



Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

---

# Паралелизација алгоритма за решавање судокуа

---

*Аутор:*  
Александар Стојановић

*Индекс:*  
Е2 119/2023

16. јануар 2024.

**Сажетак**

Судоку је логичка игра где је циљ играча да попуни матрицу димензија  $n \times n$  тако да сваки ред, колона и блок садрже сваки број из скупа  $\{1, 2, \dots, n\}$  тачно једном. У овом раду покушана је паралелизација алгорита за решавање ове загонетке користећи *OpenMP* примитиве и *task*-ове, као и *OpenMPI*. На крају се испоставило да паралелне имплементације за матрице малих димензија дају знатно лошије перформансе у поређењу са секвенцијалном имплементацијом.

## Садржај

<b>1</b>	<b>Увод</b>	<b>1</b>
<b>2</b>	<b>Основни појмови и постојећа решења</b>	<b>2</b>
2.1	Пропагирање ограничења . . . . .	2
2.2	Претрага простора могућих решења . . . . .	2
<b>3</b>	<b>Секвенцијална имплементација</b>	<b>4</b>
3.1	Структуре података . . . . .	4
3.2	Пропагирање ограничења . . . . .	4
3.3	Претрага простора могућих решења . . . . .	8
<b>4</b>	<b>Паралелна имплементација</b>	<b>10</b>
4.1	<i>OpenMP</i> имплементација . . . . .	10
4.2	<i>OpenMP</i> имплементација употребом <i>task</i> -ова . . . . .	12
4.3	<i>OpenMPI</i> имплементација . . . . .	14
4.3.1	Структуре података . . . . .	14
4.3.2	Хијерархија процеса и начин рада . . . . .	15
<b>5</b>	<b>Поређење перформанси</b>	<b>18</b>
<b>6</b>	<b>Закључак</b>	<b>20</b>

## Списак изворних кодова


1	Неопходне структуре података . . . . .	4
2	Пропагирање ограничења за ћелију . . . . .	6
3	Имплементација примене другог правила . . . . .	7
4	Алгоритам претраге простора могућих решења . . . . .	9
5	<i>OpenMP</i> имплементација претраге . . . . .	11
6	<i>OpenMP</i> имплементација претраге употребом <i>task</i> -ова . . . . .	13
7	Додатне структуре података . . . . .	14
8	Иницијална пропација ограничења . . . . .	15
9	Пример конструкције за асинхроно ослушкивање порука . . . . .	16
10	Делегирање дела посла слободним процесима . . . . .	17

**Списак слика**

1	Пример нерешеног и решеног судокуа . . . . .	1
2	Превенција узалудне претраге стабла уз помоћ паралелизације . . . .	10
3	Резултати секвенцијалне и <i>MP</i> имплементације . . . . .	18
4	Резултати <i>MPI</i> имплементације . . . . .	19

## 1 Увод

Судоку је популарна загонетка са бројевима која се среће у многобројним дневним новинама и часописима. Директан превод са јапанског имена ове загонетке је "јединствен број" што и осликава задатак ове игре, формирати квадратну матрицу димензија  $n \times n$  (најчешће  $9 \times 9$ ) тако да се у свакој колони и врсти може затећи само једна појава бројева од 1 до  $n$ . Исто правило примењује се и на одређене подматрице чије су димензије  $\sqrt{n} \times \sqrt{n}$ . Играчу је на почетку игре дата матрица са неколико већ додељених вредности, а његов задатак је да матрицу попуни до краја, а да не прекрши правило јединствености појављивања бројева. Пример нерешене, а затим успешно решене загонетке може се видети на слици 1.



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

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

Слика 1: Пример нерешеног и решеног судокуа

Циљ овог рада је паралелизација алгорита за решавање ове загонетке. У даљим поглављима биће додатно појашњени основни појмови потребни за разумевање алгорита, постојећа решења, а затим и ауторове имплементације овог алгорита различитим приступима (*OpenMP* примитиви и *task*-ови, *OpenMPI*), као и поређење њихових перформанси.

## 2 Основни појмови и постојећа решења

Да би разумевање рада алгоритама за решавање ове загонекте било лакше, потребно је увести неколико основних појмова:

- Појединачно поље у које се може сместити број назива се **ћелија**.
- Ћелија која у датом тренутку решавања има само један број који је могуће уписати, без да се наруше ограничења, назива се *singleton*.
- Подматрица димензија  $\sqrt{n} \times \sqrt{n}$  која у себи мора садржати свих  $n$  бројева назива се *box*.
- *Peer*-ом се назива свака ћелија која се налази у истој врсти, колони или *box*-у (било која од ове три групације генерално се може називати *unit*) у односу на посматрану ћелију.

Оно што је заједничко за све постојеће радове [2, 6, 7] као и ауторову имплементацију је то да се решавање загонетке заснива на два алгорита - пропагацији ограничења и претрази простора могућих решења.

### 2.1 Пропагирање ограничења

У уводном поглављу споменуто је основно ограничење загонетке које захтева да се унутар сваког *unit*-а морају наћи сви бројеви и то тачно једном. Приликом решавања загонетке, ово ограничење се може рашчланити на два правила:

- Ако се у некој од ћелија *unit*-а посматране ћелије већ налази неки број, тај број се уклања из листе могућих бројева посматране ћелије.
- Ако све ћелије *unit*-а посматране ћелије сем ње имају одређен број избачен из њихове листе могућих бројева, тај број припада посматраној ћелији.

Узевши ова правила у обзир први корак решавања алгорита била би њихова максимална примена широм целе матрице, све док се не дође до ситуације када још увек постоје празне ћелије, али немогуће их је попунити само на основу задатих правила.

### 2.2 Претрага простора могућих решења

Овај алгоритам полази од максимално попуњене матрице на основу правила из 2.1, затим користећи се скупом могућих бројева неке ћелије нагађа који би број могао стајати у њој и након тога наставља са пропагацијом ограничења, све док поново не

дође до ситуације када мора да нагађа, ситуације када долази до крајњег решења или до контрадикторне ситуације. У случају проналаска решења алгоритам се овде завршава, а у случају контрадикторне ситуације алгоритам се враћа до последњег валидног стања матрице и покушава са новим бројем. Може се приметити да је овај алгоритам класичан пример *back tracking* алгорита [1].



### 3 Секвенцијална имплементација

У овом поглављу биће описана секвенцијална имплементација, на којој ће се темељити и све наредне паралелне имплементације.

#### 3.1 Структуре података

На почетку потребно је моделовати појединачну ћелију судоку матрице, као и читаву матрицу 1. Ћелија је представљена као структура која од поља садржи коначну вредност (*value*), која може варирати од 1 до  $n$ , као и поље *possibilities* које ће се, ради уштеде меморијског простора, третирали као бинарни низ могућих вредности ћелије, где јединица на одређеном биту означава могућност појаве броја. На пример у случају 9x9 матрице број 0b1\_1010\_0010 значио би да су могуће вредности те ћелије бројеви 2, 6, 8 и 9. Читава матрица моделована је као дводимензионални низ ћелија. Уз ове структуре података имплементирани су и тривијалне помоћне методе за учитавање и писање матрице у *csv* фајл, испис матрице на екран, копирање, као и проверу да ли је судоку решен тако што се проверава да ли све ћелије имају вредност у опсегу од 1 до  $n$ , као и да им је низ могућих бројева једнак нули (0b0\_0000\_0000).

```
1 typedef struct {
2     int value;
3     unsigned possibilities;
4 } Cell;
5
6 typedef Cell* CellArray;
7 typedef CellArray* SudokuGrid;
8
```

Изворни код 1: Неопходне структуре података

#### 3.2 Пропагирање ограничења

Срж имплементације налази се у *solve* функцији, која се позива након учитавања нерешене судоку матрице. Као што је претходно поменуто, она се есенцијално састоји из два алгорита, алгорита за пропагирање ограничења и алгорита за претрагу простора могућих решења.

Пропагирање решења почиње применом претходно наведена 2 правила (2.1) над свим *singleton*-има, а затим се у бесконачној петљи наизменично врше две акције.

У првој се проналазе новонастали *singleton*-и, додељује им се једини могући број и затим пропагирају ограничења која су произведена додавањем броја у ћелију. У другој акцији примењује се друго правило, где се за одређену ћелију проверава да ли постоји број који је избачен из низа потенцијалних бројева у свакој од ћелија *unit*-а посматране ћелије и ако је то случај ћелији се додељује тај број и пропагирају новонастала ограничења. Петља се прекида у тренутку када се нађе коначно решење (ово се може десити у лакшим загонеткама) или док се не примети да не постоји разлика у насталим судоку матрицама у претходне две итерације, што би значило да су ограничења максимално пропагирана и да је дошло време за претрагу простора могућих решења. Може се такође догодити ситуација када се пропагирањем ограничења дође у контрадикторно стање, у ком случају *solve* функција враћа *NULL* као решење. Овакво понашање битно је за наредни алгоритам.

---

```
1  bool propagateRulesForCell(SudokuGrid grid, int row, int
   ↪ col) {
2
3      //Check for contradiction box
4      int sudokuSizeRoot = (int) sqrt(SUDOKU_SIZE);
5      int horizontal_start = (row / sudokuSizeRoot) *
   ↪ sudokuSizeRoot;
6      int vertical_start = (col / sudokuSizeRoot) *
   ↪ sudokuSizeRoot;
7      for(int i = horizontal_start; i < horizontal_start +
   ↪ sudokuSizeRoot; i++){
8          for(int j = vertical_start; j < vertical_start +
   ↪ sudokuSizeRoot; j++){
9              if( i == row && j == col) { continue;}
10             if(grid[i][j].value == grid[row][col].value){
11                 return false;
12             }
13         }
14     }
15
16     //Remove from box unit
17     for(int i = horizontal_start; i < horizontal_start +
   ↪ sudokuSizeRoot; i++){
18         for(int j = vertical_start; j < vertical_start +
   ↪ sudokuSizeRoot; j++){
19             grid[i][j].possibilities =
20                 subtractPossibility(
21                     grid[i][j].possibilities,
22                     possibility_convertToBin(
23                         grid[row][col].value));
24         }
25     }
26
27     //. . .
28     // Na sličan način propagiranje se vrši i po vrstama i
   ↪ kolonama
29     // (nije prikazano zbog preglednosti)
30     //. . .
31
32     return true;
33 }
34
```

---

---

```
1 //By doing ~ you get all forbidden ones for cell
2 //Combining all forbidden of one unit with & you get common
  ↳ forbidden for unit
3 //After that you see which of forbidden, are possible for
  ↳ current column by doing &
4 //of all forbidden and possible of that column
5 //If you get just one possible return it
6 int getCommonForbiddenForUnit(SudokuGrid grid, int row, int
  ↳ col){
7     if(grid[row][col].value != 0){
8         return 0;
9     }
10
11     //. . .
12     // Na sličan način propagiranje se vrši i po vrstama i
      ↳ kolonama
13     // (nije prikazano zbog preglednosti)
14     //. . .
15
16     //Box
17     result = ALL_POSSIBILITIES_VAL;
18     int sudokuSizeRoot = (int) sqrt(SUDOKU_SIZE);
19     int horizontal_start = (row / sudokuSizeRoot) *
      ↳ sudokuSizeRoot;
20     int vertical_start = (col / sudokuSizeRoot) *
      ↳ sudokuSizeRoot;
21     for(int i = horizontal_start; i < horizontal_start +
      ↳ sudokuSizeRoot; i++){
22         for(int j = vertical_start; j < vertical_start +
      ↳ sudokuSizeRoot; j++){
23             if(row == i && col == j) { continue; }
24             unsigned forbidden = ~grid[i][j].possibilities;
25             result = result & forbidden;
26         }
27     }
28
29     result = result & grid[row][col].possibilities;
30     if(countOnes(result) == 1){
31         return possibility_convertToDec(result);
32     }
33
34     return 0;
35 }
36
```

---

Да се приметити да одлука за бинарну представу могућих бројева ћелије, поред уштеде меморијског простора, омогућава и ефикасније баратање низовима у овим случајевима коришћења где су итерације кроз низ у потпуности замењене бинарним операцијама  $O(1)$  комплексности. У случају коришћења проналаска броја који је избачен из низа могућих бројева свих чланова *unit*-а, сем из низа посматране ћелије 3 прво се свим низовима уради негација и тако добију сви забрањени бројеви, затим се комбиновањем свих низова помоћу бинарног "и" исфилтрирају бројеви и добију само они који су забрањени свим ћелијама. На крају се исфилтрирани низ забрањених бројева искомбинује са низом довољених бројева посматране ћелије, такође помоћу бинарног "и" и на тај начин добију бројеви који се могу уписати у ћелију. Ако се у низу налази само један број, он се одмах може уписати.

### 3.3 Претрага простора могућих решења

Након што се максимално испропагирају ограничења, прелази се на претрагу простора могућих решења 4. Да нагађање не би било потпуно насумично, проналази се "оптимални кандидат" за нагађање, то јест ћелија која има најмањи број потенцијалних бројева. На овај начин минимизује се број гранања, а тиме и укупно време претраге у најгорем случају. Затим се пролази кроз сваки могући број посматране ћелије, додељује се ћелији и са тако измењеном судоку матрицом рекурзивно се позива *solve* функција. Као што је поменуто у 3.2, рекурзивна претрага наставља се све док се не дође до коначног решења или у случају доласка у контрадикторно стање матрице, где се онда алгоритам враћа до последњег валидног стања и даље покушава са другим бројем.

---

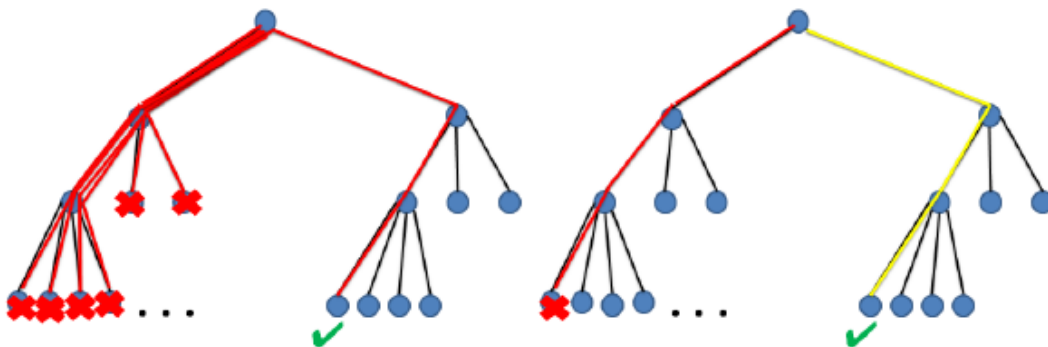
```
1  int optimal_i, optimal_j;
2  findOptimalCandidate(grid, &optimal_i, &optimal_j);
3
4  int* possibilities;
5  int possibilities_length;
6  extractPossibilities(grid[optimal_i][optimal_j].possibilities,
   ↪  &possibilities, &possibilities_length);
7
8  for(int i = 0; i < possibilities_length; i++){
9      SudokuGrid guessGrid = copySudokuGrid(grid);
10     guessGrid[optimal_i][optimal_j].value =
   ↪     possibilities[i];
11     guessGrid[optimal_i][optimal_j].possibilities = 0;
12
13     SudokuGrid result = solve(guessGrid);
14     if(result == NULL){
15         //NULL means that this guess consequently violated
   ↪     rules
16         continue ;
17     }else{
18         return result;
19     }
20 }
21
```

---

Изворни код 4: Алгоритам претраге простора могућих решења

## 4 Паралелна имплементација

Анализом претходно описане секвенцијалне имплементације 3, може се закључити да је готово немогуће паралелизовати део алгорита који се бави пропацирањем ограничења, због тога што би свака промена ћелије захтевала синхорнизацију свих нити/процеса ради коректне даље пропације ограничења, што би резултовало *lock-step* паралелизацијом нити са великом фреквенцијом усклађивања. Са друге стране, алгоритам претраге простора могућих решења због свог гранања на више независних претрага у дубину стабла могућих решења природно подлеже паралелизацији. Свака нит би модификовала судоку матрицу на другачији начин и тако започела своју претрагу за решењем и када би нека нит пронашла решење сигнализирила би крај претраге. На овај начин избегава се ситуација када би алгоритам узалудно претраживао велики део стабла због лоше почетне претпоставке 2, а и свакако се убрзава сама претрага.



Слика 2: Превенција узалудне претраге стабла уз помоћ паралелизације

### 4.1 *OpenMP* имплементација

За разлику од постојећих решења која су се користила *pthread*s [5] библиотеком, у овом пројекту је као алат за имплементацију изабран *OpenMP* [3] ради декларативнијег приступа имплементацији.

---

```

1      #pragma omp parallel num_threads(sudokuSizeSqrt)
      ↪ shared(finalResult, stopFlag)
2      {
3          #pragma omp for
4          for (int i = 0; i < possibilities_length; i++) {
5              //Check if some thread found solution
6              if (finalResult != NULL || *stopFlag) {
7                  continue;
8              }
9
10             SudokuGrid guessGrid = copySudokuGrid(grid);
11             guessGrid[optimal_i][optimal_j].value =
              ↪ possibilities[i];
12             guessGrid[optimal_i][optimal_j].possibilities =
              ↪ 0;
13
14             if (finalResult != NULL || *stopFlag) {
15                 continue;
16             }
17             SudokuGrid result = solve(guessGrid, stopFlag);
18             if (result == NULL) {
19                 //NULL means that this guess consequently
              ↪ violated rules
20                 continue;
21             } else {
22                 #pragma omp critical
23                 {
24                     *stopFlag = true;
25                     finalResult = result;
26                 }
27             }
28         }
29     }
30

```

---

Изворни код 5: *OpenMP* имплементација претраге

За потребе паралелизације, петља унутар *solve* функције која је служила за рекурзивну претрагу морала је подлећи одређеним модификацијама. Помоћу *MP* ди-



ректива, петља је паралелизована, тако да сада свака њена итерација која испробава један модел решења може да се извршава на засебној нити. Уместо да се резултат решења директно враћа из петље, сада се јавила потреба за дељеном променљивом *finalResult* којој се приступа унутар критичне секције и уписује решење уколико се пронађе. Да би се одмах након проналаска решења прекинуо рад и свих осталих нити, уводи се још једна дељена променљива *stopSignal*, чију вредност нит која је дошла до решења мења на *true* и тако сигнализира свим осталим нитима да истог тренутка прекину са радом.

## 4.2 *OpenMP* имплементација употребом *task*-ова

Ова имплементација концептуално је слична *MP* имплементацији преко примитива 4.1. Разлика је у томе што сада само једна нит унутар петље креира *task*-ове који се затим распоређују по свим нитима унутар паралелног региона 6. Начин пропагације коначног решења и прекида рада свих осталих *task*-ова остаје непромењен.

---

```
1  #pragma omp parallel num_threads(sudokuSizeSqrt)
   ↪  shared(finalResult, stopFlag)
2  {
3      #pragma omp single nowait
4      for (int i = 0; i < possibilities_length; i++) {
5          // Create tasks for each possibility
6          #pragma omp task
7          {
8              // Check if some task found a solution
9              if (!(finalResult != NULL || *stopFlag)) {
10
11                  SudokuGrid guessGrid = copySudokuGrid(grid);
12                  guessGrid[optimal_i][optimal_j].value =
13                      ↪  possibilities[i];
14                  guessGrid[optimal_i][optimal_j].possibilities
15                      ↪  = 0;
16
17                  SudokuGrid result = solve(guessGrid,
18                      ↪  stopFlag);
19
20                  #pragma omp critical
21                  {
22                      if (result != NULL) {
23                          *stopFlag = true;
24                          finalResult = result;
25                      }
26                  }
27              }
28          }
29  }
```

---

Изворни код 6: *OpenMP* имплементација претраге употребом *task*-ова

### 4.3 *OpenMPI* имплементација

Будући да се начин функционисања *OpenMPI*-а [4] разликује од *OpenMP*-а, ова имплементација се знатно разликује од све три претходне. Оно што разликује ову *OpenMPI* имплементацију од постојећих је покушај делегирања посла једног процеса на све расположиве процесе, за разлику од тренутних имплементација где би процес био у могућности да половину свог посла делегира само суседном процесу (посматрајући *id* процеса).

#### 4.3.1 Структуре података

Поред постојећих структура података за ћелију и читаву судоку матрицу додат је низ *unsigned* вредности *availabilities* величине једнаке броју процеса који води рачуна о заузетости сваког од процеса. Такође било је потребно дефинисати нове *MPI* типове података за размену ћелија и информација о доступности процеса између процеса 7, као и написати помоћне функције за постављање судоку матрице у континуалан део меморије како би се она могла размењивати између процеса.

---

```
1 //Defining data types
2 MPI_Datatype cell_type;
3 MPI_Type_contiguous(2, MPI_UNSIGNED, &cell_type);
4 MPI_Type_commit(&cell_type);
5
6 MPI_Datatype availability_type;
7 MPI_Type_contiguous(2, MPI_UNSIGNED, &availability_type);
8 MPI_Type_commit(&availability_type);
9
10 unsigned availabilities[num_proc];
11 for(int i = 0; i < num_proc;i++){
12     availabilities[i] = true;
13 }
14
15
16
```

---

Изворни код 7: Додатне структуре података

### 4.3.2 Хијерархија процеса и начин рада

Читав рад заснива се на бесконачној петљи у којој се сви процеси налазе и чекају да им посао буде делегиран. Изузетак је процес са ранком 0, који је проглашен координатором који се бави вођењем евиденције о заузетости процеса и обавештавањем других процеса о тој информацији, као и очекивању коначног решења и финализацији алгоритма. Алгоритам почиње тако што координатор процес покушава пропацију ограничења и ако из првог покушаја успе да реши загонетку одмах завршава програм, у супротном судоку матрицу са до тог тренутка максимално прорпагираним ограничењима прослеђује као задатак за решавање процесу 1 8.

---

```

1 // Load and initial constraint propagation
2 if(rank == 0){
3     //Load
4     SudokuGrid grid = loadSudokuGrid(argv[1]);
5
6     SudokuGrid result = constraintPropagation(grid);
7
8     if(solved(result)){
9         Cell flattened[SUDOKU_SIZE*SUDOKU_SIZE];
10        flattenSudokuGrid(result, flattened);
11        MPI_Isend(flattened, SUDOKU_SIZE*SUDOKU_SIZE,
12                ↪ cell_type, 0, 0, MPI_COMM_WORLD, NULL);
13    } else {
14        //Start processing with process 1
15        //Send job
16        Cell flattened[SUDOKU_SIZE*SUDOKU_SIZE];
17        flattenSudokuGrid(result, flattened);
18        MPI_Send(flattened, SUDOKU_SIZE*SUDOKU_SIZE,
19                ↪ cell_type, 1, 2, MPI_COMM_WORLD);
20    }
21 }
```

---

#### Изворни код 8: Иницијална пропација ограничења

Будући да би сви процеси требали да се баве својим задацима и асинхронно ослушкују да ли им је стигла нека порука, на разним местима у коду налази се слична конструкција која је приказана у коду 9. Сваки процес на почетку петље проверава да ли му је упућена нека порука и уколико јесте на основу њеног тага, одређује да ли

се ради о поруци за завршетак процеса, поруци о прослеђеној матрици (решеној или спремној за обраду) или поруци за освежавање информације о доступности процеса.

---

```
1 // Check if there is an incoming message
2 int message_flag;
3 MPI_Status status;
4 MPI_Iprobe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &message_flag,
   ↪ &status);
5
6 if (message_flag) {
7     int tag = status.MPI_TAG;
8
9     if (tag == 2) {
10         //... Do something...
11     } else if (tag == 1) {
12         // Stop signal received
13         break;
14     }
15 }
16
```

---

#### Изворни код 9: Пример конструкције за асинхроно ослушкивање порука

У код *solve* методе коју позивају процеси када добију матрицу за обраду додато је неколико нових делова. На самом почетку ажурира се листа доступних процеса за делегирање посла и након пропагације ограничења пре него што се уђе у петљу за претрагу простора могућих ограничења процес прво делегира део посла свим доступним процесима<sup>10</sup>. Након делегације посла, процес ће истражити преостале случајеве који нису делегирани. У случају проналаска решења оно се шаље директно координаторском процесу и завршава се програм.

---

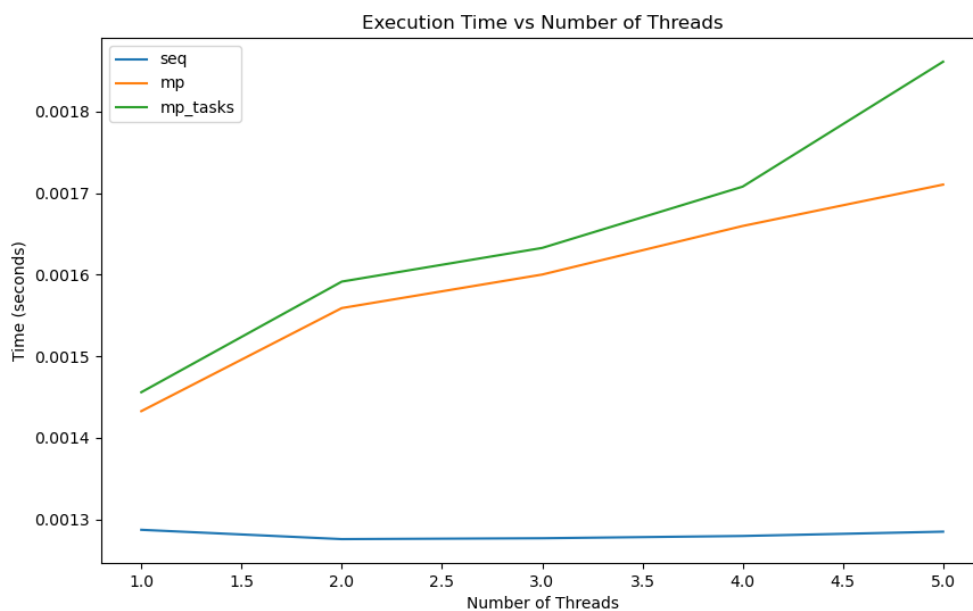
```
1 //Delegate work
2 if(possibilities_length > 1){
3     for(int i = 1; i < num_proc;i++){
4         if(i == rank) { continue;}
5         if(availabilities[i]){
6             SudokuGrid guessGrid = copySudokuGrid(result);
7             guessGrid[optimal_i][optimal_j].value =
8                 ↪ possibilities[delegated];
9             guessGrid[optimal_i][optimal_j].possibilities =
10                 ↪ 0;
11             availabilities[i] =false;
12
13             Cell flattened[SUDOKU_SIZE*SUDOKU_SIZE];
14             flattenSudokuGrid(guessGrid, flattened);
15             MPI_Send(flattened, SUDOKU_SIZE*SUDOKU_SIZE,
16                 ↪ cell_type, i, 2, MPI_COMM_WORLD);
17             delegated++;
18         }
19     }
20 }
```

---

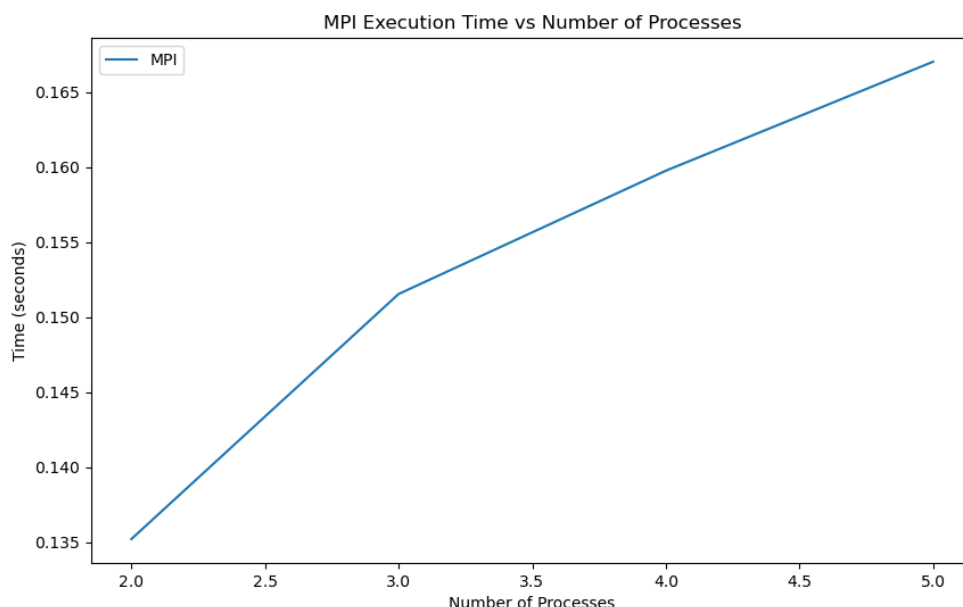
Изворни код 10: Делегирање дела посла слободним процесима

## 5 Поређење перформанси

За мерење перформанси кориштени су примери судоку загонетки најтежег нивоа (*evil*) димензија 9x9 и 16x16 и да би се додатно испровоцирао алгорита за претрагу простора постојећих решења из тих примера обрисани су неки од бројева уз проверу да се њиховим брисањем није онемогућило решавање загонетке. Током тестирања мењан је број кориштених нити и процеса да би се уочио утицај паралелизације на брзину извршавања алгорита. Модел процесора који је кориштен за тестирање је *AMD Ryzen 7 7730U (base 2.0GHz, max 4.5GHz)*. Добијени су следећи резултати 34.



Слика 3: Резултати секвенцијалне и *MP* имплементације

Слика 4: Резултати *MPI* имплементације

На основу резултата да се уочити да је упркос разним покушајима паралелизације алгоритма за претрагу простора могућих решења, ипак секвенцијално решење најбрже и да се повећавањем броја нити и процеса само повећава *overhead* потребан за координацију нити и процеса који додатно повећава време извршавања. Утицај на брзину секвенцијалног алгоритма највероватније има оптимизација алгоритма за пропацију ограничења (проналажење оптималног кандидата и коришћење бинарног низа за складиштење могућих бројева), као и модернији модел процесора (мало мањи удео). Огромно одступање брзине *MPI* имплементације оправдано је великим *overhead*-ом који је потребан за координацију процеса, као и чинњеницом да се судоку матрица мора сваки пут трансформисати за потребе међусобне комуникације процеса, а њена величина није занемарљива.

Код *OpenMP* имплементација може се приметити да је број итерација паралелизоване петље која се користи за претрагу простора могућих решења директно зависан од димензија судоку матрице, те је у примеру 9x9 матрице и највећи број итерација који се може јавити 9 што и не представља неки велики број итерација чак ни за секвенцијалну имплементацију. Могуће је да би *OpenMP* имплементација имала боље перформансе у поређењу са секвенцијалном имплементацијом када би димензије матрице биле много веће, јер би у случају погрешне почетне претпоставке секвенцијална имплементација потрошила много времена док не исцрпи



све случајеве и пређе на наредну претпоставку док би паралелна имплементација истовремено покушавала све претпоставке и прекинула извршавање по проналаску тачног решења (детаљније објашњено у поглављу 4).

## 6 Закључак

У овом раду изучени су алгоритми који се користе за решавање судоку загонетке произвољне тежине (пропагација ограничења, претрага простора могућих решења), као и одређени приступи њиховој оптимизацији (бинарна представа могућих бројева ћелије, проналажење оптималног кандидата за даљу претрагу). Фокус је био на паралелизацији алгорита за претрагу простора могућих решења. Проучавање имплементације користиле су се *OpenMP* примитивима и *task*-овима, као и *OpenMPI*-ем и увеле неке нове приступе у односу на постојећа решења. Након мерења перформанси сваке од имплементација, дошло се до закључка да је секвенцијална имплементација ипак најбржа, због оптимизованости алгорита за пропагирање ограничења и највећим делом *overhead*-а који уводи потреба за координацијом нити и процеса.

## Библиографија

- [1] Backtracking. <https://en.wikipedia.org/wiki/Backtracking/>, 2024. Последњи приступ: 14.1.2024.
- [2] Multi threaded sudoku solver. [https://web.archive.org/web/20200229125814id\\_/https://www.researchgate.net/profile/Ebin\\_Scaria/publication/339536682\\_Multi-Threaded\\_Sudoku\\_Solver/links/5e57db73a6fdccbeba07358b/Multi-Threaded-Sudoku-Solver.pdf](https://web.archive.org/web/20200229125814id_/https://www.researchgate.net/profile/Ebin_Scaria/publication/339536682_Multi-Threaded_Sudoku_Solver/links/5e57db73a6fdccbeba07358b/Multi-Threaded-Sudoku-Solver.pdf), 2024. Последњи приступ: 14.1.2024.
- [3] Openmp. <https://www.openmp.org/resources/refguides/>, 2024. Последњи приступ: 14.1.2024.
- [4] Openmpi. <https://www.open-mpi.org/doc/>, 2024. Последњи приступ: 14.1.2024.
- [5] pthreads. [https://docs.oracle.com/cd/E26502\\_01/html/E35303/tlib-1.html](https://docs.oracle.com/cd/E26502_01/html/E35303/tlib-1.html), 2024. Последњи приступ: 14.1.2024.
- [6] Sudoku report. <http://www.individual.utoronto.ca/rafatrashid/Projects/2012/SudokuReport.pdf>, 2024. Последњи приступ: 14.1.2024.
- [7] Using mpi one-sided communication for parallel sudoku solving. <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1766597&dswid=6848>, 2024. Последњи приступ: 14.1.2024.