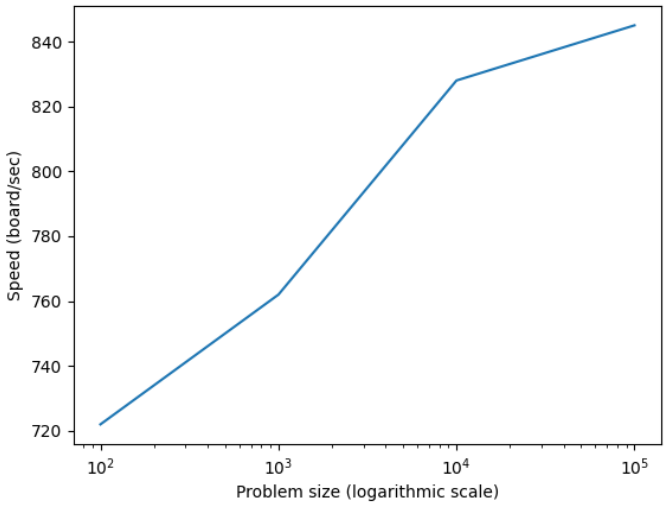
Scalability Plots

Fixed problem size, increasing thread count. We see that increasing thread counts provides a speedup, however the performance increase is diminishing significantly after 4 threads. From the plot we can infer that scaling the program to a very high number of threads won't be helpful.

Benchmark: dataset_benchmark



Fixed thread count(8), increasing problem size. We see that increasing problem size causes a small speedup, this is probably due to benchmark warmup and cache cold start. From the plot we can infer that th



Final Discussion

1. Parallel for structure was faster than tasks.

I expect this is because the runtime is lighter and we have less memory copy operations.

2. Dynamic scheduling is faster than static.

I thought dynamic would be faster, and the experiments supported this hypothesis. It is most probably due to subproblems having different size, hence having variance in the thread's job durations. Dynamic scheduling balances this variance.