Druga domaća zadaća

Igra Connect4

1. Implementacija

Upute: U ovom odjeljku potrebno je opisati ključne dijelove funkcionalnosti koristeći isječke programa i snimke zaslona. Obavezno uključite sljedeće elemente s odgovarajućim komentarima:

- *Isječak programa koji prikazuje pripremu poslova na glavnom (master) procesu.*
- *Isječke programa koji pokazuju kako se zadaci prenose s glavnog (master) procesa na radničke (worker) procese.*
- *Snimku zaslona koja prikazuje posljednja dva koraka igre u kojoj računalo pobjeđuje.*

Ključni dijelovi:

```
class Node:
    Iva Goluža *
    def __init__(self, B: Board):
        self.id = id(B)
        self.B = B
        self.children = []
        self.value = None
```

Node je korišten za izgradnju stabla mogućih poteza min/max igrača. Svaki Node ima **id** pomoću kojeg nakon evaluacije svih zadataka čvorovima dodjeljujem vrijednosti, rezultate evaluacije. **B** je ploča s trenutnim stanjem igre, **children** je lista čvorova djece ovog čvora. **Value** je rezultat evaluacije poteza i stanja igre koji predstavlja taj čvor.

```
class Task:
    Iva Goluža
    def __init__(self, node, depth):
        self.node = node
        self.depth = depth
```

Task ima Node, čvor za koji mora obaviti evaluaciju o depth, dubinu provjere poteza do koje mora ići.

```
class Board:
    Iva Goluža *
    def __init__(self, rows=6, cols=7):
        self.rows = rows
        self.cols = cols
        self.LastMover = EMPTY
        self.LastCol = -1
        self.field = np.full( shape: (rows, cols), EMPTY)
        self.height = np.zeros(cols, dtype=int)
```

Board ima broj redaka i stupaca ploče, zadnjeg igrača (CPU ili HUMAN), zadnje odigrani stupac, polje sa trenutnim stanjem ploče, visine popunjenosti stupaca.

```
DEPTH = 8
ROWS = 6
COLS = 7
LEVEL = 3  # 1: 7 tasks, 2: 49 tasks, 3: 343 tasks
```

Konstante korištene u programu su DEPTH (dubina pretraživanja stabla), ROWS i COLS (dimenzije ploče igre), LEVEL (razina aglomeracije).

Isječak programa koji prikazuje pripremu poslova na glavnom (master) procesu

```
def generate_tasks(node: Node, level, iDepth, tasks, nodes, agglomeration_level):
    # add a node in nodes map (map value setter)
    nodes[node.id] = node
    # set node value if game end
    if node.B.GameEnd(node.B.LastCol):
        node.value = 1 if node.B.LastMover == CPU else -1
        return
    # if max level & not game end -> generate task & stop generating tree
    if level == agglomeration_level:
        task = Task(node, iDepth-level)
        tasks.put(task)
        return

    for possible_move in range(0, node.B.cols):
        if not node.B.MoveLegal(possible_move):
            continue
        B_copy = copy.deepcopy(node.B)
        B_copy.Move(possible_move, HUMAN if node.B.LastMover == CPU else CPU)
        child_node = Node(B_copy)
        node.children.append(child_node)
        generate_tasks(child_node, level+1, iDepth, tasks, nodes, agglomeration_level)
```

Poziv iz master procesa: `generate_tasks(root, 0, iDepth, tasks, nodes, LEVEL)`

Ova rekurzivna funkcija inicijalno prima root čvor na kojega će nadograditi ostatak stabla, a istovremeno iste čvorove sprema u mapu nodes s ključem node.id. Tu mapu kasnije koristim za pristup čvoru kojem trebam pridijeliti rezultat evaluacije. Funkcija prima i level stabla koji trenutno generira, iDepth kao dubinu pretrage stabla, tasks je red zadataka u koji funkcija doda sve stvorene zadatke. Nodes je mapa čvorova i agglomeration_level je razina 1, 2 ili 3, po čemu se odredi na kojoj razini stabla se generiraju zadaci.

Ako je trenutni čvor stanje u kojem igra završava, postavlja se vrijednost čvora i dalje se ne generiraju djeca tog čvora.

Ako se trenutno nalazimo u razini stabla koja je jednaka razini levela aglomeracije, stvara se zadatak i sprema u red.

Inače se generira stablo tako da se za svaki legalni potez iz trenutnog stanja ovog čvora načini novo stablo, Node i doda se u listu djece trenutnog čvora. Za svaki novi čvor se ponovno poziva funkcija, ali s većim levelom (dublja razina stabla).

Na kraju master dobije popunjen red zadataka i root čvor na kojeg je sad povezano cijelo stablo čvorova mogućih stanja, ali bez vrijednosti. Vrijednosti se dodaju nakon što svi procesi izračunaju sve evaluacije poteza listova.

Isječci programa koji pokazuju kako se zadaci prenose s glavnog (master) procesa na radničke (worker) procese

Funkcija koja šalje zadatak na zadanu destinaciju s tagom 1.

```
def send_task(task, dest):
    comm.send(task, dest=dest, tag=1)
    # print(f"Master sent task {task.node.B.LastCol} to worker {dest}", flush=True)
```

```
def cpu_make_move(B, iDepth):
    # generate tree nodes & tasks
    tasks = queue.Queue()
    root = Node(B)
    nodes = {}
    generate_tasks(root, level: 0, iDepth, tasks, nodes, LEVEL)
    pending_results = tasks.qsize()  # number of waiting tasks results
    # print(f"0TASKS INIT: {tasks.qsize()}")
    # spread out the tasks
    free_workers = list(range(1, size))
    while not tasks.empty() or pending_results != 0:
        while free_workers and not tasks.empty():
            task = tasks.get()
            send_task(task, free_workers.pop(0))   # send task to a free worker
        # all workers have a task, check for the results
        status = MPI.Status()
        # is there any task result message available
        message_waiting = comm.Iprobe(source=MPI.ANY_SOURCE, tag=2, status=status)
        if message_waiting:
            task_result = comm.recv(source=status.Get_source(), tag=2)
            # print(f"Master accepts result from {status.Get_source()}", flush=True)
            nodes[task_result['id']].value = task_result['value']  # save the result value in its node
            pending_results = pending_results - 1
            free_workers.append(status.Get_source())  # this worker is now available
        else:
            # if there is no waiting message, master is working on a task
            if not tasks.empty():
                task = tasks.get()
                dResult = Evaluate(task.node.B, task.node.B.LastMover, task.node.B.LastCol, task.depth)
                pending_results = pending_results - 1  # one more task is done
                nodes[task.node.id].value = dResult  # save the result task number in its node using a
    # all tasks results are present, calculate the root result
    evaluate_node(root)
    best_root_child: Node = max(root.children, key=lambda child: child.value)  # this is the best CPU m
    B.Move(best_root_child.B.LastCol, best_root_child.B.LastMover)   # CPU makes its best move possible
    print(f"The best CPU move: {best_root_child.B.LastCol}, value: {best_root_child.value}", flush=True
    return B
```

Tu je cijela cpu_make_move funkcija, žuto su označeni bitni dijelovi gdje master zadatak šalje slobodnom procesu radniku. Raspodjela poslova funkcionira na način da će master slati poslove slobodnim radnicima dok ih ima. Nakon toga provjerava je li dobio odgovor od nekog radnika, ako jest, taj rezultat postavlja u vrijednost pripadnog čvora, a source od kojeg je dobio

odgovor vraća u listu slobodnih radnika. Ako ipak nema odgovora od radnika, master sam uzima zadatak i obavlja ga, a zatim ponovno izvršava ovu petlju sve dok ima neurađenih zadataka ili dok nije primio rezultate svih zadataka.

```
# WORKER process
else:
    while True:
        status = MPI.Status()
        task = comm.recv(source=0, tag=MPI.ANY_TAG, status=status)
        tag = status.Get_tag()

        if tag == 1:
            # print(f"Worker {rank} received task {task.node.B.LastCol}.", flush=True)
            # do the task
            dResult = Evaluate(task.node.B, task.node.B.LastMover, task.node.B.LastCol, task.depth)
            # send result to the master (but include the task.node.id also)
            data = {'id': task.node.id,
                    'value': dResult}
            comm.send(data, dest=0, tag=2)
            # print(f"worker {rank} sent task result {dResult} for col {task.node.B.LastCol}.", flush=True)
        elif tag == 3:
            # there is no more tasks, master is terminating the worker process
            # print(f"Worker {rank} received termination signal.", flush=True)
            break
```

Isječak koda gdje worker proces prima zadatak od mastera ako je tag bio 1, a zatim radi evaluaciju čvora tog zadatka i masteru šalje rezultat i id čvora čiji je to rezultat.

Snimka zaslona koja prikazuje posljednja dva koraka igre u kojoj računalo pobjeđuje

```
The best CPU move: 6, value: 0.23711056502135613
CPU move calculation time: 9.29 seconds
Current board state:
*-------------*
| | | | | | | |
| | | | | | | |
|O| | | | |X| |
|X| | |X| |O|O|
|X| |X|O|O|O|X|
|X| |X|O|O|O|X|
*-------------*
 0 1 2 3 4 5 6
Play your move! [Column 0-6]:2
The best CPU move: 6, value: 1
CPU move calculation time: 5.90 seconds
Game end. CPU is the best! hehe
Current board state:
*-------------*
| | | | | | | |
| | | | | | | |
|O| | | | |X|O|
|X| |X|X| |O|O|
|X| |X|O|O|O|X|
|X| |X|O|O|O|X|
*-------------*
 0 1 2 3 4 5 6
```

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).*

Level: 1 (7 zadataka), Dubina pretraživanja:

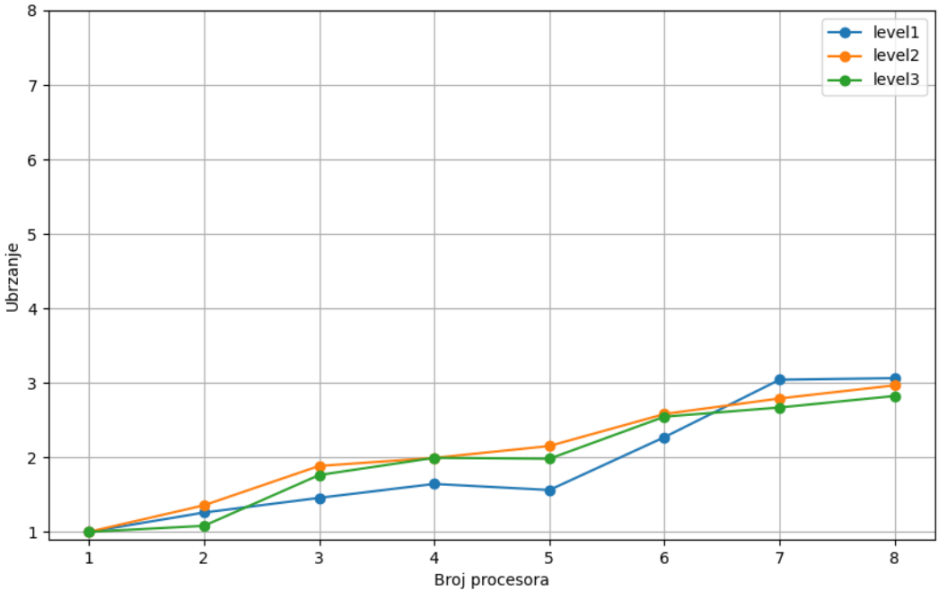
P: broj procesora	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
T: trajanje (s)	*44.12*	*35.03*	*30.29*	*26.84*	*28.25*	*19.42*	*14.50*	*14.40*
Ubrzanje: T1/Tp	*1*	*1.2595*	*1.4566*	*1.6438*	*1.5618*	*2.2719*	*3.0426*	*3.0638*
Učinkovitost: T1/P*Tp	*1*	*0.6298*	*0.4855*	*0.4120*	*0.3123*	*0.3786*	*0.4347*	*0.3829*

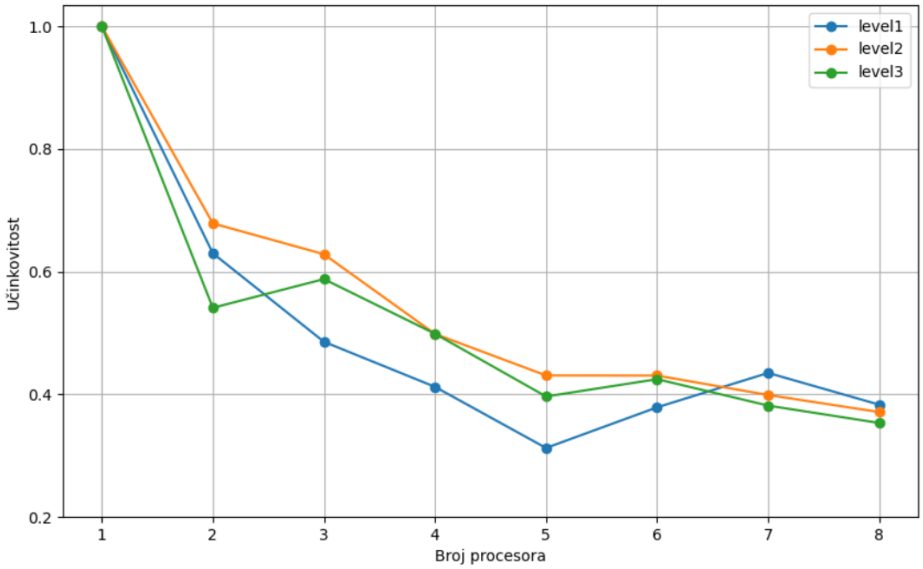
Level: 2 (49 zadataka), Dubina pretraživanja:

P: broj procesora	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
T: trajanje (s)	*45.58*	*33.59*	*24.18*	*22.87*	*21.18*	*17.65*	*16.33*	*15.36*
Ubrzanje: T1/Tp	*1*	*1.3569*	*1.8850*	*1.9930*	*2.1529*	*2.5824*	*2.7912*	*2.9674*
Učinkovitost: T1/P*Tp	*1*	*0.6785*	*0.6283*	*0.4983*	*0.4306*	*0.4304*	*0.3987*	*0.3709*

Level: 3 (343 zadatka), Dubina pretraživanja:

P: broj procesora	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
T: trajanje (s)	*45.79*	*42.33*	*25.97*	*22.96*	*23.11*	*17.98*	*17.15*	*16.21*
Ubrzanje: T1/Tp	*1*	*1.0817*	*1.7631*	*1.9943*	*1.9814*	*2.5467*	*2.6699*	*2.8248*
Učinkovitost: T1/P*Tp	*1*	*0.5409*	*0.5877*	*0.4986*	*0.3963*	*0.4245*	*0.3814*	*0.3531*





Iz grafova se odmah može uočiti da je radom 8 procesora najbolje funkcionirao level1, što je primjer krupnozrnatosti gdje imamo samo 7 većih zadataka, ali se najmanje komunicira između mastera i radnika. To je iz razloga što tada svaki zadatak obavlja poseban procesor, a master obavlja samo poslove mastera (ne obavlja zadatke), i komunikacijski overhead je minimalan. Mjerenja su napravljena na samom početku igre kada bi svaki zadatak trebao biti jednake složenosti, tako da se neće dogoditi da neki procesori obavljaju teže zadatke, a ostali završe ranije i trebaju ih čekati, tj. svi bi završili u približnom vremenskom rasponu. Suprotnost ovog slučaja je level3 sa 343 manja zadatka, gdje će procesori manje računati jednostavnije zadatke, ali će komunikacijski overhead biti veći zbog puno veće količine komunikacije između mastera i radnika. U ovakvom rješenju zadatka, komunikacijski overhead je značajan jer master radniku šalje cijeli Task objekt koji u sebi ima Node s Board i još nekim podacima, što je prikazano na samom početku. Level2 je kao balans ove dvije krajnosti, tako da imamo 49 manjih zadataka od zadataka u level1, ali i manje komunikacijskog overhead-a nego u slučaju korištenja level3. Zbog navedenog je i na grafovima vidljivo variranje ubrzanja i učinkovitosti levela 2 i levela 3 ovisno o broju procesora, gdje se level2 većinski drži iznad levela3, što je zapravo i očekivano zbog manjeg komunikacijskog overhead-a. Za analizu grafa ubrzanja korisno se spomenuti Amdahlovog zakona po kojem ubrzanje paralelnog programa ovisi o udjelu programa koji se ne može paralelizirati, po čemu je ubrzanje veće kada imamo više manjih zadataka koji se mogu obavljati paralelno, dok je kod manje većih zadataka veći dio programa neparaleliziran. To potvrđuje graf na kojem je level1 većinom ispod ostalih, upravo zbog krupnozrnatosti. Pokazalo se da je potrebno balansirati količinu komunikacije i veličinu zadataka, što je upravo opcija korištenja levela 2 (49 zadataka).