Semestral project MIE-PAR 2013/2014:

Parallel algorithm for problem "Magic Square"(MAS)

Ciupin Iaroslav

master degree, FIT ČVUT, Thákurova 9, 160 00 Praha 6

December 3, 2013

1 Problem definition and description of sequential solution

The sequential algorithm is of type SB-DFS with the search depth bounded by n^2. An internal state is a permutation of the numbers from set M in an $N * N$ matrix. Possible final states are permutations of numbers from set M that satisfy the property of a magic square. A backtrack is taken if at least one row, column, or diagonal has a sum not equal to S. The algorithm terminates after finding the first solution. It always exists.

Specifics of my implementation of this algorithm:

- Algorithm takes as input initial configuration of NxN square(with some values already put) and tries to find solution that satisfies initial configuration. If it finds such solution, it fills input square with correct values. Otherwise, it leaves square unchanged.
- Depth-first search is implemented using recursion. Algorithm works as follows:

  1 Analises current state

  2 Gets a list with possible next states

  3 Gets first "next state" from list

  4 Applies it current state

  5 Recursively calls function on updated state

  6 After recursive call it reverts to initial state

  7 Go to step 3

- Because this algorithm is extremely slow - O($n^2!$) complexity, I don't use as input N - magic square side - and empty matrix. Instead, I fixed input to be N=6 and on every test change number of given correct numbers. So input is K - number of empty cells, $1 <= K <= N * N$, and initial configuration is square with $N * N - K$ correct values given. Input grows as I give less information to algorithm. Thus algorithm has O($K!$) complexity.

- Output: Prints filled magic square

- Small optimisations:

  1 Magic square is stored in memory as NxN matrix, but states generated on recursive calls are stored as transformations(list of values and indices, where to put these values). Usually list consists of 1 element.

  2 If it is possible to precompute last value in row, column or diagonal, algorithm stores several values in one transformation(that's why list is used). One transformation may contain up to 4 values. This reduces time complexity a bit.

  3 So space complexity becomes O($K! * const$), not O($K! * N^2$) as in case of N*N matrix usage for state storage.

Measurements

Input: K=21. Output in 814 seconds.

1	35	34	3	32	6
30	8	28	27	11	7
24	23	15			

1	35	34	3	32	6
30	8	28	27	11	7
24	23	15	12	20	17
19	9	18	26	10	29
4	31	14	21	25	16
33	5	2	22	13	36

Input: K=22. Output in 1159 seconds.

1	35	34	3	32	6
30	8	28	27	11	7
24	23				

1	35	34	3	32	6
30	8	28	27	11	7
24	23	2	17	20	25
14	15	22	31	10	19
13	26	9	12	33	18
29	4	16	21	5	36

Input: K=23. Output in 3271 seconds.

1	35	34	3	32	6
30	8	28	27	11	7
24					

1	35	34	3	32	6
30	8	28	27	11	7
24	2	9	21	19	36
10	18	22	31	16	14
20	25	5	17	29	15
26	23	13	12	4	33

Input: K=24. Output in 2464 seconds.

1	35	34	3	32	6
30	8	28	27	11	7

1	35	34	3	32	6
30	8	28	27	11	7
2	4	22	23	24	36
19	21	13	31	10	17
26	25	9	15	20	16
33	18	5	12	14	29

Interesting observation: For K=23 time is longer than for K=24. The thing is that outputs are different. And different magic squares are obtained in different time, because algorithm brute-forces all possible variants in ascending order.

2 Description of parallel algorithm and its implementation using MPI

Parallel implementation of this algorithm is quite straightforward. Program consists of 2 parts: master and slave. These 2 parts are enclose in if-else statement. If current's processors rank is 0, then it's master, otherwise it's slave.

Master starts calculation process. It begins with job generation for slaves. For this it generates partially filled magic squares(it puts 2 more values to initial global configuration) and then ask slaves to continue search in that direction. For job generation breadth-first search is used. When job generation is finished, master sends to all slaves NxN matrices representing initial configuration for each slave. And slaves begin depth-first searching on given configuration. If slave finds solution, it sends solution to master and quit. Master in its order, send to all slaves so called "zero-job", square filled with zeros which means all slaves should terminate. If slave doesn't find any solution for given configuration, it sends to master "zero-response", master

understands this and send to slave another job.

While slaves are working, master is listening to their responses in a loop. Slaves in the beginning when master generates jobs, are listening to their first job.

At the end of execution master and all slaves report their successful termination.

Master also provides user with useful information like how many slaves are working, how many jobs were generated, what is size of problem and what is message size.

3 Measured results and evaluation

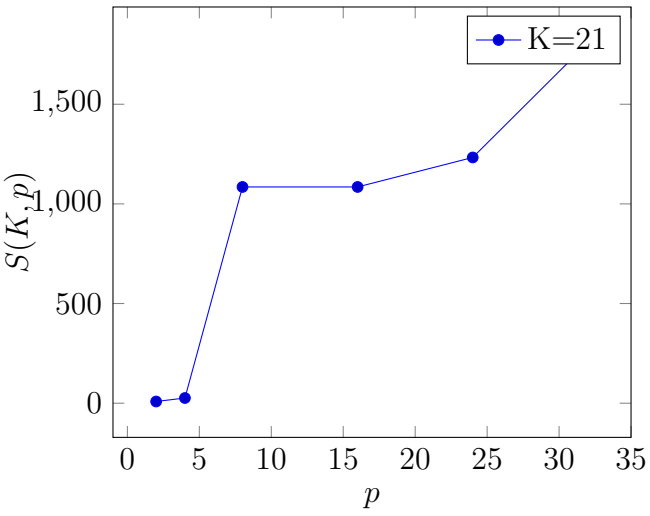
1. As I said before, because this algorithm is extremely slow, as input I use not N, but K - number of free cells in N*N square with N=6, $1 <= K <= 32$. As K grows, grows time of execution. I will use K = 21, 22, 23, 24 for p = 2, 4, 8, 16, 24, 32

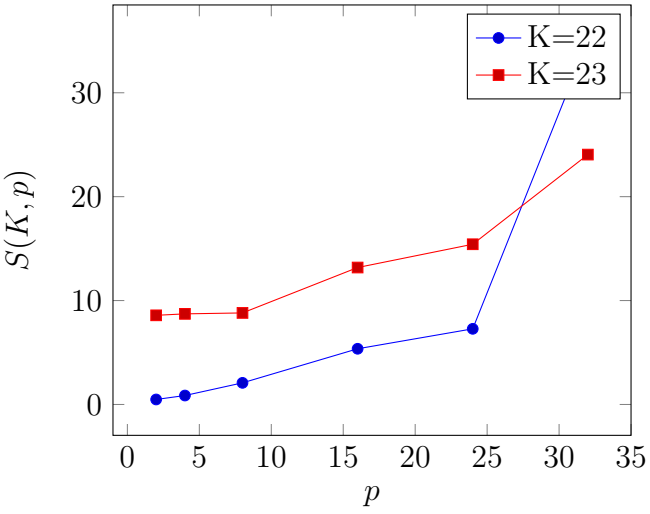
2. Parallel time T(K,p) in seconds for different K and p

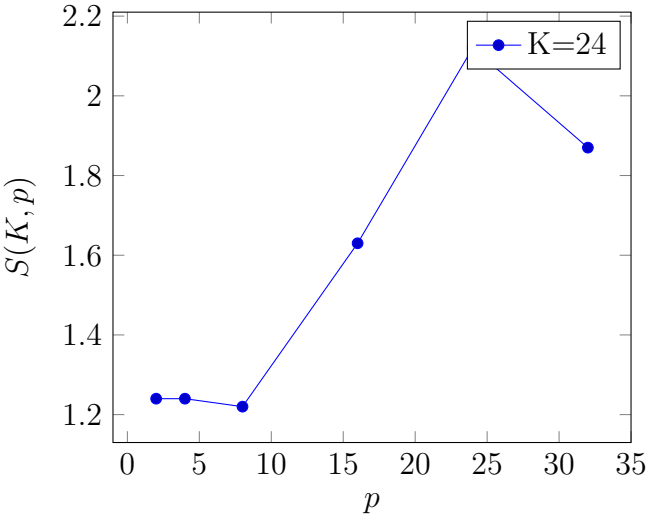
K \ p	2	4	8	16	24	32
21	96	31	0.75	0.75	0.66	0.45
22	> 40min	1341	555	216	159	33
23	381	375	371	248	212	136
24	1974	1984	2019	1510	1162	1311

3. Speed-up S(K,p) for different K and p

K \ p	2	4	8	16	24	32
21	8,47	26	1085	1085	1233	1808
22	$< 0,48$	0,86	2,08	5,36	7,28	35
23	8,58	8,72	8,81	13,18	15,42	24,05
24	1,24	1,24	1,22	1,63	2,12	1,87


K=21
1,500
1,000
500
0
$S(K,p)$
0
5
10
15
20
25
30
35
p


K=22
K=23
30
20
10
0
$S(K,p)$
0
5
10
15
20
25
30
35
p



Here comes the tricky part with speed-up:

Problem is **different magic squares need different time to find them**.

Sequential algorithm finds different magic square for different initial configuration. For K growing as general pattern time is also growing, but with perturbations. If you increase K by 1, it doesn't mean that always time will grow. It depends on distance between closest magic square and initial configuration in search tree of magic squares.

Same thing holds for parallel algorithm, but here we also have different search techniques involved. Master generates jobs using breadth-first search, then slaves use depth-first. While sequentially we use only depth-first search. That's why sometimes we can obtain superliniar speed-up or even speed-up less than 1. But as general pattern, as p grows T(K,p) decreases. And as K grows T(K,p) increases.

4. **Communication subsystem:**

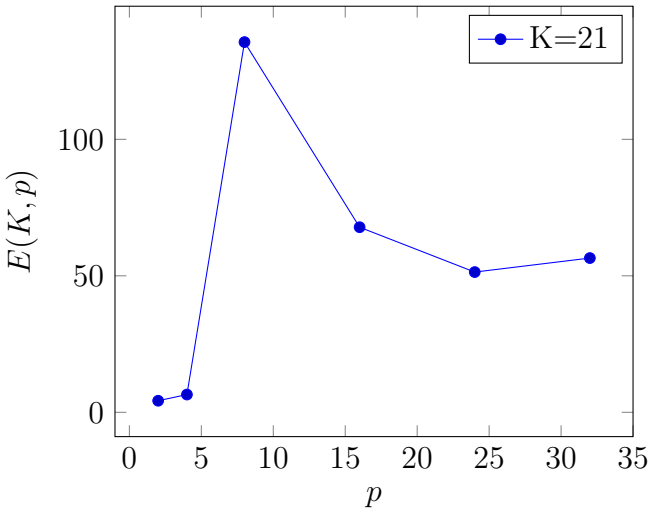
   Message size $\mu = O(N^2)$ Job is usually half filled NxN matrix, zero-jobs or zero-response are zero NxN matrices. In worst case communication overhead is number of jobs times size of matrix times 2:

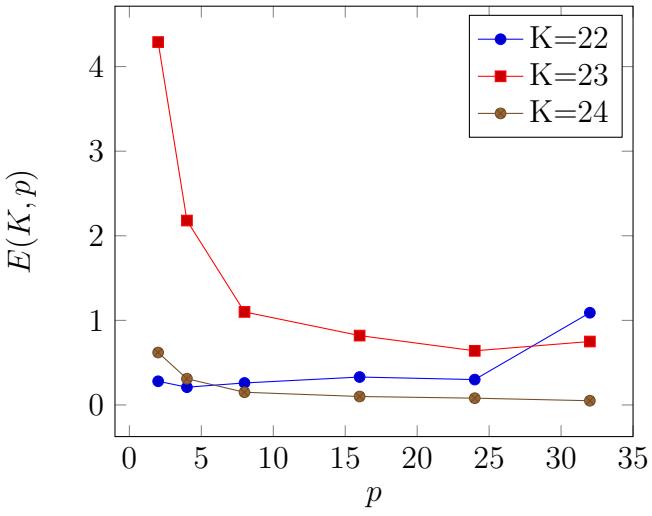
   Communication Overhead $= O(2 * K * (K-1) * N^2)$

   Can be optimised by reducing message size for zero-jobs and zero-responses.

Efficiency E(K,p)

K \ p	2	4	8	16	24	32
21	4.23	6.5	135.62	67.8	51.37	56.5
22	< 0.28	0.21	0.26	0.33	0.30	1.09
23	4.29	2.18	1.1	0.82	0.64	0.75
24	0.62	0.31	0.15	0.1	0.08	0.05





From this table we can conclude that algorithm is not efficient, unless we have luck with input. Efficiency is reduced mainly by property that

different magic squares are found in different times. Depth-first search in sequential algorithm finds first magic square in lexicographic order. Whereas parallel search first uses breadth-first which may significantly slow search.

In order to precisely describe scalability for this problem, one should study distribution of magic squares in total search space and estimate that for given p, N and K we will have luck to wait at most some expected time.

5. If we take a quick look at efficiency table we can find their some maxima in every row that gives us general idea about optimal granularity of this problem. Basically, each row's maxima gives us optimal K and p. Values before these maxima are increasing and after are decreasing. So it's efficient to select these particular values for my particular implementation.

4 Conclusion

This work was an interesting experience for me. I had possibility to create, run and analyse parallel application on real cluster. I've learned how important is to analyse your problem's nature, try to describe it with formulas, find golden ration between input data and computational resources.

5 Bibliography

- "Parallel Systems and Computing" Pavel Tvrdik
- `https://edux.fit.cvut.cz/courses/MI-PPR.2/en/labs/topics_of_semestral_projects`
- `http://users.fit.cvut.cz/~soch/mie-par`