Parallel and Distributed Computing
Project 1
Group 4

70969 - Mário Reis 75657 - Paulo Gouveia 76213 - Gonçalo Lopes

1. Serial Implementation

The serial implementation consists of a simple iterative DFS (deep first search) with two optimizations. The first is that only valid values are added to the search stack which saves wasted pushes to and pops from the stack. The second is that instead of each expanded node containing an entire copy of the sudoku board, because there are no different concurrent search paths, it is possible to store just the changes applied to the board. Reverting back the changes will take at most as much time as copying the entire matrix (worst case the path to revert is N^2 long) and storing one single integer takes less space than N^2 integers.

2. OpenMP Implementation

2.1. Parallelization

To implement the brute-force approach using a similar DFS, we used a Producer-Consumer pattern.

At the beginning of each iteration, each thread removes the top node from the stack. The node is analyzed and if applicable, it's child nodes are generated and placed on the top of the stack.

Each node corresponds to a state of the board and it includes:

- (x,y) pair corresponding to the cell that was changed in this node
- the value that was written
- the board with the written value

2.2. Synchronization and Load Balancing

The main synchronization concerns and our approach for each one are the following:

- **Push/Pop operations on the stack**

Each stack operation is enclosed inside a `#pragma omp critical` directive. This ensures there are no race conditions between threads when accessing the stack. The stack was also turned into a linked list, so the accesses to the top of the stack are easier, only needing to read the head of the list.

- **Terminating the main loop**

Because OpenMP does not allow an exit point in the middle of the parallel region, if there are no nodes to be processed and a thread isn't processing a node it falls into trap until there are new nodes in the stack. If all of the threads fall into the trap, it means there are no more nodes to be processed. When this happens, a shared

variable that is checked at the beginning of each iteration (`terminated`) is updated and the program exits the parallel region.

- **Load Balancing**

Using the Producer-Consumer pattern, each thread processes one node a time. Unless there are no nodes on the stack, the threads are continuously generating new nodes and checking for a solution.

- **Backtracking from nodes that have no children**

While in the serial version we use a stack pointer to traverse the stack, with multiple threads accessing the nodes and working on different paths it would be difficult to manage such structure. For this reason we included the boards on each node, sacrificing memory in exchange for not having to resolve the different concurrent paths should there be a need to recover a previous state.

2.3. Performance Analysis

The cpu used for all our tests was a 4-core intel i7 with 4Ghz clock speed. i7's have intel's hyper-threading technology which allows similar tasks to run over the same core as it were two cores. The operating system treats these cpus as if they had 8 cores and this is the reason for us to test with 16 threads as well: there shouldn't be any improvements to the speedup when using more than 8 threads, in fact it should be worse because of the overheads related to the creation and management of threads.

It is also noticeable that when we use a single thread, which is equivalent to serial mode, we actually get a worse performance due to the overheads of attempting parallelization. Yet another reason for the longer execution times of the parallel implementation is that for each node generated, memory needs to be allocated for its matrix, followed by a copy of the board state into that allocated matrix.

In particular for small boards, such as the 4x4, the performance is decreased the more threads that are used. This is expected because the computations done by each thread are smaller than the overheads injected by the management of them.

The following tables include the obtained times in seconds and the speedup for each parallel execution of each file. Omitted cells represent a time > 30 minutes.

threads	4x4	9x9	9x9-nosol	16x16-zeros	16x16	16x16-nosol
serial (1)	0,000015	0,036180	4,367735	0,000120	6,251977	488,065896
1	0,000019	0,493666	17,168447	0,000651	-	-
2	0,000062	1,174216	41,361520	0,004037	-	-
4	0,000074	0,141966	23,565137	0,006373	58,503361	-
8	0,000273	0,003471	15,944283	0,009650	29,028911	1 175,263775
16	0,000245	0,000733	15,028960	0,013019	-	-

Table 1: Performance results (in seconds) for both the serial and parallel code.

threads	4x4	9x9	9x9-nosol	16x16-zeros	16x16	16x16-nosol
1	0,7894737	0,0732884	0,2544048	0,1843318	-	-
2	0,2419355	0,0308120	0,1055990	0,0297250	-	-
4	0,2027027	0,2548498	0,1853473	0,0188294	0,1068653	-
8	0,0549451	10,4235091	0,2739374	0,0124352	0,2153707	0,4152820
16	0,0612245	49,3587995	0,2906212	0,0092173	-	-

Table 2: Speedups (in seconds) for each number of threads used with OpenMP.