



Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Паралелизација алгоритма за решавање судоку слагалице

Аутор:
Андреа Сабо Цибоља

Индекс:
Е2 91/2022

18. јануар 2023.

Сажетак

Паралелно програмирање има за циљ да искористи све ресурсе на рачунару, како би се смањило време извршавања одређених програма и алгоритама. Овај рад се бави разматрањем могућности увођења паралелизма за решавање проблема судоку слагалице.

Садржај

1	Увод	1
2	Судоку	2
3	Алгоритми за решавање судоку-а	4
3.1	Људски алгоритам	4
3.1.1	Стратегија елиминације	4
3.1.2	Стратегија „усамљени ренџер“	4
3.1.3	Близанци	4
3.1.4	Тројке	5
3.2	„Brute-force“ алгоритам	5
4	Избор технологије за паралелизацију	7
5	Имплементација решења <i>Brute-force</i> алгорита за решавање судоку за- гонтке у програмском језику C	8
5.1	Секвенцијална имплементација	8
5.2	Паралелна имплементација	8
6	Резултати тестирања	13
7	Закључак	15

Списак изворних кодова

1	Функција <i>solve</i> секвенцијално решење	9
2	Функција <i>solve</i> паралелно решење	11

Списак слика

1	Пример судоку загонтеке	2
2	Решење судоку загонетке са слике 1	3
3	Усамљени ренџер	4
4	Близанци	5
5	Погрешно решење без <i>taskwait</i> -а	10

Списак табела

1	Резултати тестирања	13
---	-------------------------------	----

1 Увод

Предности паралелног програмирања се огледају у томе што рачунари могу да извршавају кодове на више ефикасан начин, што може да уштеди време и новац. Паралелно програмирање користи више језгара рачунара истовремено. Како би ово имало ефекта могуће је кодове и алгоритме написати у стилу паралелне архитектуре. За разлику од секвенцијалне архитектуре, паралелна архитектура разбија посао на независне целине како би се оне могле извршавати истовремено. За формирање паралелне архитектуре су често неопходне минималне измене на секвенцијалном коду.

Овај рад бави се имплементацијом „*Brute-force*” алгорита за решавање судоку слагалице. Тестирани су различити начини паралелизације, где је истраживано који би то начин највише убрзао извршавање програма.

2 Судоку

Судоку је логичка загонетна игра у облику квадратне решетке. Решетка је најчешће у формату 9x9, а начињена од подрешетки формата 3x3 што представља регионе. Циљ је да се решетка испуни бројевима од 1 до 9 у свим пољима. Свака колона, сваки ред и свака подрешетка (регион) мора садржати све бројеве од 1 до 9 који се не смеју понављати. За решавање судоку пузле неопходно је само стрпљење и добра логика. На почетку решавања ове слагалице, одређена поља су попуњена, а играч има задатак да попуни остатак тако да испоштује сва правила. Иако је верзија судоку-а која је најпопуларнија управо са димензијама 9x9, постоје и верзије већих димензија како би се повећала комплексност судоку-а. Пример комплексније димензије пузле је 16x16, варијанта која се састоји од 16 колона и 16 врста са подрешеткама димензија 4x4. Још комплексније од овога може да буде димензија судоку-а 25x25 са подрешеткама димензија 5x5, где се решетке попуњавају бројевима од 1 до 25. Код димензије је једино правило да је квадратни корен димензије цео број.

На почеку игре, када се судоку решава, добијају се поједини бројеви на одређеним местима у одређеном реду и врсти. На основу датих бројева, играч има задатак да попуни остатак поља по горе наведеним правилима.

Судоку је настао у Јапану 1986. године, а у Уједињеном Краљевству је постао популаран 2005. године.

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

Слика 1: Пример судоку загонтеке

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

Слика 2: Решење судоку загонетке са слике 1

3 Алгоритми за решавање судоку-а

Постоји више стратегија за решавање судоку слагалице, али је битно напоменути да неки почетни проблеми ни немају решење, а могуће је да поједине судоку слагалице имају и више решења. Тежина решавања судоку слагалице зависи од величине решетке и од броја попуњених поља.

3.1 Људски алгоритам

Људски алгоритам функционише по принципу елиминације вредности. За свако поље се проналазе све могуће вредности које могу ту да се налазе, а затим их елиминишемо примењујући одређене стратегије.

3.1.1 Стратегија елиминације

Ако један исти број може да буде решење више поља, у једном реду, врсти или подрешетци, и ако постоји једно поље где решење може да буде само тај број, закључак је да тај број сигурно треба да стоји на месту где је он једина опција и елиминише се као могућност из свих осталих поља.

3.1.2 Стратегија „усамљени ренцер“

Ако постоји број који је валидан на једном једином месту у реду, врсти или подрешетци, иако на том месту као опција може да се појављује још бројева, узима се он као решење, јер он може да се налази искључиво у том пољу и нигде другде.

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

Слика 3: Усамљени ренцер

3.1.3 Близанци

Близанци у судоку представљају пар од два броја који се појављују само по два пута и то оба пута заједно, тада можемо да елиминишемо све остале опције у тим пољима.

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

Слика 4: Близанци

3.1.4 Тројке

Тројке функционишу по истом принципу као и близанци, само сада посматрамо три боја од којих се сваки јавља као опција у три поља и сваки пут се јављају заједно. Тада можемо да елиминишемо све осим та три броја у та три поља.

Свака од ових стратегија се понавља само једном узастопно. Увек се креће од елиминације, па ако елиминација не реши проблем, покушавамо редом методе усањеног ренџера, па прелажимо на све остале редом, као што пише у [1].

У случају да овај људски алгорита, који функционише по истом принципу као када би сам човек решавао проблем, остави нека поља непопуњена, јер применом овог алгорита није могло никако да се дође до решења, на сцену ступа тзв. „*Brute-force*” алгорита.

3.2 „*Brute-force*” алгорита

Приликом решавања судоку слагалице, на почетку се добија решетка са одређеним бројем попуњених поља. Ради лакшег рада са подацима у улазној датотеци сва непопуњена поља ће бити попуњена са 0.

„*Brute-force*” алгорита функционише тако што поставља прво непопуњено поље на најмању валидну вредност из скупа валидних вредности за то поље, затим ради исто са следећим првим непопуњеним пољем и тако све док постоји могућност за то. Ако на овај начин нађе решење, то је одлично и завршава се са извршавањем програма. Међутим, много је већа вероватноћа да нису све најмање могуће вредности у сваком непопуњеном пољу решење наше судоку слагалице, већ ће некада да дође у ситуацију да ће доћи до поља за које није остало ниједно валидно решење. Када наиђе на прво поље за које не постоји могућност да се попуни ни једном вредношћу, јер се негде нешто не поклапа, алгорита се враћа уназад и претходно поље попуњава првом валидном вредношћу која је већа од броја из претходног покушаја, ако не постоји таква вредност, враћамо се уназад за још један корак (поље). Када дође до првог поља где може да увећа вредност, ту је увећа и попуњава остатак поља по

описаном принципу. Овај алгоритам се изнова и изнова извршава све док постоје слободна поља. Када се дође до решења прекида се извршавање програма.

4 Избор технологије за паралелизацију

При имплементацији, разматрано је коришћење *OpenMP*-а и *OpenMPI*-а.

Оно што *OpenMPI* чини погодним јесте добар начин комуникације између процеса. Ова технологија би могла да буде корисна за имплементацију, с обзиром да би требало након што прва нит дође до решења, да се прекине са извршавањем свих осталих нити.

Међутим, проблем код *MPI*-а би био тај што је синхронизација нити скупа и што би вероватно требало да се ради са верзијом судоку-а која је много већих димензија (више од 100x100) како би се превазишло преоптерећење које настаје самом организацијом. Задачи који се овде решавају су мали и нису толико компликовани да би било потребе за толико комплексном организацијом.

За паралелизацију је изабран *OpenMP*. Оно што је погодно код коришћења *OpenMP*-а јесте што користи дељену меморију, а приликом рада са судоку, ми радимо са истом матрицом и требало би на неки начин да раде нити са истим подацима, све у зависности на који начин се врши паралелизација. Ни *OpenMP* није идеално решење, јер је проблем прекидање осталих нити када једна заврши, јер након што је једна пронашла решење, оно се узима као коначно и нема потреба да остале нити истражују остала могућа решења и уопште даље да се извршавају. *OpenMP* поседује *cancel* механизам, којим једна нит може да иницира прекидање одређеног *omp* блока. Остале нити у обележеним тачкама проверавају да ли је иницирано прекидање, и ако јесте, престају са извршавањем, међутим и ово носи одређене проблеме са собом, као што је наведено у [2].

5 Имплементација решења *Brute-force* алгоритма за решавање судоку загонтке у програмском језику C

5.1 Секвенцијална имплементација

Описана логика решавања судоку загонтке је имплементирана у функцији *solve*. Након учитавања загонетке из датотеке чији се назив прослеђује као аргумент командне линије приликом покретања, позива се функција *solve*. У функцији *solve*, прво што се ради јесте да се проверава да ли има непопуњених поља.

Функција *find_unassigned* проналази прво слободно поље. Вредност врсте и колоне се поставља на позиције где је пронађено то поље, а повратна вредност означава да ли је уопште пронађено такво поље.

У случају да није пронађено непопуњено поље, слагалица је решена, а у случају да јесте настављамо са алгоритмом. Затим, за сваки од бројева који могу да буду у једном пољу (за судоку димензије 9x9, то су бројеви од 1 до 9), проверавамо да ли по правилима судоку-а тај број може да се нађе баш у том пољу, зато постоји линија 11 у коду 1, која позива функцију *is_safe_num*. Она даље проверава помоћу функција *is_exist_row*, *is_exist_col* и *is_exist_box* да ли се прослеђени број већ проналази у врсти, колони и подрешетци у ком се налази прослеђено поље. У случају да се не налази ни на једном месту тај број је могућа опција.

Solve функција затим копира целу матрицу у *copy_grid*, а затим смешта најмањи пронађени број на то место, а затим рекурзивно опет позива саму себе, прослеђујући *copy_grid*. Рекурзија се врти све док се не попуне сва поља, а онда када се то деси испише се решење и прекида се програм.

5.2 Паралелна имплементација

За остваривање паралелизма имплементирано је решење у *OpenMP* технологији.

Код судоку слагалице свако поље зависи од свих вредности у осталим пољима, из тог разлога није најпогодније за паралелизацију, јер када би долазило до ситуације да свака нит мења нешто у матрици, а друга нит чита са тог места, дошло би до великог броја преплитања. Не можемо да гарантујемо нити да утврдимо редослед извршавања нити, из тог разлога свака нит ради са својом копијом судоку матрице.

Најбржа паралелизација је остварена тако што је намештено да различите нити извршавају различите путеве, узимају различита решења као опцију, све док нека

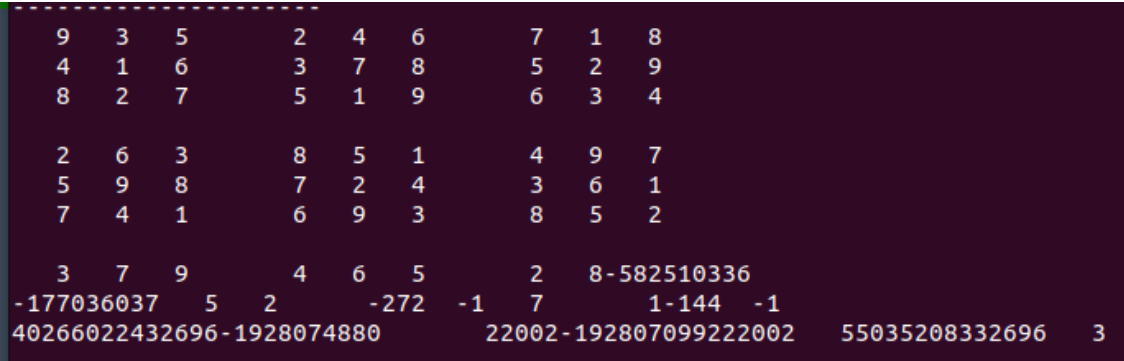
```
1  int solve(int grid[SIZE][SIZE]) {
2
3      int row = 0;
4      int col = 0;
5
6      if (!find_unassigned(grid, &row, &col)){
7          return 1;
8      }
9
10     for (int num = 1; num <= SIZE; num++ ) {
11         if (is_safe_num(grid, row, col, num)) {
12             int val = 0;
13             {
14                 int copy_grid[SIZE][SIZE];
15                 for (int row = 0; row < SIZE; row++) {
16                     for (int col = 0; col < SIZE; col++) {
17                         copy_grid[row][col] = grid[row][col];
18                     }
19                 }
20
21                 copy_grid[row][col] = num;
22                 val = solve(copy_grid);
23
24                 if(val) {
25                     print_grid(copy_grid);
26                     end = clock();
27                     double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
28                     printf("\nProgram se izvrsavao %f sekundi.\n", time_spent);
29                     exit(0);
30                 }
31             }
32
33             grid[row][col] = UNASSIGNED; //back to unassigned because ne
34         }
35     }
36
37     return 0;
38 }
```

Изворни код 1: Функција *solve* секвенцијално решење

нит не дође до резултата, након чега долази до прекидања програма позивом *exit* функције.

Исписом у „*logs.txt*“ видимо да се нити наизменично смењују.

На крају *for* петље стоји „*#pragma omp taskwait*“ како би се сачекало са извршавањем задатака. У случају да се то не стави, проблем је што се извршање јеног задатка неће завршити до краја, а други ће већ почети са истом нити, а пошто иако они раде са приватним променљивама, они их деле међу собом, иста нит, иста приватна променљива, долази до бркања података, па се дешава да нам уместо очекиваног резултата резултат буде нешто као на слици 5, а програм се много брже завршава, јер поља бивају попуњена неодговарајућим вредностима. Нит не заврши до краја са копирањем *grida* у *copy_grid* због тога се јављају насумичне неиницијализоване вредности из меморије. Овај позив „*#pragma omp taskwait*“ баш мора да стоји у 35. линији функције *solve* у коду 2, након завршетка *for* петље, јер када би се налазио унутар *for* петље, проблем би био што би за креирање новог задатка било неопходно да се претходно креирани изврши, не би се користиле све нити и програм би се понашао као секвенцијални.



```

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

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

  3  7  9      4  6  5      2  8-582510336
-177036037  5  2      -272  -1  7      1-144  -1
40266022432696-1928074880      22002-192807099222002      55035208332696  3

```

Слика 5: Погрешно решење без *taskwait*-а

Најбрже се показало решење, где смо убацили нивое у наш алгоритам (*level*) где сваки пут када алгоритам улази корак даље (поље више) наш ниво се повећава. Захваљујући овим нивоима нећемо правити нове задатке сваки подниво, јер би их било превише и паралелизација не би била исплатива, нивоа има онолико колико има слободних поља. Најбрже су се показала решења где смо нове задатке креирали само за први ниво, што се остварује помоћу навођења *final* у 11. линији кода у коду 2. У случају изостављања *final*-а, код ће се извршавати четири пута спорије у односу на

```
1  int solve(int grid[SIZE][SIZE], int level, FILE* f) {
2      int row = 0;
3      int col = 0;
4      if (!find_unassigned(grid, &row, &col)){
5          return 1;
6      }
7      int correct = 0;
8      for (int num = 1; num <= SIZE; num++ ) {
9          if (is_safe_num(grid, row, col, num)) {
10             int val = 0;
11             #pragma omp task firstprivate(grid, row, col, val, num, level)
12             {
13                 int copy_grid[SIZE][SIZE];
14                 for (int row = 0; row < SIZE; row++) {
15                     for (int col = 0; col < SIZE; col++) {
16                         copy_grid[row][col] = grid[row][col];
17                     }
18                 }
19                 copy_grid[row][col] = num;
20                 val = solve(copy_grid, ++level, f);
21                 if(val) {
22                     #pragma omp critical
23                     {
24                         correct=1;
25                         print_grid(copy_grid);
26                         end = clock();
27                         double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
28                         printf("\nIzvrstava se %f s\n", time_spent);
29                         exit(0);
30                     }
31                 }
32             }
33         }
34     }
35     #pragma omp taskwait
36     grid[row][col] = UNASSIGNED;
37
38     return correct;
39 }
```

Изворни код 2: Функција *solve* паралелно решење

секвенцијални за матрицу формата 9x9. (Видети поглавље са резултатима)

Овако, ако рецимо прво празно поље има пет могућности, креираће се пет задатака. Ако радимо са четири нити прво ће *taskwait* омогућити рад са четири задатка одједном, онда ако се један од њих заврши тако да није пронашао решење, креираће се и пети задатак и извршаваће се све док не дође до решења, тј. све док нека нит не дође до попуњене слагалице, тада се прекида извршавање програма, а решење се исписује на конзолу. Задаци се креирају унутар секција *#pragma_omp_single* како би само једна нит вршила поделу проблема на задатке.

6 Резултати тестирања

У овом поглављу приказани су резултати тестирања у погледу времена извршавања у секундама. Покретани су различити примери. Примери се разликују по величини слагалице и/или броју непопуњених поља. Приликом различитих покретања мењан је број нити који се користи и ниво до ког се креирају задаци приликом паралелног извршавања.

Резултати тестова налазе се у табели 1.

Димензије	Непопуњена поља	Ниво	Број нити	Секвенцијално	Паралелно
9x9	64/81(79,01%)	1	3	13.79	1.9
9x9	64/81(79,01%)	2	3	13.79	4.63
9x9	64/81(79,01%)	2	6	13.79	15.83
9x9	64/81(79,01%)	1	4	13.79	2.9
9x9	64/81(79,01%)	1	6	13.79	6.21
16x16	158/256(61,72%)	1	3	523.12	342.14
16x16	158/256(61,72%)	2	3	523.12	1057.44
16x16	158/256(61,72%)	2	6	617.8.79	1545.83
16x16	158/256(61,72%)	1	4	617.8	539.21
16x16	158/256(61,72%)	1	4	617.8	624.01
16x16	128/256(48,83%)	1	3	0.007	0.02
25x25	191/625(30,56%)	1	3	131.14	227.88
25x25	165/625(26,4%)	1	3	620.45	694.28
25x25	165/625(26,4%)	1	4	620.45	574.75
25x25	165/625(26,4%)	1	6	620.45	755.58
25x25	165/625(26,4%)	2	6	620.45	807.41

Табела 1: Резултати тестирања

Може се приметити да се за повећање броја нити не добија нужно и брже решење, јер за велики број нити дође и до успорења, што је често последица времена које је утрошено на стварање и гашење нити, дешава се да чак и секвенцијално решење буде краће него паралелно, све у зависности од величине слагалице, тежине слагалице, броја нити, као и ниво до ког паралелизујемо. Битно је напоменути да тежина слагалице не зависи само од броја непопуњених поља, већ и од распореда како су посласана попуњена поља.

За димензију слагалице 9x9 најбоње се показало решење где се покрећу три, а креирање нових задатака се врши само на првом нивоу. Гледајући резултате, није

могуће са сигурношћу да утврдимо никаква правила, нити када је корисно користити паралелизацију и колико нити доноси боље решење, јер све зависи од индивидуалног случаја загонетке.

7 Закључак

Тема овог рада је истраживање могућности паралелизације „*brute-force*” алгорита за решавање судоку загонетке. На крају истраживања је успостављено да паралелизација може да допринесе бољим перформансама извршавања алгорита до одређене мере, али некада не може и да све зависи од комплексности и тежине саме загонетке. Додатно би могло да се тестира и *OpenMPI* решење, које овде није разматрано као опција.

Библиографија

- [1] Sruthi Sankar. Parallelized sudoku solving algorithm using openmp, 2014.
- [2] Петар Трифуновић. Паралелизација налажења најдужега заједничког подниза. Master's thesis, Факултет техничких наука Нови Сад, 2022.