

Druga domaća zadaća

Igra Connect4

1. Implementacija

Prije svega bitno je napomenuti da sam kao podlogu koristio program dostupan u repozitoriju predmeta (connect4) te sam paralelizirao dio koji evaluira trenutne korake na način da sam dijelio stupce procesima.

Prva prepreka koju sam morao riješiti je inicijalizacija posla u glavnom (rank 0) procesu.

To sam ostvario na sljedeći način:

```
if (rank == 0){
    if(argc<2)
    {
        cout << "Uporaba: <program> <fajl s trenutnim stanjem> [<dubina>]" << endl;
        MPI_Finalize();
        return 0;
    }
}

B.Load(argv[1]);
if(argc>2)
    iDepth = atoi(argv[2]);

if(rank == 0){
    srand( (unsigned)time( NULL ) );
    // provjerimo jel igra vec gotova (npr. ako je igrac pobijedio)
    for(int iCol=0; iCol<B.Columns(); iCol++)
        if(B.GameEnd(iCol))
        {
            cout << "Igra završena!" << endl;
            MPI_Finalize();
            return 0;
        }
}

MPI_Bcast(&iDepth, 1, MPI_INT, 0, MPI_COMM_WORLD);

do
{
    if(rank == 0) cout << "Dubina: " << iDepth << endl;

    vector<SimpleTask> allTasks;

    // Generiraj zadatke (samo na procesu 0)
    if(rank == 0) {
        // Odredimo dubinu za generiranje zadataka - prilagodi broju procesora
        int taskGenDepth = TASK_GENERATION_DEPTH;

        // Povečaj dubinu generiranja ako imamo puno procesora
        if(size >= 8) taskGenDepth = min(taskGenDepth + 1, iDepth - 3);
        if(size >= 16) taskGenDepth = min(taskGenDepth + 1, iDepth - 3);

        // PAZI: generiramo previše ili premalo zadataka
        // dovoljno dubine za Evaluate funkciju
        taskGenDepth = max(1, min(taskGenDepth, iDepth - 4));
        if(taskGenDepth < 1) taskGenDepth = 1;

        cout << "Generiranje zadataka do dubine " << taskGenDepth
            << " za " << size << " procesora..." << endl;

        vector<int> currentMoves;
        GenerateTaskSequences(B, 0, taskGenDepth, currentMoves, allTasks, EMPTY);

        cout << "Generirano " << allTasks.size() << " zadataka" << endl;
    }
}
```

Glavni proces prvo provjerava argumente naredbenog retka, učitava trenutno stanje igre iz datoteke i inicijalizira generator slučajnih brojeva. Ovdje je također vidljivo da se *iDepth* *broadcasta* svim procesima u svrhu sinkronizacije.

Sljedeća prepreka je bila podjela posla ostalim procesima.

Na slici je vidljivo kako je to razrješeno:

```
// Gather rezultata
vector<double> allColumnValues(7 * size, -2.0);
MPI_Gather(columnValues.data(), 7, MPI_DOUBLE,
          allColumnValues.data(), 7, MPI_DOUBLE, 0, MPI_COMM_WORLD);

dBest = -2.0;
iBestCol = -1;

if(rank == 0) {
    for(int p = 0; p < size; p++) {
        for(int col = 0; col < 7; col++) {
            double value = allColumnValues[p * 7 + col];
            if(value > -2.0 && B.MoveLegal(col)) {
                if(iBestCol == -1 || value > dBest ||
                   (value == dBest && rand() % 2 == 0)) {
                    dBest = value;
                    iBestCol = col;
                }
            }
        }
    }
} else {
    // Rad s generiranim zadacima
    if(rank != 0) allTasks.resize(numTasks);

    // Pošaljemo zadatke (jednostavnije - samo sekvence poteza)
    vector<int> taskSizes(numTasks);
    vector<int> allMoves;
    vector<int> originalCols(numTasks);

    if(rank == 0) {
        for(int i = 0; i < numTasks; i++) {
            taskSizes[i] = allTasks[i].moves.size();
            originalCols[i] = allTasks[i].originalColumn;
            for(int move : allTasks[i].moves) {
                allMoves.push_back(move);
            }
        }
    }

    MPI_Bcast(taskSizes.data(), numTasks, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(originalCols.data(), numTasks, MPI_INT, 0, MPI_COMM_WORLD);

    int totalMoves = allMoves.size();
    MPI_Bcast(&totalMoves, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(rank != 0) allMoves.resize(totalMoves);
    MPI_Bcast(allMoves.data(), totalMoves, MPI_INT, 0, MPI_COMM_WORLD);

    // Rekonstruiraj
    if(rank != 0) {
        int moveIndex = 0;
        for(int i = 0; i < numTasks; i++) {
```

Početak je do-while petlja koju smo imali kada smo koristili sekvencijalni algoritam sa istim uvjetima. Ovdje je jasno vidljivo kako radimo prodjelu poslova procesima za recimo 6x7 polje i 4 procesa proces 0 će dobiti na evaluaciju retke 0 i 4, proces 1 će dobiti retke 1 i 5, proces 2 retke 2 i 6, proces 3 redak 3 tj. svi će dobiti u ovom slučaju ili 2 retka ili 1 za evaluaciju što je puno bolje nego da jedan proces radi sve.

Kada je sve izračunato morao sam pronaći način da sva ta rješenja kombiniram i dođem do rješenja.

```
73 int main(int argc, char **argv)
74 do
75     } else {
76         for(int i = rank; i < numTasks; i += size) {
77
78             if(validSequence) {
79                 // vrijednost za ostale dubine
80                 int remainingDepth = iDepth - allTasks[i].moves.size() - 1;
81                 if(debug){
82                     cout << "Proces " << rank << ": Task " << i
83                         << ", moves=" << allTasks[i].moves.size()
84                         << ", remaining_depth=" << remainingDepth << endl;
85                 }
86                 if(remainingDepth > 0) {
87                     // Probaj sve CPU poteze iz ovog stanja
88                     double bestValue = -2.0;
89                     bool foundMove = false;
90
91                     for(int col = 0; col < taskBoard.Columns(); col++) {
92                         if(taskBoard.MoveLegal(col)) {
93                             foundMove = true;
94                             Board tempBoard = taskBoard;
95                             tempBoard.Move(col, CPU);
96                             double value = Evaluate(tempBoard, CPU, col, remainingDepth);
97                             if(value > bestValue) {
98                                 bestValue = value;
99                             }
100                         }
101                     }
102
103                     taskResults[i] = foundMove ? bestValue : 0.0;
104                     if(debug){
105                         cout << "Proces " << rank << ": Task " << i
106                             << " result=" << taskResults[i] << endl;
107                     }
108                 } else {
109                     taskResults[i] = 0.0;
110                     if(debug){
111                         cout << "Proces " << rank << ": Task " << i
112                             << " depth exhausted, result=0.0" << endl;
113                     }
114                 }
115             }
116         }
117
118         // Saberi rezultate
119         vector<double> allResults(numTasks, -2.0);
120         MPI_Allreduce(taskResults.data(), allResults.data(), numTasks,
121                     MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
122     }
123 }
```

Koristi se MPI_Allreduce operacija. Prvo se prikupljaju informacije o broju rezultata od svakog procesa, a zatim se svi rezultati prikupljaju u jedan vektor. Svaki rezultat se kodira kao tri integera (stupac, vrijednost kao double pretvorena u int, i legalni flag).

Za kraj još bih volio samo prikazati kako se „pobjednik” stupaca računao.

```
// Saberi rezultate
vector<double> allResults(numTasks, -2.0);
MPI_Allreduce(taskResults.data(), allResults.data(), numTasks,
              MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

// Pronađi najbolji potez
dBest = -2.0;
iBestCol = -1;

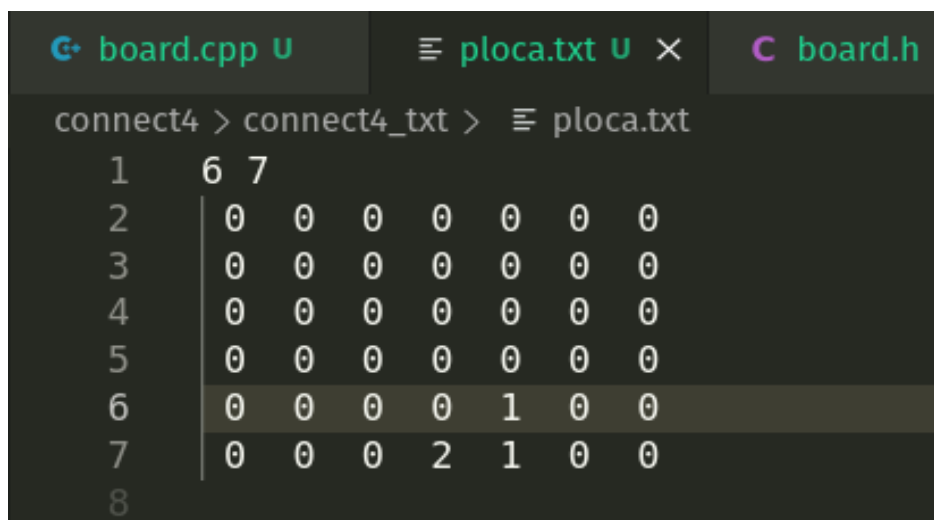
if(rank == 0) {
    vector<double> columnSums(B.Columns(), 0.0);
    vector<int> columnCounts(B.Columns(), 0);

    for(int i = 0; i < numTasks; i++) {
        if(allResults[i] > -2.0) {
            int col = allTasks[i].originalColumn;
            if(col >= 0 && col < B.Columns()) {
                columnSums[col] += allResults[i];
                columnCounts[col]++;
            }
        }
    }

    for(int col = 0; col < B.Columns(); col++) {
        if(B.MoveLegal(col) && columnCounts[col] > 0) {
            double avgValue = columnSums[col] / columnCounts[col];
            if(iBestCol == -1 || avgValue > dBest ||
               (avgValue == dBest && rand() % 2 == 0)) {
                dBest = avgValue;
                iBestCol = col;
            }
            cout << "Stupac " << col << ", vrijednost: " << avgValue
                  << " (iz " << columnCounts[col] << " zadataka)" << endl;
        }
    }
}
}
```

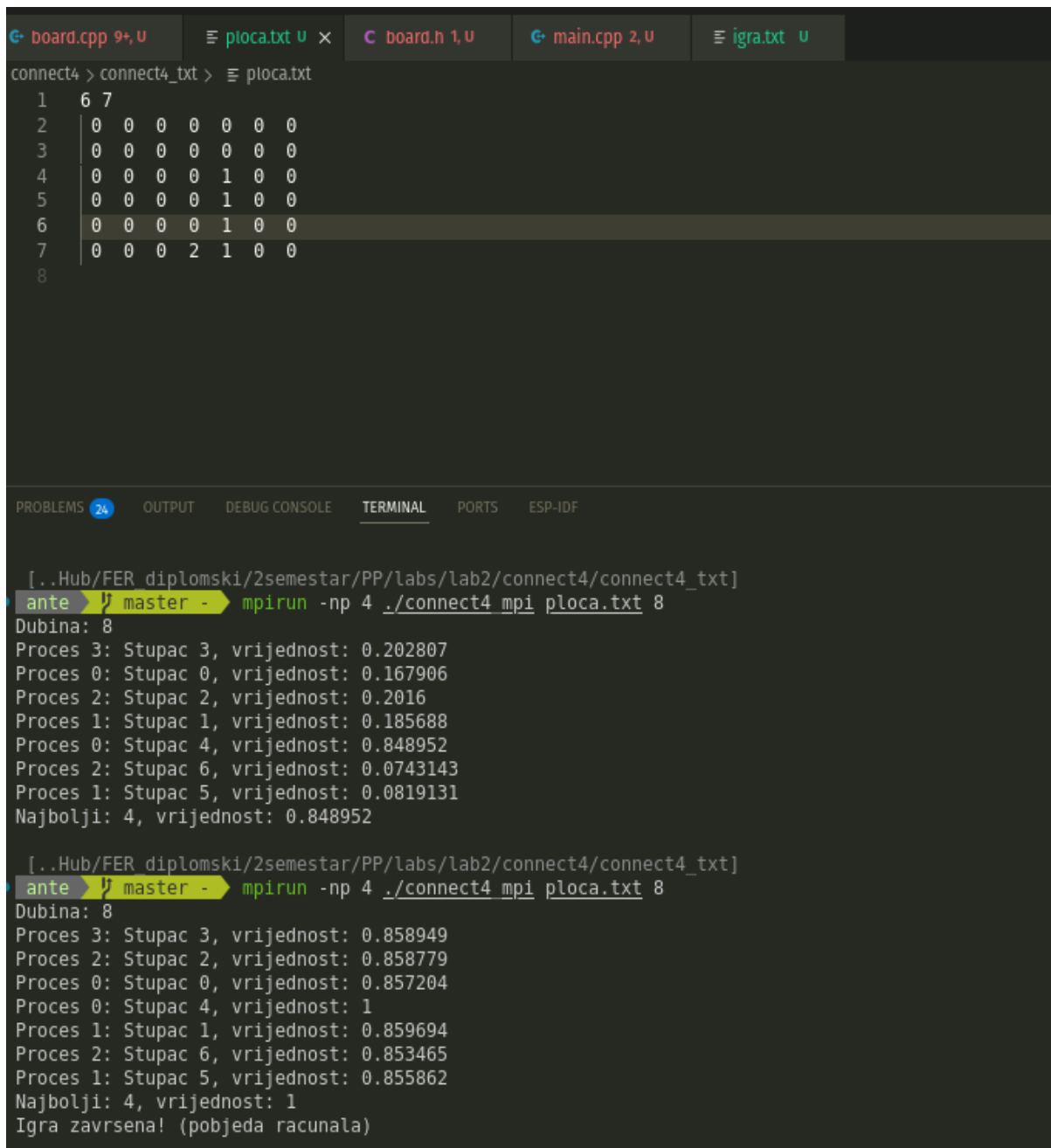
Još je bitno napomenuti da za 1 proces imamo conditional na početku koji radi obični sekvencijalni run (kod koji smo dobili na početku) te da ukoliko imamo više procesa od stupaca, procesima koji ne bi dobili nijedan zadatak dajemo dummy zadatke kako bi tok programa ostao isti.

Za primjer pobjede u 2 koraka sam modificirao ploca.txt :



```
connect4 > connect4_txt > ploca.txt
1      6 7
2      0 0 0 0 0 0 0
3      0 0 0 0 0 0 0
4      0 0 0 0 0 0 0
5      0 0 0 0 0 0 0
6      0 0 0 0 1 0 0
7      0 0 0 2 1 0 0
8
```

Kao što vidimo potrebno je računalu 2 koraka do pobjede. Sada ćemo iskoristiti program da prikazemo ta zadnja 2 koraka:



The screenshot shows a code editor with several files open: `board.cpp`, `ploca.txt`, `board.h`, `main.cpp`, and `igra.txt`. The `ploca.txt` file displays a Connect 4 board state:

```
1 6 7
2 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0
4 0 0 0 0 1 0 0
5 0 0 0 0 1 0 0
6 0 0 0 0 1 0 0
7 0 0 0 2 1 0 0
8
```

The terminal window shows the execution of the MPI program. The first run shows the game state and the best move (column 4) with a value of 0.848952. The second run shows the game state and the best move (column 4) with a value of 1, indicating a win for the computer.

```
[..Hub/FER diplomski/2semestar/PP/labs/lab2/connect4/connect4_txt]
ante master - mpirun -np 4 ./connect4 mpi ploca.txt 8
Dubina: 8
Proces 3: Stupac 3, vrijednost: 0.202807
Proces 0: Stupac 0, vrijednost: 0.167906
Proces 2: Stupac 2, vrijednost: 0.2016
Proces 1: Stupac 1, vrijednost: 0.185688
Proces 0: Stupac 4, vrijednost: 0.848952
Proces 2: Stupac 6, vrijednost: 0.0743143
Proces 1: Stupac 5, vrijednost: 0.0819131
Najbolji: 4, vrijednost: 0.848952

[..Hub/FER diplomski/2semestar/PP/labs/lab2/connect4/connect4_txt]
ante master - mpirun -np 4 ./connect4 mpi ploca.txt 8
Dubina: 8
Proces 3: Stupac 3, vrijednost: 0.858949
Proces 2: Stupac 2, vrijednost: 0.858779
Proces 0: Stupac 0, vrijednost: 0.857204
Proces 0: Stupac 4, vrijednost: 1
Proces 1: Stupac 1, vrijednost: 0.859694
Proces 2: Stupac 6, vrijednost: 0.853465
Proces 1: Stupac 5, vrijednost: 0.855862
Najbolji: 4, vrijednost: 1
Igra završena! (pobjeda racunala)
```

Na kraju smo uspješno pobijedili koristeći 4 procesora i dubinu 8.

Smatram da je bitno nadodati kako generiram zadatke kao što je vidljivo iz priložene slike.


```
struct SimpleTask {
    vector<int> moves; // sekvenca poteza
    int depth;
    double value;
    int originalColumn; // koji je početni stupac
};

// Funkcija za generiranje sekvenci poteza (bez čuvanja Board objekata)
void GenerateTaskSequences(Board& board, int currentDepth, int maxDepth,
    vector<int>& currentMoves, vector<SimpleTask>& tasks,
    CellData currentPlayer, int originalCol = -1) {

    if(currentDepth >= maxDepth) {
        SimpleTask task;
        task.moves = currentMoves;
        task.depth = currentDepth;
        task.value = 0.0;
        task.originalColumn = (originalCol == -1) ?
            (currentMoves.empty() ? -1 : currentMoves[0]) : originalCol;
        tasks.push_back(task);
        return;
    }

    CellData nextPlayer = (currentPlayer == CPU) ? HUMAN : CPU;

    for(int col = 0; col < board.Columns(); col++) {
        if(board.MoveLegal(col)) {
            // Dodaj potez
            currentMoves.push_back(col);
            board.Move(col, nextPlayer);

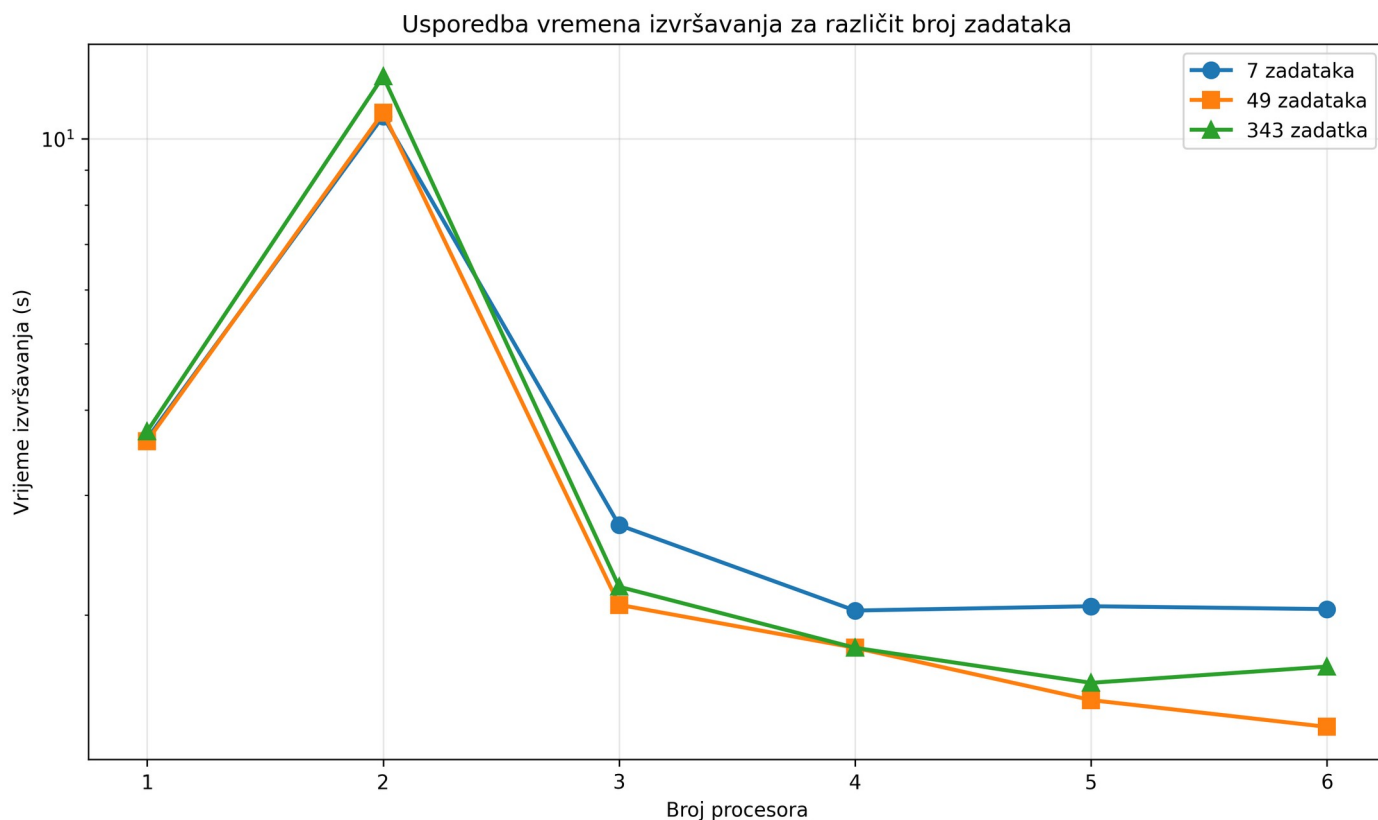
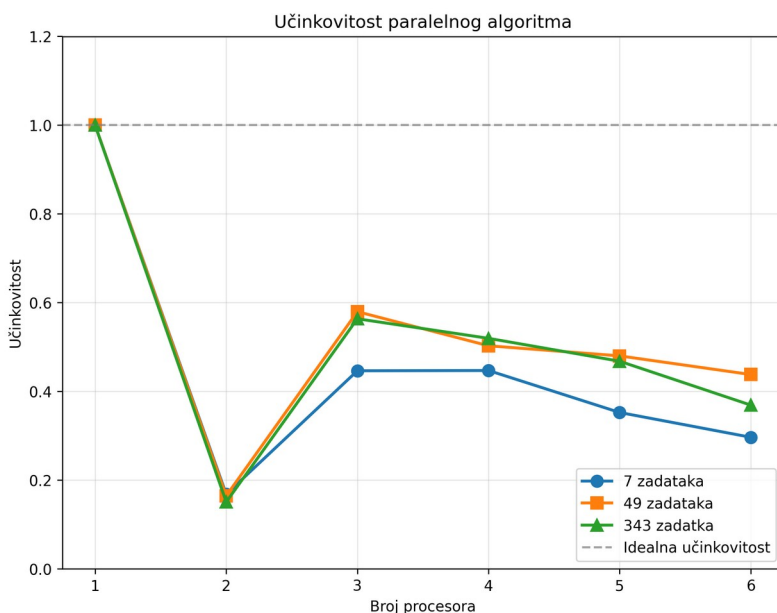
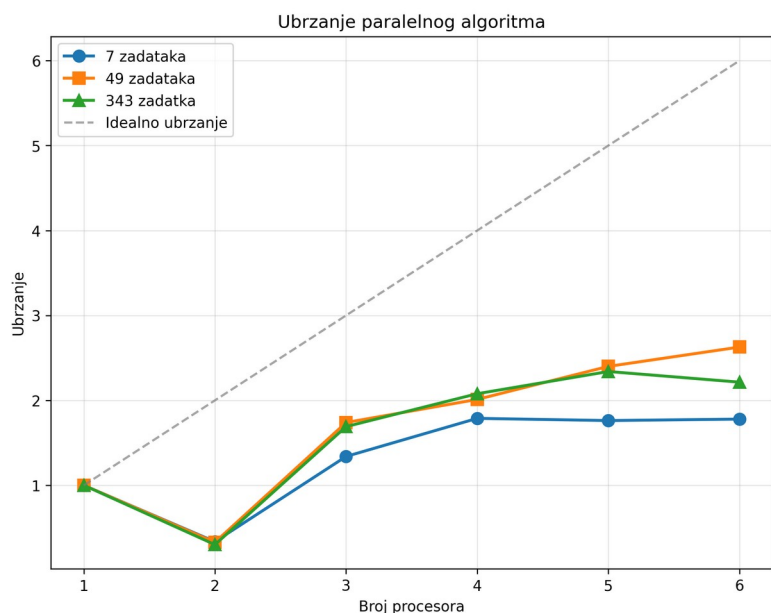
            // Provjeri je li igra završena
            if(board.GameEnd(col)) {
                SimpleTask task;
                task.moves = currentMoves;
                task.depth = currentDepth + 1;
                task.originalColumn = (originalCol == -1) ? currentMoves[0] : originalCol;
                if(nextPlayer == CPU) {
                    task.value = 1.0; // pobjeda
                } else {
                    task.value = -1.0; // poraz
                }
                tasks.push_back(task);
            } else {
                // Rekurzivno generiraj dalje
                int origCol = (originalCol == -1) ? col : originalCol;
                GenerateTaskSequences(board, currentDepth + 1, maxDepth,
                    currentMoves, tasks, nextPlayer, origCol);
            }

            // Ukloni potez
            board.UndoMove(col);
            currentMoves.pop_back();
        }
    }
}
```

2. Kvantitativna analiza

Upute: U ovom dijelu potrebno je priložiti **tablice s rezultatima mjerenja te grafove ubrzanja i učinkovitosti** za tri različita scenarija: kada paralelni algoritam ima 7, 49 i 343 zadatka (uz aglomeraciju na dubini 1, 2 i 3). Mjerenja

treba provesti tako da je najmanje mjereno trajanje (za 8 procesora) reda veličine barem **nekoliko sekundi** (definirajte potrebnu dubinu pretraživanja). Uz grafove, dodajte kratki komentar koji opisuje kako broj zadataka utječe na ubrzanje i učinkovitost (uzevši u obzir utjecaj zrnatosti zadataka, komunikacijskog overhead-a, te udjela programa koji se ne može paralelizirati).



Prikazi grafova vidljivi gore jasno pokazuju promjene i odnos granularnosti i komunikacijskog overhead-a. Iz mojih podataka je vidljivo da se prilikom pokretanja sa 2 procesa javlja veliki zastoje – to je anomalija koju sam nisam dokučio zašto se javlja no vidimo da se za ostatak program ponaša „normalno“.

Maksimalno ubrzanje:

7 zadataka: 1.79x (s 4 procesora)

49 zadataka: 2.63x (s 6 procesora)

343 zadatka: 2.34x (s 5 procesora)

Najbolja učinkovitost (osim 1 procesora):

7 zadataka: 0.45 (s 4 procesora)

49 zadataka: 0.58 (s 3 procesora)

343 zadatka: 0.56 (s 3 procesora)

Broj procesora	7 zadataka - Vrijeme (s)	7 zadataka - Ubrzanje	7 zadataka - Učinkovitost	49 zadataka - Vrijeme (s)	49 zadataka - Ubrzanje	49 zadataka - Učinkovitost	343 zadataka - Vrijeme (s)	343 zadataka - Ubrzanje	343 zadataka - Učinkovitost
1	3.63	1	1	3.6	1	1	3.72	1	1
2	10.79	0.34	0.17	10.93	0.33	0.16	12.35	0.3	0.15
3	2.71	1.34	0.45	2.07	1.74	0.58	2.2	1.69	0.56
4	2.03	1.79	0.45	1.79	2.01	0.5	1.79	2.08	0.52
5	2.06	1.76	0.35	1.5	2.4	0.48	1.59	2.34	0.47
6	2.04	1.78	0.3	1.37	2.63	0.44	1.68	2.21	0.37

- **7 zadataka (gruba zrnatost)**: Najgore skaliranje zbog nedovoljne paralelizacije

Maksimalno ubrzanje $\sim 1.8x$, što ukazuje na ograničenu mogućnost paralelizacije

- **49 zadataka (srednja zrnatost)**: Najbolje skaliranje i učinkovitost

Maksimalno ubrzanje $\sim 2.6x$ s dobrom učinkovitošću na višim brojevima procesora

- **343 zadataka (fina zrnatost)**: Umjereno skaliranje zbog komunikacijskog overhead-a

Maksimalno ubrzanje $\sim 2.3x$, ali s padom učinkovitosti na 6 procesora