CPD 17/18
KUDURO: SOLVING SUDOKU USING OPENMPI

Authors: Mariana Martins, Miguel Pinho & João Guerreiro

IST ID: 80856, 80826 & 81248

1. INTRODUCTION

The aim of the project was to develop a parallel Sudoku solver using the OpenMPI library, in which the workload is to be divided along a number of communicating processes. Not only the workload distribution is of importance, but also the communication hierarchy/structure. A serial solution is also provided, as comparison for benchmarking.

2. SERIAL

The serial implementation has changed since the first delivery in order to include a bit mask solution. This new solution is limited to maps of max dimension 128 but it provides increased performance on most maps due to the usage of simpler operations. As an example, a $25x25$ map is solved approximately 2.6s using the new serial method compared to ∞ with the old algorithm.

3. PARALLELISM CHALLENGES

The search for a Sudoku solution is not an easy problem to parallelize, as it is difficult to divide the problem in a balanced workload beforehand. The sub-trees size is unpredictable due to the constraints imposed by the Sudoku's rules.
Moreover, a distributed solution using MPI also implicates an overhead in communication, which can hinder the speed-up if not addressed in the design. A dynamic workload balancing can reduce the time processes are idle, but it can also implicate more time spent in communication.

4. PROPOSED SOLUTION

The solution proposed is meant to be a master-slave system where the master distributes the work amongst slaves, and then redistributes it according to the salves need.
To hand out the jobs for the first time, the master generates a stack of work, whose quantity is estimated based on an heuristic, to send work to each process and still keep some.
Each process is then sent work, which consists on the so-far filled positions. The slaves try to solve their assignment in depth and may be called in a round-robin approach by the master to share their work with other slaves. The shared work is extracted from the top of the work tree as the next value alternative. The master, however, gives priority to expend the work it already has in stack, before requesting more. If at one moment the master doesn't have any requests, it gathers work from slaves to push to the stack.
This was the basic idea to implement, yet by trial and error, some variations explained below were implemented leading to the final solution.

5. FINAL SOLUTION

When testing, the proposed solution had two major issues. Firstly, the slaves would end their work too fast and so the master would be overloaded with redistribution jobs, losing precious slave processing time. This was also caused by the second issue, which is passing all the positions filled so-far on each communication.
To correct this problems, a depth threshold stopping a too early redistribution was set. A very big threshold would lead to stopped slaves waiting for work and also a bigger set of positions filled to pass over.
The minimum stack size allowed was also a key variable, if too big it could be stopping the slavery traffic.
Overall, the redistribution did not improve the time, therefore the final decision was to remove it, and simply gather an initial work stack and distribute it amongst the processes as they needed it.

6. SYNCHRONISATION

The processes have to communicate between them to guarantee each is doing different work. In our final solution with a master/slave scheme, the master monopolises the work division, so there is not much of a problem.
In the solution with work-balance distribution, this responsibility is distributed and was handled by guaranteeing the work generated by a slave is a sub-tree of the

search tree it was given, starting from a lower level. The whole path in the sub-tree leading to that sub-tree has to be communicated.

7. BENCHMARKS

These benchmarks were made on the $cpd-x$ machines using the following profiles:

A(p 16)	B(p 8)	C(p 4)	D(p 2)
cpd-3 slots=4	cpd-3 slots=2	cpd-3 slots=1	cpd-3 slots=1
cpd-4 slots=4	cpd-4 slots=2	cpd-4 slots=1	cpd-4 slots=1
cpd-6 slots=4	cpd-6 slots=2	cpd-6 slots=1	
cpd-7 slots=4	cpd-7 slots=2	cpd-7 slots=1	

All of the binaries used were compiled with the following flags "-march=native -std=gnu11 -O2 -Os -ffast-math" The measurement of the elapsed time was done with the *time* directive, but due to server occupation the values had to be based on $user + system$ time and not on *elapsed* time. The serial results were obtained on a "i5-2500K 3.6GHz 24GiB RAM" but testing the same program on $cpd-6$ gave nearly the same results. Defining speedup (S) as $S = \frac{T_1}{T_N}$ and efficiency (E) as $E = \frac{T_1}{T_N \cdot N}$, the following results were obtained from 10 samples from each profile:

P|Time(s)	Median	Average	SpeedUp	Efficiency
Serial	2.09	2.13	-	-
A	1.38	1.27	1.68	0.11
B	0.90	0.81	2.62	0.33
C	0.31	0.33	6.47	1.62
D	0.39	0.39	5.48	2.74

Table 1. 25x25.txt

P|Time(s)	Median	Average	SpeedUp	Efficiency
Serial	4.03	4.04	-	-
A	2.80	2.88	1.40	0.09
B	8.30	7.67	0.53	0.07
C	10.02	10.12	0.40	0.10
D	12.86	13.24	0.30	0.15

Table 2. 16x16.txt

P|Time(s)	Median	Average	SpeedUp	Efficiency
Serial	278.35	278.35	-	-
A	11.76	11.76	23.67	1.48
B	9.465	9.465	29.41	1.84
C	95.095	95.095	2.93	0.73
D	190.22	190.22	1.46	0.73

Table 3. 16x16-nosol.txt, 2 samples due to run time

P|Time(s)	Median	Average	SpeedUp	Efficiency
Serial	7.83	7.89	-	-
A	3.31	3.32	2.38	0.15
B	1.82	1.80	4.38	0.55
C	0.46	0.46	17.01	4.25
D	2.37	2.39	3.31	1.65

Table 4. 9x9-worstcase.txt

P|Time(s)	Median	Average	SpeedUp	Efficiency
Serial	5.20	5.25	-	-
A	2.25	2.26	2.33	0.15
B	1.66	1.65	3.19	0.40
C	0.72	0.72	7.28	1.82
D	2.62	2.65	1.98	0.99

Table 5. 9x9-nosol.txt

All raw data and calculations on $9x9.txt$ is available in the git repository associated with this project.

8. RESULTS

The performances obtained are in general good, but depend a lot in the specific tests. The speed-ups in the Sudoku with solution are unpredictable and depend a lot in which branch the solution is, but seems to benefit from more processes, as there are more chances of finding the solution.
The no solution examples are better representatives of the workload distribution effectiveness, and tend to show good speed-ups, although these tend to stagnate or even lower with the increase of processors.

9. DISCUSSION

The results obtained were not as good as expected in terms of scalability and workload redistribution.
The solution with redistribution proved unusable in most practical cases, as the communication was far too frequent and too costly. At some point the system enters in such a state where the work is being swapped between processes instead of being solved.
The scalability is limited by the limitations of the master/slave model used, especially when all communication is done exclusively using the master. The master becomes can become a bottleneck, when too many processes are locked waiting for a reply from it.

9.1. Further improvments

A better model would have been to have a tree hierarchy of masters and slaves. The hierarchy height should scale

(logarithmically) with the increase of the processors and the work should be divided in a finer grain in the lower the hierarchies.
The redistribution could perhaps be fixed if it was done less frequently (tighter conditions) and by the initiative of the slave, after a fixed portion of work is solved.