UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Rihar

Povzporejanje algoritma CMA-ES

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2017

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nejc Rihar

Povzporejanje algoritma CMA-ES

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: prof. dr. Branko Šter

SOMENTOR: izr. prof. dr. Peter Korošec

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Raziščite različne možnosti pohitritve evolucijskega algoritma CMA-ES na osnovi vzporednega procesiranja. Opišite različne pristope, jih implementirajte in eksperimentalno ovrednotite. Na koncu podajte tudi ugotovitve, do katerih ste prišli na osnovi dobljenih rezultatov.

Zahvaljujem se prof. dr. Branku Šteru, za pomoč pri urejanju diplomskega dela. Prav tako se zahvaljujem izr. prof. dr. Petru Korošču za pomoč pri razumevanju obravnavane teme in strukturiranju diplomskega dela. Hvala tudi staršem za vso podporo tekom študija.

Kazalo

Povzetek

Abstract

Poglavje 1 Uvod 1

Poglavje 2 Evolucijski algoritmi 3

2.1 Algoritem CMA-ES 4

2.1.1 Inicializacija 5

2.1.2 Generiranje populacije 5

2.1.3 Ovrednotenje populacije 6

2.1.4 Prilagoditev strateških parametrov 7

Poglavje 3 Vzporedni pristopi 9

3.1 Večnitnost 9

3.2 Programiranje z vmesnikom MPI 11

3.3 Programiranje na GPE 12

Poglavje 4 Implementacija 15

4.1 Vzporedna implementacija z večnitnostjo 16

4.1.1 Generiranje populacije 17

4.1.2 Ovrednotenje populacije 17

4.1.3 Izračun kovariančne matrike 18

4.2 Vzporedna implementacija z uporabo vmesnika MPI 18

4.2.1 Generiranje populacije 20

4.2.2 Ovrednotenje populacije 21

4.2.3 Izračun kovariančne matrike 21

4.3 Vzporedna implementacija z uporabo grafične kartice 22

4.3.1 Generiranje populacije 25

4.3.2 Ovrednotenje populacije 27

4.3.3 Izračun kovariančne matrike 27

Poglavje 5 Rezultati meritev 29
5.1 Meritve časov izvajanja povzporejanih korakov algoritma 29
5.1.1 Meritve vzporednega pristopa z večnitnostjo 30
5.1.2 Meritve vzporednega pristopa MPI 35
5.1.3 Meritve vzporednega pristopa MPI z večnitnostjo 38
5.1.4 Meritve vzporednega pristopa OpenCL 41
5.1.5 Primerjava učinkovitosti vzporednih pristopov 45
5.2 Primerjava zaporedne in vzporedne implementacije algoritma 47
5.3 Splošne ugotovitve 50
Poglavje 6 Sklepne ugotovitve 51
Literatura 53

Seznam uporabljenih kratic

kratica	angleško	Slovensko
CMA-ES	covariance matrix adaptation evolution strategy	evolucijska strategija s prilagajanjem kovariančne matrike
OpenCL	open computing language	odprti programski jezik
CPE	central processing unit	centralna procesna enota
GPE	graphics processing unit	grafična procesna enota
MPI	message passing interface	vmesnik za izmenjavo sporočil
CUDA	compute unified device architecture	platforma za vzporedno procesiranje
GPGPU	general-purpose computing on graphics processing units	splošnonamensko računanje na grafičnih procesnih enotah
JOCL	java bindings for OpenCL	javanski vmesnik OpenCL
HTT	Hyper-Threading technology	Intelova večnitna tehnologija

Povzetek

Dandanes se na področju računalništva z razvojem vse boljše strojne opreme kaže vse večja potreba po čimhitrejšem izvajanju algoritmov. Temelj diplomskega dela je poskus pohitritve evolucijskega algoritma CMA-ES z uporabo različnih vzporednih pristopov. V prvem delu diplomskega dela smo predstavili temelje delovanja evolucijskih algoritmov ter podrobneje opisali posamezne korake algoritma CMA-ES. V nadaljevanju smo predstavili izbrane vzporedne pristope in opisali postopek povzporejanja algoritma pri njihovi uporabi. Pri tem smo izpostavili glavne posebnosti posameznih vzporednih pristopov in težave, na katere smo pri njihovi uporabi naleteli. Poleg tega smo opisali še razne optimizacije, s katerimi smo izboljšali njihovo učinkovitost. V zadnjem delu diplomskega dela smo predstavili in analizirali konkretne rezultate meritev v različnih kontekstih. Primerjali smo čase izvajanja posameznih korakov algoritma pri različnih nastavitvah vzporednih pristopov in pri različnih vhodnih parametrih algoritma. Na koncu smo primerjali še čase izvajanja zaporedne in vzporedne implementacije algoritma in iz pridobljenih rezultatov podali nekaj splošnih ugotovitev.

Ključne besede: Evolucijski algoritmi, CMA-ES, centralna procesna enota, grafična procesna enota, OpenCL, MPI, večnitnost, vzporedni pristop.

Abstract

Nowadays, with the development of increasingly better computer hardware we can witness an ever-growing need for faster execution of algorithms. The basis of this thesis is an attempt to speed-up the evolutionary algorithm CMA-ES using various parallel approaches. Firstly we present the foundations of evolutionary algorithms and describe in detail each step of the algorithm CMA-ES. In the following chapters we present each parallel approach and describe the procedure of parallelizing the algorithm with their use. We outline the main particularities of each parallel approach and present various problems which we encountered in their application. In addition, we describe a variety of optimizations which can be used to improve the effectiveness of each parallel approach. In the last part of the thesis we present and analyze the results of our measurements in different contexts. We compare the execution times of individual parallelized algorithm steps while running the algorithm with different input parameters. Afterwards we compare the execution times of the serial and parallel implementations of the algorithm and highlight some general findings.

Key words: Evolutionary algoritms, CMA-ES, central processing unit, graphics processing unit, OpenCL, MPI, multithreading, parallel approach.

Poglavje 1 Uvod

Računalništvo in svet biologije se laiku morda na prvi pogled zdita kot povsem različna svetova, vendar se ti dve področji dandanes zelo pogosto prepletata. Računalničarji že vrsto let razvijajo algoritme za obdelavo in analizo različnih bioloških podatkov, biologi pa iz dneva v dan prihajajo do novih odkritij, ki računalničarjem služijo kot navdih pri snovanju novih algoritmov.

Biološke procese si lahko predstavljamo kot nekakšne algoritme, zasnovane s strani narave za reševanje različnih problemov. Pogosto uporabljen, ampak še vedno dober primer takega procesa je proces biološke evolucije, ki ga lahko opišemo kot spreminjanje dednih lastnosti organizmov iz generacije v generacijo. Pri tem ključno vlogo igra t.i. selektiven izbor najboljših posameznikov iz generacije, katerih genetski material se preko razmnoževanja prenese na naslednjo (boljšo) generacijo. Pomembno vlogo tu igrajo še razne mutacije, ki v posamezno populacijo vnesejo določeno mero naključnosti. Mehanizmi procesa evolucije so računalničarjem služili kot osnova za razvoj t.i. evolucijskih algoritmov, ki jih uporabljamo za iskanje čim boljših rešitev optimizacijskih problemov.

Cilj diplomskega dela je pohitritev izvajanja evolucijskega algoritma CMA-ES z uporabo različnih vzporednih pristopov. Algoritem CMA-ES je pretežno podatkovno vzporeden, kar v osnovi samo pomeni, da v korakih algoritma pogosto opravljamo razne računske operacije (kot sta npr. množenje in seštevanje) nad medsebojno neodvisnimi podatki, kar je idealen pogoj za njihovo vzporedno izvajanje.

Pri povzporejanju algoritma smo uporabljali tri različne vzporedne pristope. V sklopu prvega vzporednega pristopa smo izkoriščali centralno procesno enoto (CPE) oz. bolj natančno njene niti. V sklopu drugega vzporednega pristopa smo izkoriščali omrežje medsebojno povezanih računalnikov, kar smo storili z uporabo vmesnika za izmenjavo sporočil MPI. V okviru tretjega vzporednega pristopa pa smo izkoriščali računsko moč grafične procesne enote (GPE), kar smo storili z uporabo vmesnika OpenCL.

Diplomsko delo je tako razdeljeno na štiri glavna poglavja. V prvem poglavju smo najprej predstavili evolucijske algoritme in nato še podrobneje opisali delovanje algoritma CMA-ES. V drugem poglavju smo opisali delovanje in posebnosti izbranih vzporednih pristopov. V tretjem poglavju smo pričeli z opisom postopka implementacije zaporednega algoritma in nato nadaljevali z opisom poteka povzporejanja algoritma. V zadnjem poglavju smo predstavili in analizirali konkretne rezultate meritev v različnih kontekstih. Pričeli smo s primerjavo časov izvajanja povzporejanih korakov algoritma pri uporabi različnih vzporednih pristopov, nato pa smo nadaljevali z medsebojno primerjavo učinkovitosti vzporednih pristopov in s primerjavo dejanskega časa izvajanja vzporedne in zaporedne implementacije algoritma. Na koncu smo navedli še nekaj splošnih ugotovitev, do katerih nas je privedla izdelava diplomskega dela.

Poglavje 2 Evolucijski algoritmi

Evolucijski algoritmi so, kot nam pove že njihovo ime, algoritmi osnovani na mehanizmih biološke evolucije. Uvrščamo jih med metahevristične optimizacijske algoritme [1].

S pojmom hevristične tehnike oz. bolj pogosto jo imenujemo kar hevristika, označujemo vsak tak pristop k reševanju nekega problema, ki nam ne vrne optimalne oz. natančne rešitve, ampak tako, ki je dovolj dobra oz. zadovoljiva za naše potrebe. Hevristiko ponavadi uporabljamo pri takih problemih, kjer je iskanje optimalne rešitve nepraktično ali celo nemogoče, zaradi česar se zadovoljimo s približno rešitvijo problema.

S pojmom metahevristike označujemo postopek iskanja oz. izbire take hevristike, ki nam pri reševanju nekega problema lahko vrne zadovoljivo rešitev [2].

Optimizacijski problem pa lahko opišemo kot problem izbora najboljše rešitve (glede na neke podane kriterije) iz neke določene množice ugodnih rešitev [3]. Torej kot primer lahko nek optimizacijski problem (v našem primeru problem minimizacije) predstavimo na naslednji način:

- Podano imamo neko funkcijo $f: A \rightarrow R$ nad neko množico A, ki vsebuje realna števila.
- Iščemo pa tak element x_0 iz množice A, za katerega velja, da je $f(x_0) \leq f(x)$ za vsako vrednost x iz množice A.

Evolucijski algoritmi so torej iterativni algoritmi, ki jih uporabljamo za iskanje čim boljših rešitev optimizacijskih problemov. Pred pričetkom izvajanja algoritmu najprej podamo nek nabor prekinitvenih pogojev, ki predstavljajo oceno oz. mero kakovosti rešitve, ki jo želimo z algoritmom doseči in mora biti ob koncu njegovega izvajanja zadoščena.

Izvajanje evolucijskih algoritmov pričnemo z generiranjem nabora posameznikov, ki predstavljajo populacijo. V naslednjem koraku nadaljujemo z ovrednotenjem posameznikov iz novonastale populacije, kar storimo z uporabo t.i. testne funkcije [4]. Nato nadaljujemo z naslednjo fazo algoritma, katero sestavlja več ponavljajočih se korakov, ki jih izvajamo dokler ne izpolnimo izbranih prekinitvenih pogojev.

Ti koraki so sledeči:

- Izbor najbolj uspešnih posameznikov iz trenutne populacije za razmoževanje (starši).
- Ustvarjanje potomcev iz staršev preko procesa rekombinacije in procesa mutacije.
- Ovrednotenje uspešnosti novonastalih posameznikov.
- Zamenjava trenutne populacije z novonastalo populacijo.

V zgornjih korakih smo omenili procesa rekombinacije in mutacije. V okviru procesa rekombinacije na podlagi dveh ali več staršev ustvarimo enega ali več potomcev, kar storimo s kombiniranjem lastnosti staršev. Z mutacijami pa poskrbimo za spreminjanje naključnih lastnosti naključno izbranih posameznikov, s čimer v novonastalo populacijo vnesemo neko določeno mero naključnosti. Uporabo evolucijskih algoritmov lahko zasledimo na veliko različnih področjih, kot so strojništvo, ekonomija, robotika, sociologija, umetnost ipd.

2.1 Algoritem CMA-ES

Sedaj, ko poznamo osnove delovanja evolucijskih algoritmov, lahko nadaljujemo še s podrobnejšim opisom delovanja algoritma CMA-ES. Slednjega uvrščamo v podskupino evolucijskih algoritmov, ki jih imenujemo evolucijske strategije [5] (angl. *Evolution Strategies*). Evolucijske strategije so stohastične metode brez derivatov, ki se uporabljajo za numerično optimizacijo nelinearnih ali nekonveksnih optimizacijskih problemov. Stohastika je pojem, s katerim označujemo uporabo naključnih spremenljivk pri izvajanju algoritmov [6].

Delovanje algoritma CMA-ES je v osnovi podobno delovanju tipičnih evolucijskih algoritmov opisanih zgoraj, s to razliko, da si pri iskanju čim boljših rešitev problemov pomagamo še z uporabo t.i. kovariančne matrike. To uporabljamo za prilagajanje določenih strateških parametrov algoritma, ki vplivajo na njegovo nadaljnje delovanje. Njena uporaba pripomore k hitrejšemu oz. bolj učinkovitemu iskanju rešitve problema. Posamezne korake algoritma smo opisali v sledečih podpoglavjih.

2.1.1 Inicializacija

Izvajanje algoritma pričnemo z inicializacijo strateških parametrov, ki vplivajo na njegovo osnovno delovanje. Mednje spadajo dimenzija problema (v nadaljevanju označena z črko N), velikost populacije (v nadaljevanju označena z črko λ), velikost koraka itd. Hkrati poskrbimo še za inicializacijo vseh struktur in spremenljivk, ki jih tekom izvajanja algoritma uporabljamo za hranjenje raznih podatkov. Poleg tega izberemo še testno funkcijo, ki jo bomo uporabljali pri ovrednotenju populacije in nastavimo prekinitvene pogoje algoritma.

2.1.2 Generiranje populacije

Ko končamo z inicializacijo algoritma, lahko nadaljujemo z izvajanjem ostalih korakov algoritma v iteracijah. V prvem koraku vsake iteracije najprej poskrbimo za generiranje populacije, kar v algoritmu CMA-ES storimo tako kot prikazuje izsek kode 1.

```
1. for (int k = 0; k < lambda; ++iNk){
2.     for (int i = 0; i < N; ++i){
3.         artmp[i] = diagD[i] * rand.nextGaussian();
4.     }
5.     for (i = 0; i < N; ++i) {
6.         for (int j = 0, double sum = 0; j < N; ++j) {
7.             sum += b[i][j] * artmp[j];
8.         }
9.     population[k][i] = xMean[i] + sigma * sum;
10.    }
11. }
```

Izsek kode 1: Generiranje populacije

Korak generiranja populacije sestavlja več gnezdenih zank *for*. V vsaki iteraciji zunanje zanke ustvarimo enega posameznika iz populacije. Vsakega posameznika predstavlja seznam realnih števil dolžine *N*, ki ga hranimo v dvodimenzionalnem seznamu populacije. Posameznike generiramo glede na trenutne vrednosti strateških parametrov algoritma. Naš algoritem je tudi stohastičen, zaradi česar pri generiranju posameznih vrednosti posameznikov uporabljamo še nabor naključnih števil, katerih vrednosti so porazdeljene po standardni normalni porazdelitvi (3. vrstica v izseku kode 1). Standardna normalna porazdelitev (2.1) je porazdelitev vrednosti s povprečjem oz. aritmetično sredino 0 in standardnim odklonom 1.

$$\theta(x) = \frac{e^{-\frac{1}{2}x^2}}{\sqrt{2\pi}} \tag{2.1}$$

2.1.3 Ovrednotenje populacije

Ko končamo z generiranjem populacije, nadaljujemo z ovrednotenjem uspešnosti posameznikov iz novonastale populacije. Kako to storimo, je odvisno od izbrane testne funkcije, v našem primeru smo pretežno uporabljali t.i. Rosenbrockovo testno funkcijo (2.2) [7].

$$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2] \tag{2.2}$$

Njeno implementacijo v našem algoritmu smo prikazali v izseku kode 2.

```
1 int res = 0;
2 for (int i = 1; i < x.length - 1; i++){
3     res += 100 * (x[i] * x[i] - x[i + 1]) * (x[i] * x[i] - x[i + 1])
4         + (x[i] - 1.) * (x[i] - 1.);
5 }
```

Izsek kode 2: Rosenbrockova testna funkcija.

Kot je razvidno iz izseka kode 2, uspešnost nekega posameznika *x* izračunamo preko raznih računskih operacij, ki jih opravimo nad vrednostmi, ki posameznika sestavljajo (2. in 3. vrstica v izseku kode 2). Ko končamo z ovrednotenjem vseh posameznikov iz populacije, nadaljujemo z izborom najbolj uspešnega izmed njih. Njegovo uspešnost nato primerjamo s trenutno najboljšo doseženo uspešnostjo algoritma. Boljša oz. nižja izmed obeh postane nova najboljša dosežena uspešnost.

2.1.4 Prilagoditev strateških parametrov

V zadnjem koraku vsake iteracije algoritma poskrbimo še za posodobitev strateških parametrov algoritma. Korak smo glede na posodabljane strateške parametre razdelili na več podkorakov. V prvem podkoraku najprej posodobimo oz. izračunamo t.i. evolucijski poti algoritma, ti vsebujeta razne podatke o korelaciji med zaporednimi iteracijami algoritma, ki jih uporabimo pri posodabljanju ostalih strateških parametrov algoritma.

Podatke v prvi evolucijski poti uporabimo pri naslednjem podkoraku, v okviru katerega izvedemo posodobitev oz. izračun kovariančne matrike, kar storimo tako, kot prikazuje izsek kode 3.

```
1  for (int i = 0; i < N; ++i) {
2       for (int j = 0; j <= i; ++j) {
3           C[i][j] = (1 - ccov) * C[i][j] + ccov * (1. / mucov)
4               * (pc[i] * pc[j]+ (1 - hsig) * cc * (2. - cc) * C[i][j]);
5
6           for (int k = 0; k < mu; ++k) {
7               C[i][j] += ccov * (1 - 1. / mucov) * weights[k]
8                   * (population[fitCol.fitness[k].i][i] - xOld[i])
9                   * (population[fitCol.fitness[k].i][j] - xOld[j])
10                  / sigma / sigma;
11          }
12      }
13 }
```

Izsek kode 3: Izračun vrednosti kovariančne matrike.

Kot lahko vidimo, podkorak sestavljajo tri gnezdene zanke *for*. V prvem delu podkoraka najprej izračunamo vrednosti v kovariančni matriki *C* na podlagi podatkov v evolucijski poti *pc* in glede na vrednosti strateških konstant algoritma *ccov, mucov, cc in hsig* (3. in 4. vrstica v izseku kode 3). Vrednosti slednjih so odvisne od velikosti populacije in dimenzije problema. V najbolj notranji zanki pa vrednosti v kovariančni matriki še dodatno posodobimo glede na lastnosti trenutne populacije in glede na vrednosti strateških konstant *weights, ccov, mucov* ter vrednost strateškega parametra *sigma* (7. do 10. vrstica v izseku kode 3).

Podatke iz druge evolucijske poti pa skupaj s posodobljeno kovariančno matriko uporabimo v naslednjem, zadnjem podkoraku v okviru katerega poskrbimo še za posodobitev strateškega parametra *sigma,* ki vpliva predvsem na način generiranja populacije in posodabljanje kovariančne matrike.

Ob koncu vsake iteracije algoritma še preverimo, če smo izpolnili katerega izmed izbranih prekinitvenih pogojev. Če nismo izpolnili nobenega, pričnemo z novo iteracijo algoritma, v nasprotnem primeru pa končamo z izvajanjem algoritma, saj smo našli zadovoljivo rešitev problema.

Poglavje 3 Vzporedni pristopi

V tem poglavju smo se predvsem posvetili pregledu osnovnega delovanja posameznih vzporednih pristopov. To nam bo predvsem služilo kot podlaga za lažje razumevanje postopka povzporejanja algoritma v naslednjem poglavju. Kot smo že omenili, smo pri povzporejanju algoritma uporabljali tri vzporedne pristope, za katere smo menili, da bodo najbolj učinkoviti.

3.1 Večnitnost

Marsikateremu uporabniku računalnika se dandanes zdi samoumevno, da lahko na računalniku hkrati izvaja več različnih programov, brez da bi zaradi tega srečeval razne probleme, kot so prekinitve, zmrzovanje ipd. Zahvala za to gre predvsem večjedrnim CPE, ki so dandanes že nadomestile tradicionalne enojedrne CPE [8]. Te sestavlja več samostojnih procesnih enot oz. jeder, ki berejo in izvajajo programske ukaze. Vsako jedro lahko izvaja eno ali več t.i. niti. Njihova uporaba nam omogoča vzporedno oz. hkratno izvajanje večih procesov na računalniku. Vsako jedro ima svoje lastne predpomnilnike, vsa jedra pa medsebojno komunicirajo preko skupnega predpomnilnika in glavnega pomnilnika RAM (angl. *Random Access Memory)* kot prikazuje tudi slika 3.1.

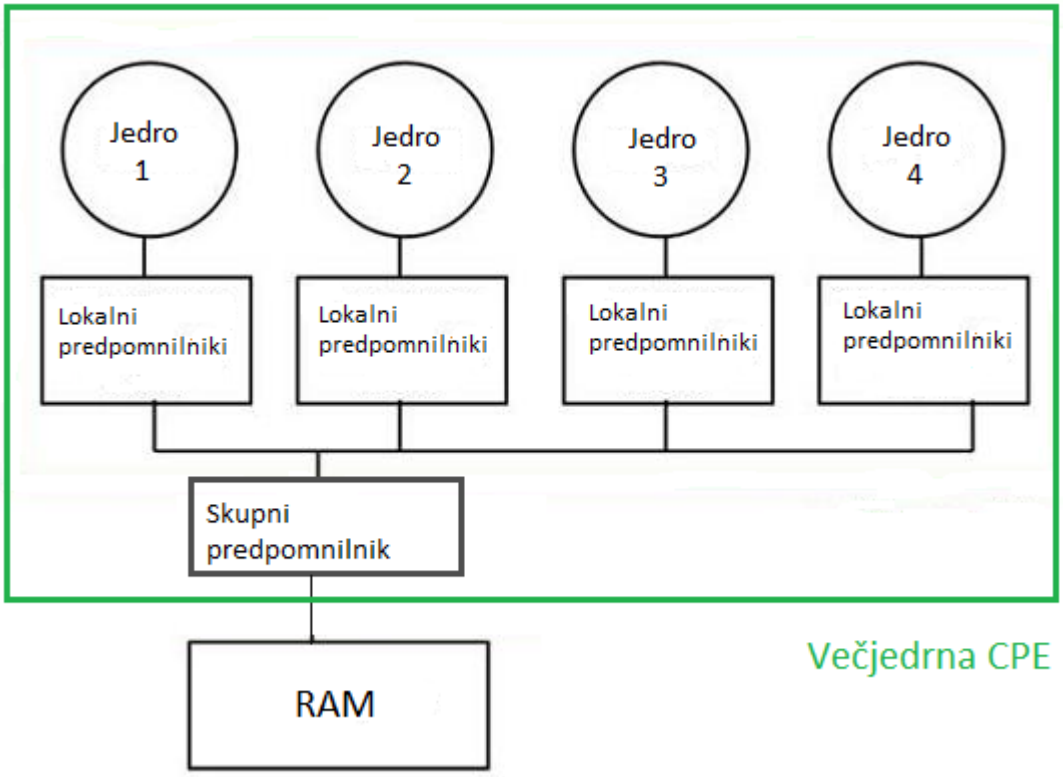
Zaradi arhitekture večjedrnih CPE se pri njihovi uporabi srečujemo še z dodatnim problemom učinkovitega upravljanja s procesi oz. opravili. Procese na računalniku želimo tekom njihovega izvajanja čim bolj učinkovito razdeliti po prej omenjenih nitih na tak način, da bodo slednje karseda dobro izkoriščene, saj bo tako celoten sistem deloval veliko hitreje, kot bi pri uporabi enojedrne CPE.

Algoritme pri uporabi pristopa večnitnosti ponavadi povzporejamo tako, da posamezne korake algoritma razdelimo na manjše kose oz. podkorake in te predamo v izvajanje nitim CPE. V teoriji je najvišji možni faktor pohitritve pri vzporednem izvajanju nekega koraka enak številu uporabljenih niti, maksimalna pohitritev celotnega algoritma pa je odvisna še od deleža algoritma, ki ga izvajamo vzporedno, kar narekuje tudi Amdahlov zakon (3.1).

$$S \leq \frac{1}{(1-\mathrm{p})} \tag{3.1}$$

V zgornji enačbi velja da:

- S predstavlja maksimalno teoretično pohitritev algoritma.
- p predstavlja delež programa, ki ga izvajamo vzporedno.

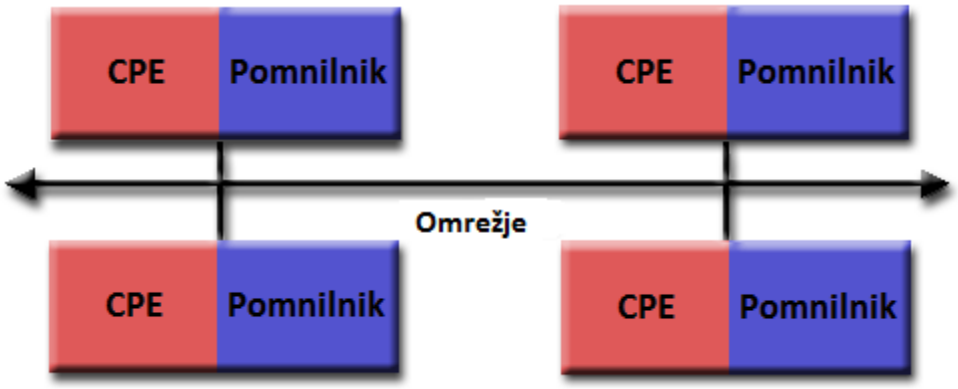


Slika 3.1: Poenostavljen primer arhitekture štirijedrne CPE.

3.2 Programiranje z vmesnikom MPI

Naš drugi vzporedni pristop za pohitritev izvajanja algoritma izkorišča vmesnik za izmenjavo sporočil med procesi (angl. *Message Passing Interface*, v nadaljevanju MPI) [9]. Ta je sicer prvotno namenjen izvajanju v programskih jezikih C, C++ in Fortran, vendar njegovo uporabo dandanes omogoča že veliko ostalih programskih jezikov. Programe MPI ponavadi izvajamo v omrežju medsebojno povezanih procesov (računalnikov), ki ga prikazuje slika 3.2. Omrežje MPI lahko simuliramo tudi z uporabo CPE, s čimer se izognemo njegovi zapleteni konfiguraciji. Vmesnik MPI je zasnovan tako, da ga lahko uporabljamo v različnih omrežjih računalnikov, neodvisno od arhitekture posameznih računalnikov v omrežju. Njegovo uporabo najpogosteje zasledimo na področju reševanja računsko zelo zahtevnih problemov z uporabo gruče (angl. *cluster*) računalnikov.

Pri uporabi MPI se na vsakem procesu v omrežju izvaja enaka instanca programa. Ker ima vsak izmed procesov v omrežju lasten in od drugih procesov ločen pomnilnik, moramo tekom izvajanja MPI programa še dodatno skrbeti za sinhronizacijo podatkov med procesi. Za ta namen imamo na voljo več vgrajenih funkcij MPI, ki jih lahko uporabljamo tako za izmenjavo in sinhronizacijo podatkov med procesi kot tudi za pridobivanje raznih informacij o omrežju, v katerem program izvajamo.



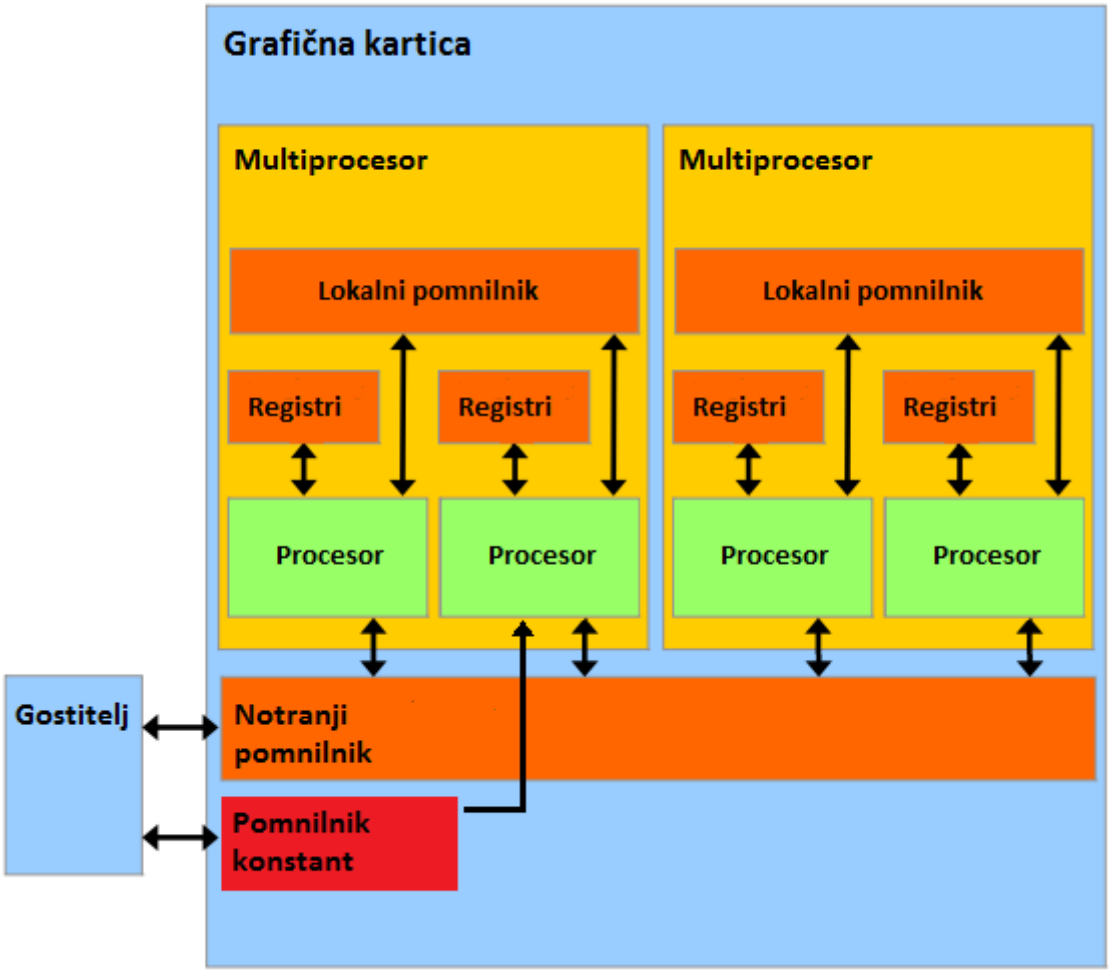
Slika 3.2: Shema omrežja MPI - več medsebojno povezanih procesov oz. računalnikov.

3.3 Programiranje na GPE

V zadnjem desetletju smo lahko opazovali razcvet igračarske industrije in z njo povezan razvoj vse bolj zmogljivih grafičnih procesnih enot (v nadaljevanju grafičnih kartic). Slednje so računsko zmogljivejše kot CPE, vendar je bila njihova računska moč še nedavno rezervirana le za uporabo na področju računalniške grafike. Dandanes pa postaja njihova uporaba vse bolj priljubljena na področjih, kjer se je dosedaj tradicionalno uporabljala CPE. Uporabo grafičnih kartic pri reševanju bolj splošnih računskih problemov imenujemo splošnonamensko računanje na grafičnih procesnih enotah (angl. *General-purpose computing on graphics processing units*), kar s kratico označimo kot GPGPU.

GPGPU je v glavnem omogočil razvoj dveh platform. V letu 2007 je podjetje NVIDIA predstavilo svojo platformo CUDA, ki je bila prva, ki je omogočala uporabo grafičnih kartic za reševanje bolj splošnonamenskih problemov. Imela je le eno glavno omejitev, njena uporaba je bila namreč podprta le s strani grafičnih kartic NVIDIA, kar pa konkurenčnim podjetjem ni bilo všeč.

Zaradi tega se je kmalu zatem pričelo snovanje alternative, ki jo danes poznamo pod imenom OpenCL [10]. Standard oz. ogrodje OpenCL je leta 2009 predstavila skupina podjetij za odprte standarde, imenovana *The Khronos Group*. Gre za bolj širokonamensko ogrodje, ki je poleg programiranju grafičnih kartic namenjeno tudi programiranju mnogo drugih naprav. Veliko podjetij kot so npr. Apple, AMD in Intel nudi svojo lastno platformo OpenCL, ki je optimizirana za boljše delovanje na njihovih napravah. Za učinkovito uporabo oz. programiranje grafičnih kartic moramo dobro poznati njihovo arhitekturo, saj se ta zelo razlikuje od arhitekture CPE. Arhitekturo tipične grafične kartice smo prikazali na sliki 3.3.



Slika 3.3: Poenostavljena arhitektura tipične grafične kartice.

Kot je razvidno iz zgornje slike, grafično kartico sestavljajo notranji pomnilnik, pomnilnik konstant in več multiprocesorjev, ki jih v terminologiji OpenCL imenujemo delovne skupine. Vsaka delovna skupina ima tudi lasten lokalni pomnilnik in nabor procesorjev, ki jih v terminologiji OpenCL imenujemo delovne enote. Vsaka delovna enota ima še lasten nabor registrov, v katere shranjuje svoje lokalne podatke.

Za učinkovito programiranje na grafični kartici moramo tudi dobro poznati lastnosti posameznih pomnilnikov, ki jo sestavljajo. Te smo podrobneje opisali spodaj:

- Notranji pomnilnik je hkrati največji in najpočasnejši pomnilnik na grafični kartici. Obenem služi tudi kot posrednik za komunikacijo in prenos podatkov med gostiteljem in grafično kartico. Zaradi njegove relativne počasnosti se njegovi uporabi pri programiranju poskušamo čim bolj izogibati, če je to seveda možno. Preko njega lahko komunicirajo vse delovne enote na grafični kartici.

- V pomnilniku konstant ponavadi hranimo razne konstante, kot nam namiguje že njegovo ime. Čas dostopa do pomnilnika konstant je nekoliko manjši kot čas dostopa do notranjega pomnilnika. V naprotju z notranjim pomnilnikom pa je pomnilnik konstant po velikosti veliko manjši, na tipični grafični kartici lahko vanj shranimo nekje od 16 do 64 KB podatkov.

- Vsak multiprocesor oz. delovna skupina na grafični kartici ima svoj lasten lokalni pomnilnik. V nasprotju z notranjim pomnilnikom je ta veliko hitrejši, ampak po velikosti tudi veliko manjši. Zaradi tega je najbolj primeren za hranjenje takih spremenljivk, do katerih tekom izvajanja programa dostopamo večkrat. Do lokalnega pomnilnika lahko dostopajo le delovne enote znotraj pripadajoče delovne skupine.

Velja še omeniti dejstvo, da medtem, ko dandanes tipično CPE sestavlja nekje od 2 do 16 jeder, lahko na grafičnih karticah najdemo tudi več tisoč procesorjev. Ti so sicer sami po sebi precej manj zmogljivi kot jedra CPE, vendar če jih pri reševanju nekega problema dobro izkoristimo, njihova številčnost to praviloma odtehta.

Poglavje 4 Implementacija

V tem poglavju smo opisali celoten postopek implementacije in povzporejanja algoritma po posameznih vzporednih pristopih ter orodja, ki so bila pri tem uporabljena. Prva naloga, ki smo jo imeli, je bila implementacija zaporednega algoritma CMA-ES v programskem jeziku Java. Že obstoječo implementacijo zaporednega algoritma smo imeli na voljo na spletu [5], zaradi česar pri tem nismo naleteli na posebne težave. V nadaljevanju smo se lotili pretvorbe algoritma v modularno obliko, kar pomeni, da smo morali algoritem razdeliti na manjše logične dele oz. module. Pri tem smo si pomagali z ogrodjem *Algorithms*, razvitem na Institutu "Jozef Stefan", ki nam je zelo olajšalo delo.

Ogrodje *Algorithms* je osnovano na ogrodju *Java Spring Framework*. Slednji deluje na osnovi konfiguracijske datoteke XML, znotraj katere definiramo posamezne korake oz. module algoritma (ki jih imenujemo *beans*) in zaporedje njihovega izvajanja. V primeru našega algoritma imamo torej eno glavno zanko, znotraj katere izvajamo posamezne korake algoritma. Ob koncu vsake iteracije zanke še preverimo, če smo izpolnili katerega izmed izbranih prekinitvenih pogojev. V primeru, da smo izpolnili enega ali več izmed njih, prekinemo z izvajanjem algoritma, v nasprotnem primeru pa pričnemo z izvajanjem naslednje iteracije.

Glavna motivacija za razdelitev algoritma na module je bila želja po tem, da bi lahko posamezne module prenašali med sorodnimi algoritmi in s tem ali ustvarili nove algoritme ali izboljšali že obstoječe. Z uporabo modularnega pristopa tudi izboljšamo preglednost programske kode, saj imamo celoten algoritem razdeljen na logične dele.

Ko smo končali s pretvorbo algoritma v modularno obliko, smo morali še preveriti pravilnost njegovega delovanja. To smo storili tako, da smo delovanje modularnega in nemodularnega algoritma primerjali pri različnih vhodnih parametrih algoritma. Ko smo se prepričali, da modularna implementacija algoritma deluje v skladu z pričakovanji, smo nadaljevali s povzporejanjem algoritma.

V prvi fazi smo najprej opravili analizo časovne in računske zahtevnosti vseh korakov algoritma. Za povzporejanje smo sprva izbrali štiri časovno in računsko najbolj zahtevne korake. Za enega izmed korakov algoritma, ki v grobem predstavlja približno 10 % do 20 % časa izvajanja algoritma, smo ugotovili, da spada med t.i. izključno serijske probleme, kar pomeni, da ga ne moremo izvajati vzporedno, saj vsak zaporedni del koraka potrebuje rezultat prejšnjega dela za pravilno delovanje. Na koncu smo torej povzporejali tri korake algoritma, postopek česar smo opisali v sledečih podpoglavjih.

4.1 Vzporedna implementacija z večnitnostjo

Pri povzporejanju korakov algoritma smo pričeli s pristopom večnitnosti, predvsem zaradi njegove enostavnosti. V programskem okolju Java se programi privzeto izvajajo na eni niti CPE. Vzporedno izvajanje programov nam omogočajo razredi znotraj paketa *Concurrent*. V našem primeru smo uporabljali razred *ThreadPool*, kateremu kot argument podamo število niti, ki jih želimo pri izvajanju programa uporabljati. Maksimalno število niti, ki jih lahko uporabljamo, je seveda enako številu niti na CPE.

Korak, ki ga želimo prirediti za vzporedno izvajanje na nitih CPE, moramo najprej razdeliti na manjše kose oz. podkorake in jih oviti v razred *Runnable*. Te nato predamo v izvajanje nitim CPE, kar storimo preko razreda *ThreadPool*. Pri tem je dobra praksa, da vsem uporabljenim nitim dodelimo približno enako količino dela, saj s tem poskrbimo za njihovo enakomerno obremenitev, kar pripomore k bolj učinkovitemu vzporednemu izvajanju koraka. Pri izvajanju korakov na nitih CPE imamo na voljo še možnost sinhronega ali asinhronega izvajanja. O sinhronem izvajanju algoritmov govorimo takrat, ko lahko z izvajanjem naslednjega koraka algoritma pričnemo šele, ko končamo z izvajanjem prejšnjega koraka. O asinhronem izvajanju pa govorimo takrat, ko lahko več korakov algoritma izvajamo istočasno oz. hkrati.

V našem primeru mora biti izvajanje algoritma izključno sinhrono, torej moramo pred pričetkom izvajanja naslednjega koraka algoritma vedno počakati, da vse niti končajo z izvajanjem svojega deleža koraka.

4.1.1 Generiranje populacije

Pri povzporejanju smo pričeli s korakom generiranja populacije, ki ga sestavlja več gnezdenih zank *for*. Podatki med posameznimi iteracijami zunanje zanke so medsebojno neodvisni, zaradi česar smo se povzporejanja koraka lotili tako, da smo zunanjo zanko *for* razdelili na več manjših kosov oz. iteracij, ki smo jih nato predali v izvajanje nitim CPE. Pri vzporednemu izvajanju koraka torej vsaka nit poskrbi za generiranje določenega dela populacije.

V prvi vzporedni implementaciji tega koraka smo v nasprotju s pričakovanji opazili poslabšanje v času izvajanja koraka. Razlog je bil v tem, da lahko do razreda *Random*, ki smo ga uporabljali za generiranje naključnih števil, dostopa le ena nit naenkrat, zaradi česar so niti med izvajanjem koraka veliko časa čakale ena na drugo. Rešitev težave smo našli v posebni implementaciji razreda *Random*, prirejeni za izvajanje v večnitnem okolju, imenovani *ThreadLocalRandom*. Ta deluje tako, da vsaki niti dodeli lasten generator naključnih števil, ki je po delovanju enak generatorju naključnih števil znotraj razreda *Random*. Z njegovo uporabo smo odpravili opisano težavo in uspešno pohitrili izvajanje koraka.

4.1.2 Ovrednotenje populacije

Povzporejanje koraka za ovrednotenje populacije je bilo zelo enostavno, saj ga sestavlja le preprosta zanka *for*, ki iterira skozi posameznike iz populacije in te ovrednoti glede na izbrano testno funkcijo. Podatki med posameznimi iteracijami zanke *for* so medsebojno neodvisni, zaradi česar smo ta korak povzporedili na enak način kot prejšnjega, torej smo zanko *for* razdelili na manjše iteracije in te predali v izvajanje nitim CPE. Pri vzporednemu izvajanju koraka torej vsaka nit poskrbi za ovrednotenje določenega dela populacije.

4.1.3 Izračun kovariančne matrike

Na koncu nam je ostal še korak izračuna kovariančne matrike, katerega sestavljajo tri gnezdene zanke *for*. Ta korak je časovno in računsko najbolj zahteven, zaradi česar se tudi najbolje povzporeja. Podatki med posameznimi iteracijami zunanje zanke so tudi v tem koraku medsebojno neodvisni, zaradi česar smo korak zopet povzporedili na enak način kot prejšnja koraka opisana zgoraj. Pri vzporednemu izvajanju tega koraka torej vsaki niti dodelimo del skupnih iteracij zunanje zanke *for*, tako da vsaka nit poskrbi za izračun določenega dela kovariančne matrike.

4.2 Vzporedna implementacija z uporabo vmesnika MPI

V nadaljevanju smo se lotili povzporejanja algoritma z uporabo vmesnika MPI. Za ta namen smo bili najprej primorani poiskati javanski vmesnik MPI. Teh je na voljo veliko, v našem primeru smo se odločili za uporabo vmesnika MPJ Express [11], predvsem zaradi tega, ker je sintaksno enak prvotnemu vmesniku MPI.

Kot smo omenili že v tretjem poglavju, programe MPI praviloma izvajamo v omrežju medsebojno povezanih računalnikov. V našem primeru smo se odločili za alternativo, torej za simulacijo omrežja MPI z uporabo CPE, predvsem zaradi tega, ker se v tem primeru izognemo zapleteni konfiguraciji omrežja MPI. Tu še velja omeniti, da so vzporedne implementacije korakov algoritma kljub temu programsko povsem enake, kot bi bile pri uporabi dejanskega omrežja medsebojno povezanih računalnikov.

Delovanje vmesnika MPI temelji na izmenjevanju sporočil med procesi. Za ta namen imamo na voljo dva tipa medsebojne komunikacije. Prvi tip komunikacije nam omogoča komunikacijo iz točke v točko, drugi tip komunikacije pa nam omogoča kolektivno komunikacijo med vsemi procesi v omrežju MPI.

Pri programiranju imamo za potrebe obeh tipov komunikacije na voljo več vgrajenih funkcij MPI.

- Za komunikacijo iz točke v točko primarno uporabljamo funkciji MPI_SEND in MPI_RECEIVE, ki služita pošiljanju in prejemanju raznih podatkov med dvema procesoma v omrežju.

- Za potrebe kolektivne komunikacije med procesi imamo na voljo več funkcij za prenos podatkov med vsemi procesi kot je npr. MPI_GATHER, ki se uporablja za zbiranje oz. združevanje podatkov iz vseh procesov. Na voljo imamo še več funkcij namenjenih sinhronizaciji procesov (funkcija MPI_BARRIER), pošiljanju podatkov iz enega procesa na vse ostale (funkcija MPI_BROADCAST), pridobivanju podatkov o omrežju (funkciji MPI_SIZE in MPI_RANK) in še veliko podobnih funkcij.

Pred pričetkom izvajanja MPI programov vedno definiramo število procesov, ki jih želimo tekom izvajanja programa uporabljati. Izvajanje programa na vseh procesih v omrežju sprožimo s klicem funckije MPI_INIT. Tekom izvajanja programa lahko uporabljamo vgrajene funkcije MPI, primere katerih smo navedli zgoraj. Ko želimo izvajanje programa na vseh procesih prekiniti, to storimo s klicem funkcije MPI_FINALIZE.

Pri povzporejanju algoritma smo že na začetku naleteli na težavo, saj so bile določene spremenljivke, kot sta npr. populacija in kovariančna matrika, shranjene v obliki dvodimenzionalnih seznamov oz. matrik. Težava je tu nastala predvsem zaradi tega, ker smo pri povzporejanju korakov algoritma pogosto uporabljali zgoraj omenjeno funkcijo MPI_GATHER, ki uporabe dvodimenzionalnih seznamov ne podpira. Zaradi tega smo morali vse dvodimenzionalne sezname algoritma pretvoriti v enodimenzionalno obliko. Zaradi lažje predstave in razlage smo način pretvorbe seznamov ponazorili kar s spodnjim primerom.

V prvotni implementaciji algoritma smo podatke o populaciji hranili v dvodimenzionalnem seznamu, ki je imel λ vrstic in N stolpcev. Vsaka vrstica v seznamu populacije je tako predstavljala enega posameznika. Ta seznam smo pretvorili v enodimenzionalni seznam dolžine $\lambda \times N$. V enodimenzionalnem seznamu populacije je tako začetni indeks nekega posameznika i, enak $i \times N$, njegov končni indeks pa $((i+1) \times N) - 1$. Na enak način smo v nadaljevanju pretvorili še ostale dvodimenzionalne sezname.

Preden nadaljujemo še omenimo, da je klic funkcije MPI_GATHER sinhron, kar v praksi samo pomeni, da ob klicu te funkcije vsak proces pred nadaljevanjem z izvajanjem algoritma počaka, da s svojim delom končajo tudi ostali procesi v omrežju. S tem zagotovimo, da imajo vsi procesi pred pričetkom izvajanja naslednjega koraka v pomnilniku vse potrebne podatke.

4.2.1 Generiranje populacije

Pri povzporejanju smo pričeli s korakom generiranja populacije, ki ga sestavlja več gnezdenih zank *for*. Kot smo že omenili, v vsaki iteraciji zunanje zanke *for* generiramo enega posameznika, podatki med posameznimi iteracijami pa so medsebojno neodvisni.

Pri uporabi razreda za generiranje naključnih števil *Random* smo tu zopet naleteli na težavo, predvsem zaradi načina delovanja MPI. Kot smo že omenili, se na vseh procesih v omrežju MPI izvajajo enake instance programa, zaradi česar smo na vseh procesih imeli instance razreda *Random* z identičnimi semeni. Seme je preprosto povedano le neko število, ki vpliva na generiranje naključnih števil. Ker smo v vsakem procesu imeli enako seme, smo zaradi tega tudi v vsakem procesu generirali identična naključna števila, kar za delovanje algoritma ni bilo optimalno. To težavo smo odpravili tako, da smo razred *Random* v vsakem procesu inicializirali s semenom odvisnim od identifikacijske številke pripadajočega procesa. S tem smo poskrbeli, da je razred *Random* v vsakem procesu v omrežju generiral drugačen nabor naključnih števil.

Pri vzporednemu izvajanju koraka vsakemu izmed M procesov v izvajanje podamo $\frac{\lambda}{M}$ iteracij zunanje zanke. Vsak proces torej generira določen del populacije, ki ga shranjuje v svoj lokalni seznam populacije. Ob koncu koraka kličemo funkcijo MPI_ALLGATHER, ki združi vse lokalne sezname populacije iz posameznih procesov nazaj v skupni seznam, ki predstavlja celotno populacijo. Ta funkcija se od prej omenjene funkcije MPI_GATHER razlikuje le po tem, da novonastali združeni seznam ob koncu koraka pošlje vsem procesom v omrežju. Z njeno uporabo poskrbimo, da imajo ob koncu izvajanja koraka vsi procesi v pomnilniku seznam celotne populacije, ki ga potrebujejo za pravilno delovanje nadaljnjih korakov algoritma.

4.2.2 Ovrednotenje populacije

Povzporejanje koraka za ovrednotenje populacije je bilo prav tako relativno trivialno. Korak sestavlja zanka *for*, ki iterira skozi posameznike populacije in te ovrednoti glede na izbrano testno funkcijo. Pri vzporednemu izvajanju koraka vsakemu izmed M procesov v izvajanje podamo $\frac{\lambda}{M}$ iteracij zanke *for*. Vsak proces torej poskrbi za ovrednotenje določenega dela populacije, rezultate izračunov pa shranjuje v svoj lokalni seznam. Ob koncu koraka s klicem funkcije MPI_ALLGATHER zopet poskrbimo za sinhronizacijo podatkov oz. rezultatov med procesi MPI.

4.2.3 Izračun kovariančne matrike

Pri povzporejanju koraka za izračun kovariančne matrike smo naleteli še na dodatno težavo. Korak sestavljajo tri gnezdene zanke *for.* Za posamezne iteracije zunanje zanke *for* velja, da tem višja kot je zaporedna številka iteracije, tem večje je skupno število računskih operacij, ki jih v dani iteraciji opravimo.

V prvi, nekoliko naivni implementaciji koraka, smo vsakemu izmed M procesov v izvajanje preprosto dodelili $\frac{N}{M}$ iteracij zunanje zanke. Vsak proces je tako poskrbel za izračun določenega dela kovariančne matrike, rezultate izračunov pa je hranil v lokalnem seznamu. Ta implementacija je sicer delovala pravilno, vendar je bila zelo neučinkovita, saj je bila porazdelitev dela med procesi neenakomerna. Težavo nam je tu povzročal predvsem klic funckije MPI_ALLGATHER ob koncu koraka. Za lažje razumevanje smo težavo ponazorili kar s spodnjim primerom.

Denimo, da naš algoritem izvajamo na 4 procesih MPI. Procesu 1 dodelimo prvih $\frac{N}{4}$ iteracij zunanje zanke, procesu 4 pa zadnjih $\frac{N}{4}$ iteracij. Kot smo že omenili, s povečevanjem zaporedne številke iteracije povečujemo tudi število računskih operacij, ki jih v dani iteraciji opravimo. Zaradi tega je proces 1 moral opraviti veliko manj dela kot proces 4. Ker pa je funkcija MPI_ALLGATHER sinhrona, je moral proces 1 pred nadaljevanjem z izvajanjem algoritma vseeno počakati, da so ostali procesi končali s svojim delom, medtem pa ni počel ničesar, zaradi česar je bil slabo izkoriščen.

Vzporedno implementacijo koraka smo torej morali prirediti na tak način, da je bila porazdelitev dela med procesi enakomerna. V tej implementaciji najprej izračunamo skupno število računskih operacij φ, ki jih v koraku opravimo. Nato vsakemu procesu v izvajanje dodelimo toliko iteracij zunanje zanke *for,* da je skupno število računskih operacij, ki jih proces opravi, enako $\frac{\varphi}{M}$. Tako smo poskrbeli za enakomerno porazdelitev dela med procesi, zaradi česar smo zmanjšali čas čakanja ob klicu funkcije MPI_ALLGATHER in konkretno izboljšali čas izvajanja koraka.

V nadaljevanju smo naredili še alternativno implementacijo MPI, ki izkorišča še večnitnost. Ta implementacija je v osnovi enaka kot osnovna implementacija MPI, z eno glavno razliko. V vsakem povzporejanem koraku algoritma vsak proces svoje delo še dodatno razdeli na manjše kose in te preda v izvajanje nitim CPE, kar stori na enak način kot pri prej opisanem pristopu z večnitnostjo. S tem pristopom smo čase izvajanja nekaterih korakov algoritma še dodatno izboljšali.

4.3 Vzporedna implementacija z uporabo grafične kartice

Na koncu smo se lotili še povzporejanja korakov algoritma z uporabo grafične kartice, kar smo storili s pomočjo ogrodja OpenCL. Ta je prvotno namenjen izvajanju v programskih jezikih C, C++ in Fortran, zaradi česar smo bili primorani poiskati javanski vmesnik zanj. Teh smo na spletu našli veliko, mi smo se odločili za uporabo t.i. JOCL [12], predvsem zaradi tega, ker je sintaksno enak prvotnemu standardu OpenCL.

Vsak OpenCL program sestavljata vsaj dva dela. Prvi se izvaja na t.i. gostitelju (ponavadi CPE), drugi pa na napravi (v našem primeru grafični kartici). Oba dela imata zelo različne naloge. Gostiteljev del skrbi za vso potrebno inicializacijo OpenCL, za podajanje ščepca (angl. *kernel*) napravi v izvajanje ter za prenos podatkov med seboj in napravo. Naprava pa poskrbi za izvajanje prejetega ščepca. Ščepec je preprosto povedano le programska koda oz. funkcija, ki se izvaja na izbrani napravi. Programski jezik, v katerem pišemo ščepce, se imenuje OpenCL C, ta je zasnovan na podlagi programskega jezika C99. Tu še omenimo da programski jezik OpenCL C ni objektno usmerjen, zaradi česar pri pisanju in izvajanju ščepcev ne moremo uporabljati raznih objektov in razredov.

Omenili smo že, da gostitelj skrbi za vso potrebno inicializacijo OpenCL. To sestavlja več korakov:

- Izbor platforme
- Izbor naprave
- Ustvarjanje programa in konteksta
- Ustvarjanje ukazne vrste

Veliko podjetij ima razvito svojo lastno platformo oz. implementacijo OpenCL, optimizirano za boljše delovanje na njihovih napravah. Med tipične platforme spadajo npr. Intel, NVIDIA, AMD, Apple itd. V sklopu izbora platforme glede na strojno opremo gostitelja izberemo eno izmed njih. Na podlagi izbrane platforme izberemo še napravo, na kateri želimo ščepce izvajati.

Nato se lotimo ustvarjanja konteksta in programa. Kontekst ustvarimo glede na izbrano platformo in napravo. Ta skrbi za upravljanje z ukazno vrsto, programom, ščepcem in pomnilnikom naprave. Po inicializaciji konteksta se lotimo ustvarjanja programa, kateremu kot argumente podamo ščepce, ki jih želimo izvajati na napravi. Ščepce ponavadi hranimo v datotekah s končnico .cl, lahko pa jih shranjujemo tudi kot nize znotraj kode. Nadaljujemo z ustvarjanjem objekta imenovanega *Kernel*, kateremu kot argumente podamo spremenljivke, ki jih želimo prenesti na izbrano napravo. Na koncu ustvarimo še ukazno vrsto, ki skrbi za upravljanje z raznimi operacijami na napravi.

Preden lahko pričnemo z izvajanjem ščepca na napravi, moramo definirati še število delovnih enot, ki jih pri izvajanju ščepca želimo uporabljati. Poleg tega lahko definiramo še t.i. lokalno velikost dela, preko katere določimo število uporabljenih delovnih skupin pri izvajanju ščepca. Če slednje ne definiramo, to stori vmesnik OpenCL sam, vendar to pogosto stori na neoptimalen način, zaradi česar tudi izvajanje ščepca ni optimalno.

Pred nadaljevanjem si oglejmo še strukturo ščepcev. V glavo ščepca podamo vse spremenljivke, ki jih želimo pri njegovem izvajanju uporabljati. Te spremememenljivke so enake tistim, ki smo jih podali kot argumente razredu *Kernel* na gostitelju. Primer glave ščepca smo prikazali v izseku kode 4.

```
1.      __kernel void matrixKernel(__global double *matrix,
2.      __global double *population, __global int *fitness,
3.      __global double *xOld, __global double *pc,
4.      __constant double *weights, double sigma, double ccov,
5.      double mucov, int mu, double cc, int hsig, int N,
6.      __local double *localMatrix)
```

Izsek kode 4: Primer glave ščepca.

Kot lahko vidimo, sezname v ščepcu podajamo s t.i. kazalci (angl. *pointer*) značilnimi za programski jezik C, ter eno izmed treh predpon:

- S predpono __global označujemo podatke shranjene v notranjem pomnilniku.
- S predpono __constant označujemo podatke, shranjene v pomnilniku konstant.
- S predpono __local pa označujemo podatke shranjene v lokalnem pomnilniku. V lokalni pomnilnik lahko podatke zapisujemo le znotraj ščepca.

Pri programiranju ščepcev si pogosto pomagamo še z raznimi vgrajenimi funkcijami OpenCL, saj lahko z njihovo uporabo izvemo razne podatke, kot so indeks trenutne delovne enote ali delovne skupine, skupno število uporabljenih delovnih enot in delovnih skupin ipd. Ob koncu izvajanja ščepcev moramo rezultate izračunov prenesti nazaj iz naprave na gostitelja. Sedaj, ko smo spoznali osnovno strukturo OpenCL programov, lahko nadaljujemo z opisom postopka povzporejanja korakov algoritma.

4.3.1 Generiranje populacije

Pri koraku generiranja populacije smo do sedaj imeli že kar nekaj problemov z generiranjem naključnih števil. Pri vzporednemu pristopu z večnitnostjo in vmesnikom MPI smo za generiranje naključnih števil uporabljali razreda *ThreadLocalRandom* in *Random*, pri programiranju našega ščepca pa teh razredov nismo imeli na voljo, saj programski jezik OpenCL C ni objektno orientiran.

V prvi, nekoliko naivni vzporedni implementaciji koraka smo zaradi tega v vsaki iteraciji algoritma naključna števila najprej generirali na gostitelju in jih nato poslali na napravo. Ta implementacija koraka je sicer delovala pravilno, vendar se je izkazala za zelo neučinkovito, saj je prenos seznama števil iz gostitelja na napravo v vsaki iteraciji algoritma terjal preveč časa.

Zaradi neučinkovitosti prve implementacije smo se odločili za implementacijo lastnega generatorja naključnih števil, kar na grafični kartici ni trivialno opravilo. Zaradi tega smo se lotili iskanja relativno preprostega ampak hkrati učinkovitega generatorja naključnih števil. Na koncu smo se odločili za uporabo t.i. Lehmerjevega generatorja naključnih števil [13]. Njegovo implementacijo v našem ščepcu smo prikazali v izseku kode 5.

```
 1.  int a = 16807;
 2.  int m = 2147483647;
 3.  int q = 127773;
 4.  int r = 2836;
 5.  int lo, hi, test;
 6.  hi = seed / q;
 7.  lo = seed % q;
 8.  test = a * lo - r * hi;
 9.  if (test > 0){
10.      seed = test;
11.  }
12.  else{
13.      seed = test + m;
14.  }
```

Izsek kode 5: Lehmerjev generator naključnih števil.

Generator naključnih števil inicializiramo tako, da v prvi iteraciji algoritma na gostitelju generiramo ustrezno velik seznam naključnih števil (ki služijo kot semena generatorja) in ga pošljemo na napravo.

Generator naključnih števil v ščepcu deluje tako, da iz ustreznega mesta v dobljenem seznamu vzame seme (v izseku kode 5. imenovano *seed*) in ga najprej deli z vrednostjo q (pri tem velja $q = \frac{m}{a}$), nato pa še po modulu z vrednostjo q in rezultata shrani v spremenljivki *hi* in *lo*. Novo naključno število nato izračuna tako, kot je prikazano med 8. in 14. vrstico v izseku kode 5 in to shrani v spremenljivko *seed*. Generirano število nato shranimo nazaj v seznam naključnih števil in ga v naslednji iteraciji algoritma zopet uporabimo kot seme za generiranje novega naključnega števila.

Naš implementirani generator naključnih števil generira naključna naravna števila z vrednostmi med 0 in 2^{31}. Kot smo že omenili, za pravilno delovanje našega koraka potrebujemo naključna števila, katerih vrednosti so porazdeljene po standardni normalni porazdelitvi. Vrednosti generiranih naključnih števil smo torej morali še ustrezno prilagoditi, kar smo v naši implementaciji storili z uporabo t.i. Box-Mullerjeve transformacije [14]. Njeno implementacijo v ščepcu smo prikazali v izseku kode 6.

```
 1.  double number,r,x,y = 0.0;
 2.      do{
 3.          seed = getRandom(seed);
 4.          number = seed * 1.0 / 2147483647;
 5.          x = 2.0 * number - 1.0;
 6.          seed = getRandom(seed);
 7.          number = seed * 1.0 / 2147483647;
 8.          y = 2.0 * number - 1.0;
 9.          r = x*x + y*y;
10.      }while (r > 1 || r == 0);
11.  double gRandom = x * sqrt(-2.0 * log(r) / r)
```

Izsek kode 6: Box-Mullerjeva transformacija naključnih števil.

Funkcija *getRandom(seed)* znotraj zanke *do-while* vrača naključna števila, generirana z uporabo našega implementiranega generatorja naključnih števil. Ustrezno porazdeljeno naključno število *gRandom* pridobimo tako kot prikazuje zgornji izsek kode 6. Z uporabo Box-Mullerjeve transformacije smo odpravili še zadnje težave povezane z generiranjem naključnih števil, zaradi česar smo lahko nadaljevali s povzporejanjem koraka.

Kot smo že omenili, korak generiranja populacije sestavlja več gnezdenih zank *for*. V ščepcu smo povzporedili vse razen najbolj notranje zanke. Pri izvajanju ščepca uporabljamo $\lambda \times N$ delovnih enot. Znotraj ščepca vsaka delovna enota najprej poskrbi za generiranje ustrezno porazdeljenega naključnega števila, katerega nato shrani v seznam naključnih števil.

Pred nadaljevanjem izvedemo še globalno sinhronizacijo delovnih enot, s katero poskrbimo, da so vse delovne enote končale z generiranjem svojega naključnega števila. Vsaka delovna enota v nadaljevanju poskrbi za izračun ene vrednosti določenega posameznika, ki jo nato shrani v seznam populacije. Ob koncu izvajanja ščepca novonastali seznam populacije prenesemo nazaj na gostitelja.

Ko smo končali z povzporejanjem koraka, smo še dodatno preverili učinkovitost delovanja implementiranega generatorja naključnih števil. To smo storili tako, da smo primerjali delovanje zaporedne in vzporedne implementacije koraka pri različnih vhodnih parametrih algoritma. Pri uporabi obeh smo dosegli zelo podobne rezultate, s čimer smo se prepričali, da implementirani generator naključnih števil deluje dovolj dobro za potrebe našega algoritma.

4.3.2 Ovrednotenje populacije

Pri povzporejanju koraka za ovrednotenje populacije smo morali povzporediti vsako izmed izbranih testnih funkcij, kar je bilo relativno enostavno, saj smo pri vsaki izmed njih povzporedili le zanko *for*, ki iterira skozi posameznike populacije. V ščepcu torej vsaka delovna enota poskrbi za ovrednotenje enega posameznika iz populacije, zaradi česar pri izvajanju ščepca uporabljamo le λ delovnih enot. Izvajanje koraka z uporabo OpenCL se ni izkazalo za učinkovito, saj je izkoriščenost grafične kartice zaradi majhnega števila uporabljenih delovnih enot preslaba, da bi njeno uporabo lahko opravičili.

4.3.3 Izračun kovariančne matrike

Na koncu nam je ostal še korak izračuna kovariančne matrike, ki ga sestavljajo tri gnezdene zanke *for*. V ščepcu smo povzporedili zunanji zanki *for*, tako da pri izvajanju ščepca uporabljamo $N \times N$ delovnih enot. Vsaka delovna enota znotraj ščepca izračuna eno vrednost v kovariančni matriki. Vzporedna implementacija koraka je bila zelo učinkovita, saj je izkoriščenost grafične kartice zaradi velikega števila uporabljenih delovnih enot in visoke računske zahtevnosti koraka v splošnem zelo dobra.

V nadaljevanju smo še dodatno izboljšali obstoječo implementacijo koraka. Opazili smo, da delovne enote med izvajanjem ščepca do kovariančne matrike dostopajo večkrat. Zaradi tega smo v tej implementaciji kovariančno matriko prenesli iz notranjega pomnilnika v hitrejši lokalni pomnilnik, s čimer smo zmanjšali čas dostopa do podatkov v kovariančni matriki. Poleg tega smo v tej implementaciji še dodatno izkoristili pomnilnik konstant, vanj smo namreč shranili vse sezname, ki se tekom izvajanja algoritma ne spreminjajo. Z opisanimi spremembami smo še dodatno optimizirali delovanje ščepca, zaradi česar smo pri uporabi te vzporedne implementacije koraka izmerili še boljše čase izvajanja koraka.

Poglavje 5 Rezultati meritev

V tem poglavju smo predstavili in podrobneje analizirali rezultate meritev učinkovitosti posameznih vzporednih pristopov. V začetku poglavja smo najprej primerjali povprečne čase izvajanja povzporejanih korakov algoritma pri uporabi posameznih vzporednih pristopov, nato pa smo nadaljevali s primerjavo učinkovitosti vzporednih pristopov ter s primerjavo časa izvajanja vzporedne in zaporedne implementacije algoritma pri različnih vhodnih parametrih algoritma.

Meritve smo opravljali na računalniku z štirijedrno CPE Intel i5-4670K, s hitrostjo ure 4.40 GHz, 8 GB RAMa ter grafično kartico AMD Radeon R9 290X, s hitrostjo ure 1 GHz in 4 GB notranjega pomnilnika.

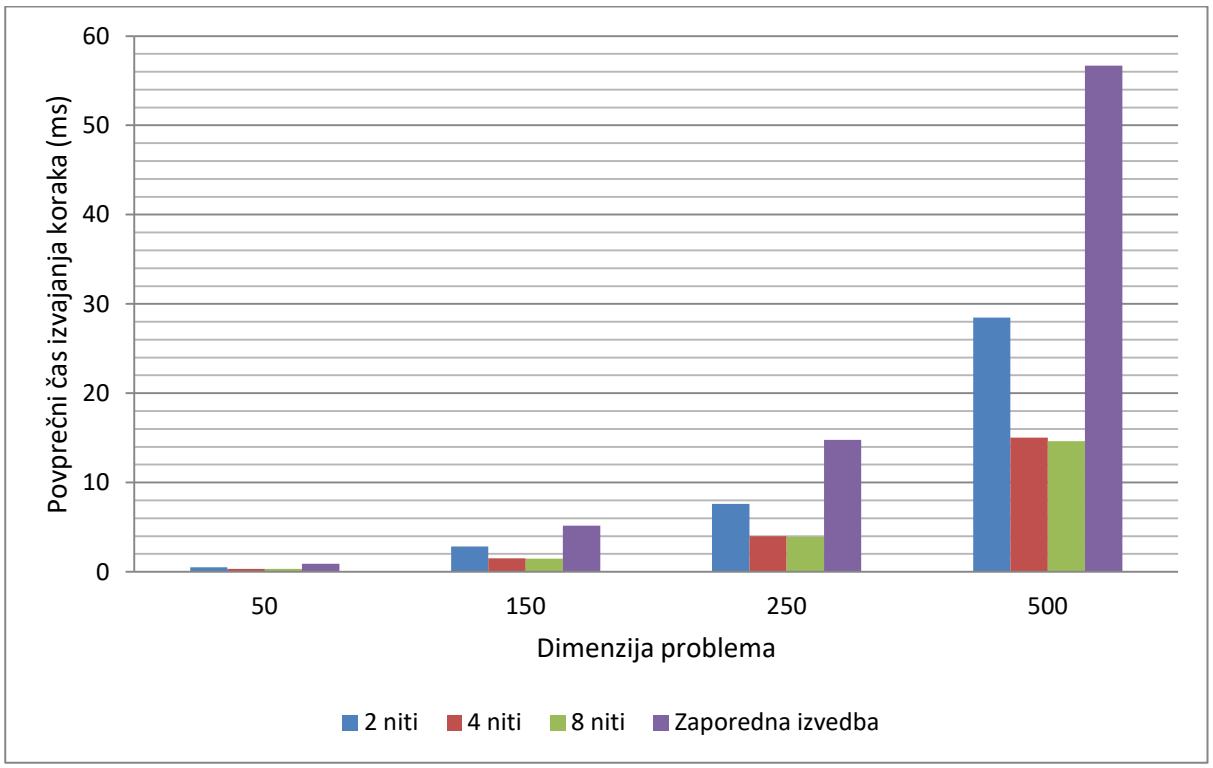
5.1 Meritve časov izvajanja povzporejanih korakov algoritma

Pri opravljanju vseh meritev smo uporabljali le najbolj optimizirane implementacije posameznih vzporednih pristopov. Zanimali so nas predvsem povprečni časi izvajanja posameznih korakov algoritma pri različnih vhodnih parametrih algoritma in pri različnih nastavitvah vzporednih pristopov. Pri ovrednotenju populacije smo uporabljali Rosenbrockovo funkcijo, ki smo jo podrobneje predstavili že v drugem poglavju.

Meritve smo opravljali pri dimenzijah problema $N = 50$, $N = 150$, $N = 250$ in N = 500 in pri velikostih populacije $\lambda = 20$, $\lambda = 100$ ter $\lambda = 250$. Izvajanje algoritma smo vedno prekinili po $N \times 10000$ evalvacijah. Zaradi velikega števila meritev smo v tem poglavju prikazali le meritve opravljene pri velikosti populacije $\lambda = 250$, saj smo v sklopu teh izmerili najvišje faktorje pohitritve.

5.1.1 Meritve vzporednega pristopa z večnitnostjo

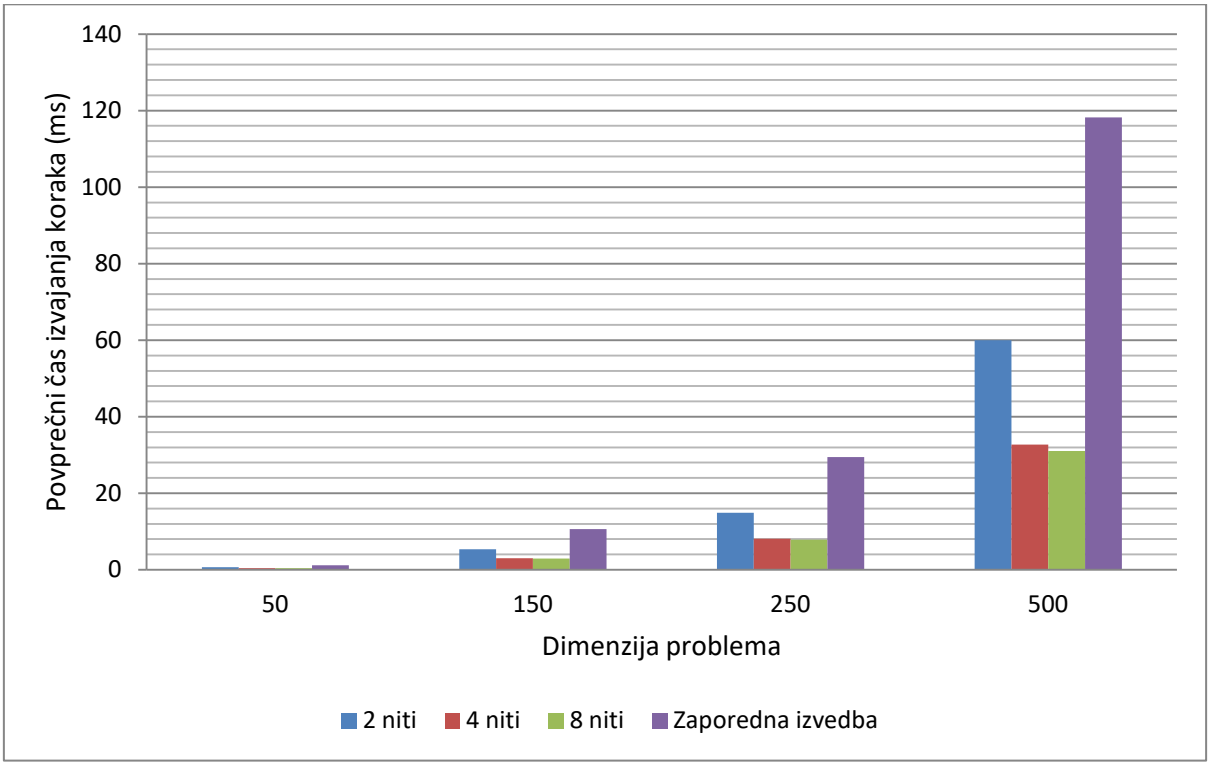
V tem poglavju smo prikazali povprečne čase izvajanja posameznih povzporejanih korakov pri uporabi različnega števila niti CPE. Naša CPE, podobno kot večina novejših CPE podjetja *Intel* uporablja t.i. tehnologijo "*Hyper-Threading*" (v nadaljevanju HTT), zaradi katere operacijski sistem računalnika vsako fizično jedro CPE obravnava kot dve virtualni (logični) jedri. Njena uporaba lahko pripomore k bolj učinkovitemu vzporednemu izvajanju programov [15]. Na našem gostitelju smo lahko zaradi HTT pri vzporednemu izvajanju korakov uporabljali 8 niti, čeprav ima naša CPE le 4 jedra. Tu še omenimo, da je zmogljivost dodatnih niti, ki jih imamo na voljo zaradi uporabe HTT nižja kot pa zmogljivost dejanskih niti CPE.



Slika 5.1 : Histogram meritev časa izvajanja koraka za generiranje populacije.

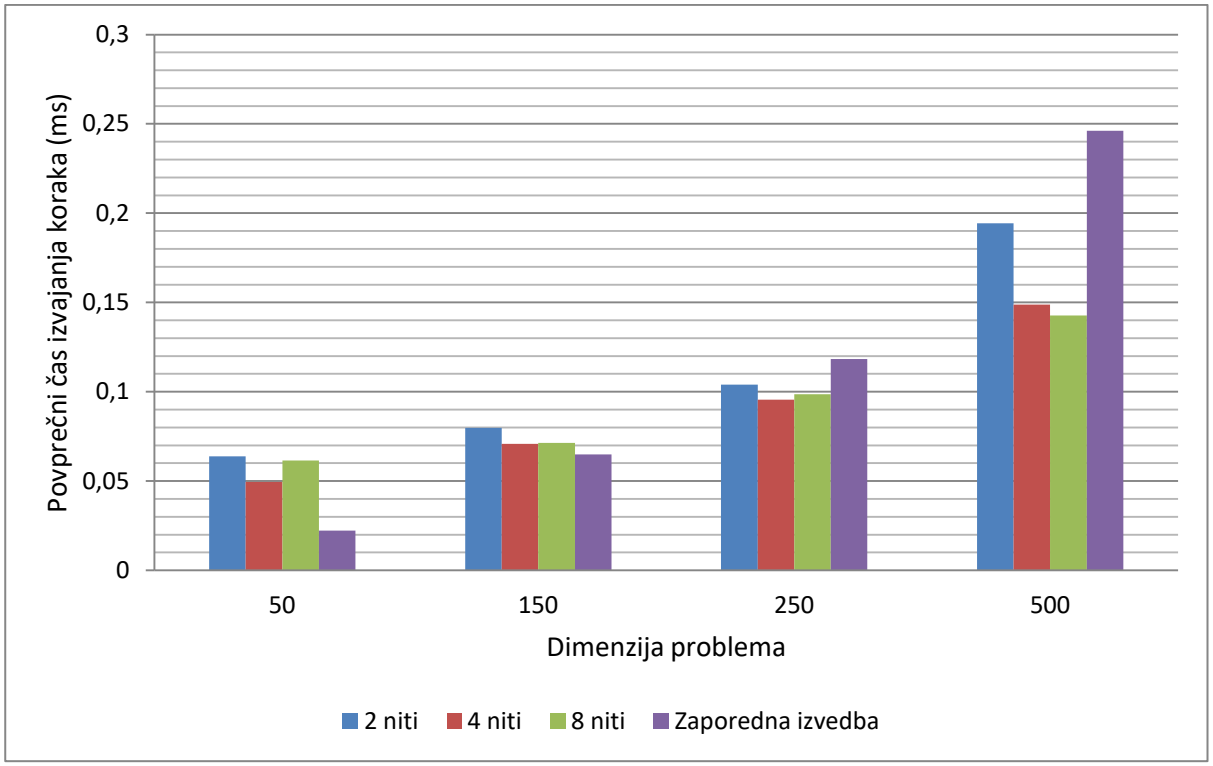
Kot je razvidno iz slike 5.1, ki prikazuje rezultate meritev opravljenih pri vzporednemu izvajanju koraka generiranja populacije, se nam uporaba večnitnosti splača že pri zelo majhnih dimenzijah problema. Kot smo že omenili, je maksimalna pohitritev koraka enaka številu uporabljenih niti. Prav zaradi tega dejstva se je uporaba dveh niti izkazala za najslabšo, saj je maksimalna pohitritev koraka pri uporabi takega števila niti le dvakratna.

V okviru naših meritev smo zaradi tega najboljše čase izvajanja dosegli pri uporabi štirih in osmih niti. Pri manjših dimenzijah problema se je uporaba štirih in osmih niti izkazala za skoraj enako učinkovito, pri večjih dimenzijah problema pa smo najboljše rezultate dosegli z uporabo osmih niti.



Slika 5.2 : Histogram meritev časa izvajanja koraka za izračun kovariančne matrike.

Na sliki 5.2 smo prikazali rezultate meritev pri vzporednemu izvajanju koraka za izračun kovariančne matrike. Kot lahko vidimo, je zgornji histogram zelo podoben histogramu na sliki 5.1, opazimo lahko le, da se korak izračuna kovariančne matrike v primerjavi s korakom generiranja populacije nekoliko bolje povzporeja, kar lahko pripišemo predvsem njegovi višji računski zahtevnosti. Uporaba štirih in osmih nitih je bila v večini primerov skoraj enako učinkovita, razlike opazimo šele pri zelo velikih dimenzijah problema, pri katerih se uporaba osmih niti zopet izkaže za bolj učinkovito.

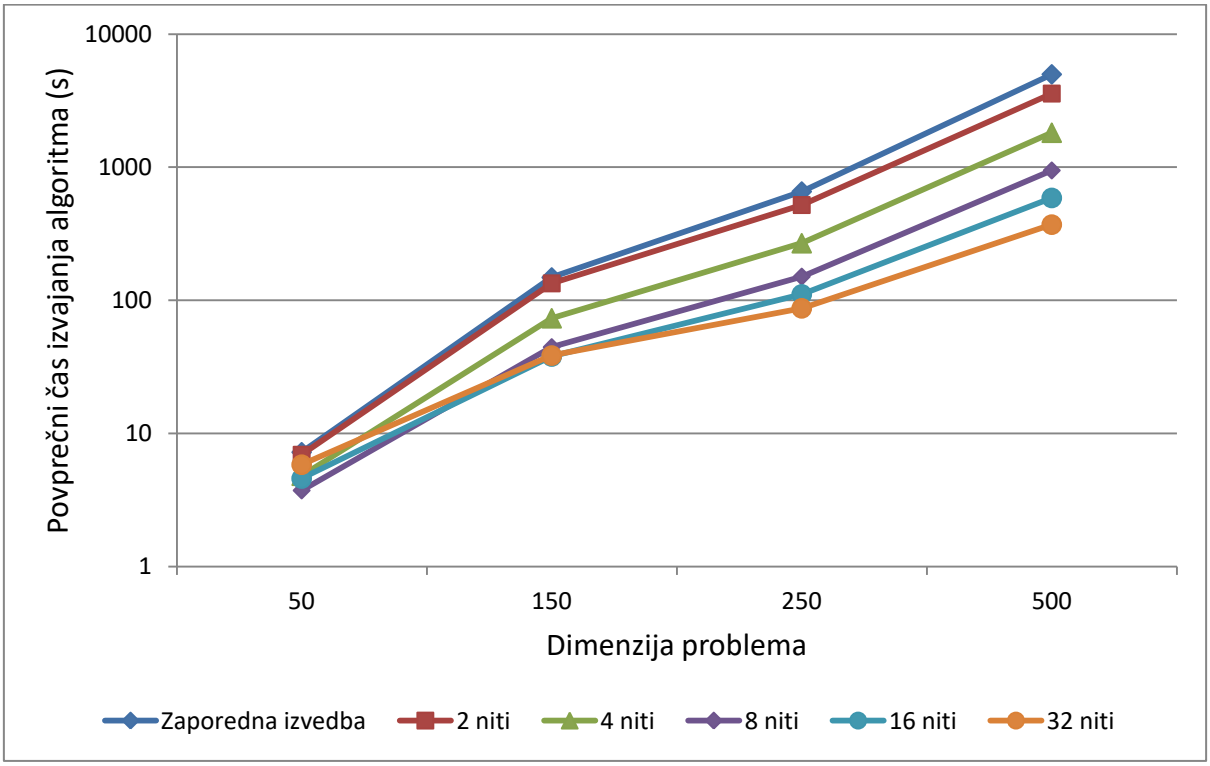


Slika 5.3 : Histogram meritev časa izvajanja koraka za ovrednotenje populacije.

Na sliki 5.3 smo prikazali še rezultate meritev za korak ovrednotenja populacije, ki so bili nekoliko bolj zanimivi. Pri majhnih dimenzijah problema (v našem primeru $N = 50$ in $N = 150$) se zaporedna implementacija koraka izkaže za boljšo od vzporedne, saj je računska zahtevnost koraka v teh primerih preprosto prenizka, da bi lahko njegovo vzporedno izvajanje opravičili. S povečevanjem dimenzije problema pa narašča tudi računska zahtevnost koraka, zaradi česar smo pri dimenziji problema $N = 250$ že izmerili pohitritev v času izvajanja koraka. Najboljše rezultate smo tudi to pot dosegli pri uporabi štirih in osmih niti.

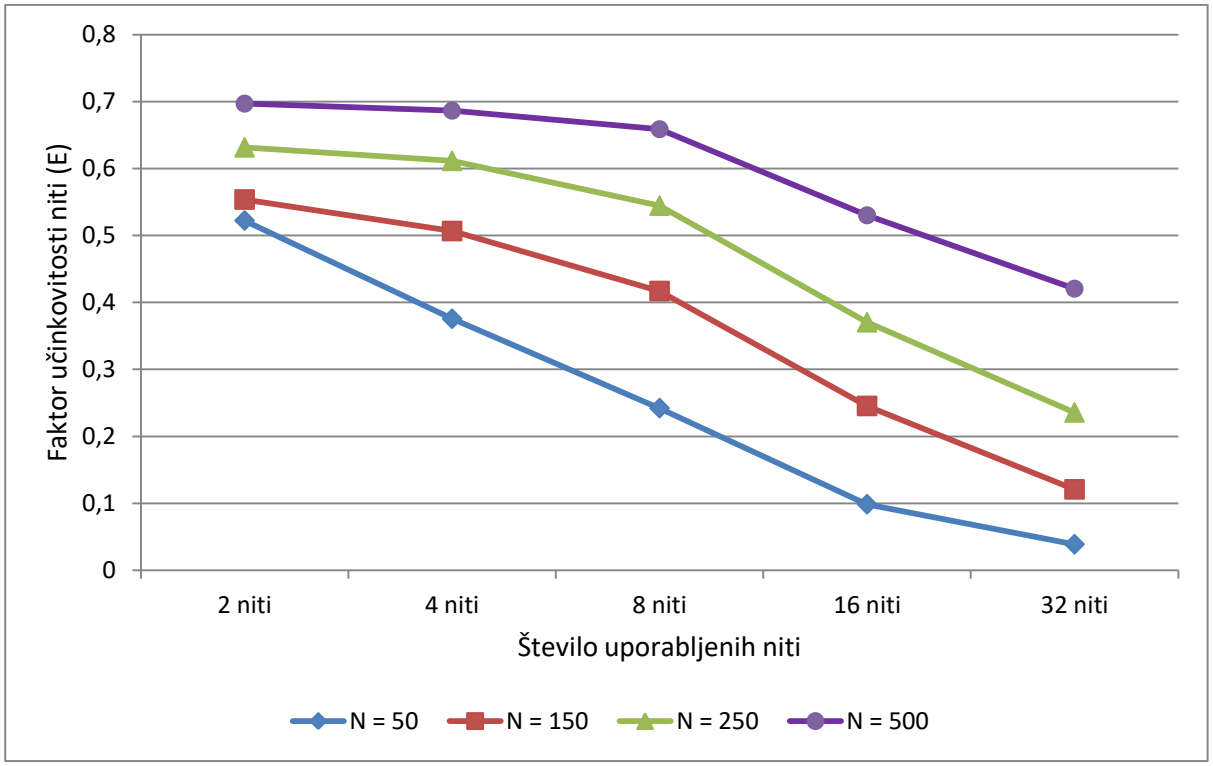
Nekaj vzorčnih meritev smo opravili tudi na računalniku v lasti Instituta "Jožef Stefan". Sestavljata ga dve CPE Intel Xeon E5-2680 v3 s hitrostjo ure 2,50 GHz in 12 jedri z vključeno HTT, 1024 GB RAMa ter grafična kartica nVidia Tesla K80, ki vsebuje dva procesorja. Institutski računalnik tako podpira uporabo 48 niti, medtem ko smo jih na našem gostitelju lahko uporabljali največ 8.

Pri opravljanju meritev nas je predvsem zanimala skalabilnost pristopa z večnitnostjo. Meritve smo opravljali pri velikosti populacije $\lambda = 250$ in pri dimenzijah problema $N = 50$, $N = 150$, $N = 250$ in $N = 500$. Rezultate meritev smo prikazali na sliki 5.4.



Slika 5.4: Graf primerjave časov izvajanja zaporedne in vzporedne implementacije algoritma pri uporabi različnega števila niti.

Kot je razvidno iz slike 5.4, je skalabilnost pristopa z večnitnostjo zelo dobra. Pri majhnih dimenzijah problema se nam najbolj splača uporaba 8-ih ali 16-ih niti, pri večjih dimenzijah problema pa smo najboljše rezultate dosegli z uporabo 32-ih niti CPE. V primerjavi z našim gostiteljem so doseženi faktorji pohitritve na institutskem računalniku višji, razen pri uporabi dveh in štirih niti, kar pa lahko pripišemo nižji hitrosti ure CPE. Naš najvišji doseženi faktor pohitritve, ki smo ga izmerili pri uporabi 32 niti in pri dimenziji problema $N = 500$ je znašal kar 13,45.



Slika 5.5: Graf učinkovitosti uporabe različnega števila niti pri različnih dimenzijah problema.

Na sliki 5.5 smo prikazali še graf učinkovitosti uporabe različnega števila niti. Faktorje učinkovitost niti (5.1) smo izračunali z uporabo rezultatov meritev opravljenih na institutskem računalniku, ki smo jih prikazali že na sliki 5.4.

$$E = \frac{S_N}{B} \tag{5.1}$$

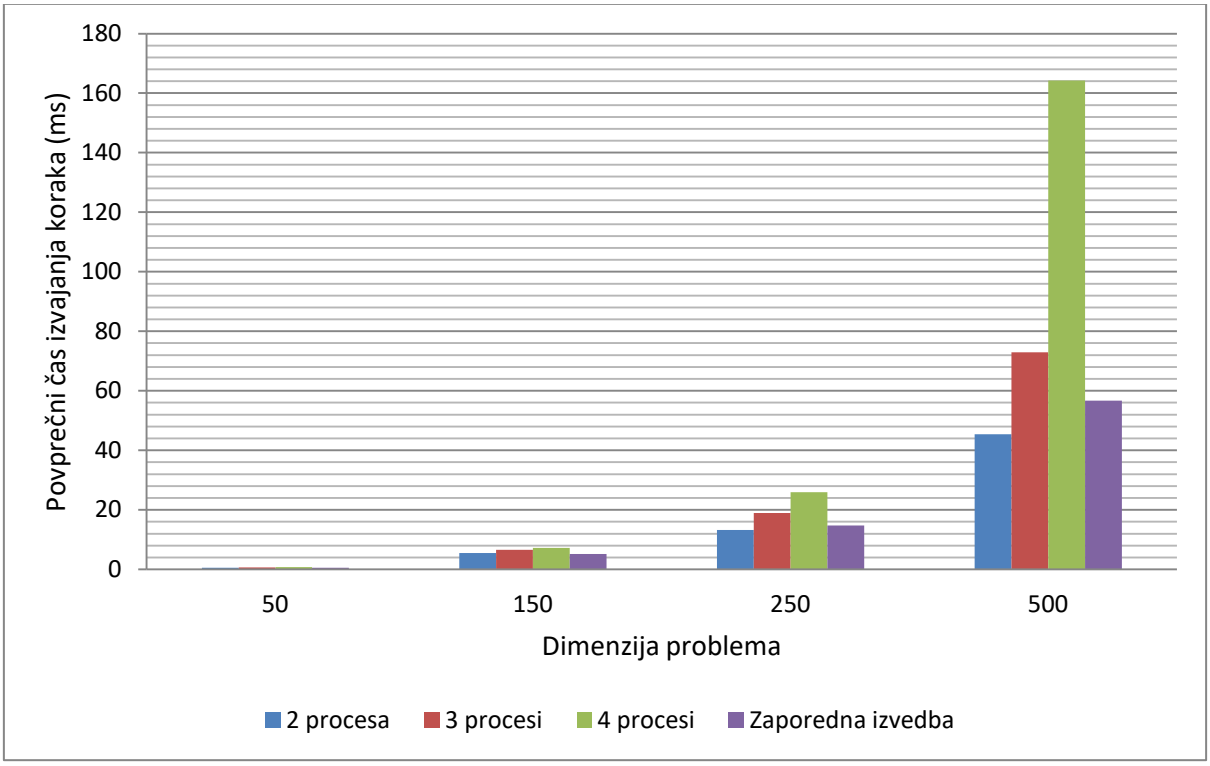
V zgornji enačbi velja, da:

- B ustreza številu uporabljenih niti pri vzporednemu izvajanju algoritma.
- S_N ustreza izmerjenemu faktorju pohitritve pri vzporednem izvajanju algoritma pri dimenziji problema N in pri uporabi B niti.

Iz zgornjega grafa lahko razberemo, da faktor učinkovitosti niti narašča s povečevanjem dimenzije problema in z zmanjševanjem števila uporabljenih niti. Zaradi tega smo najvišji faktor učinkovitosti izmerili pri dimenziji problema $N = 500$ pri uporabi dveh niti, najnižjega pa pri dimenziji problema $N = 50$ pri uporabi 32-ih niti.

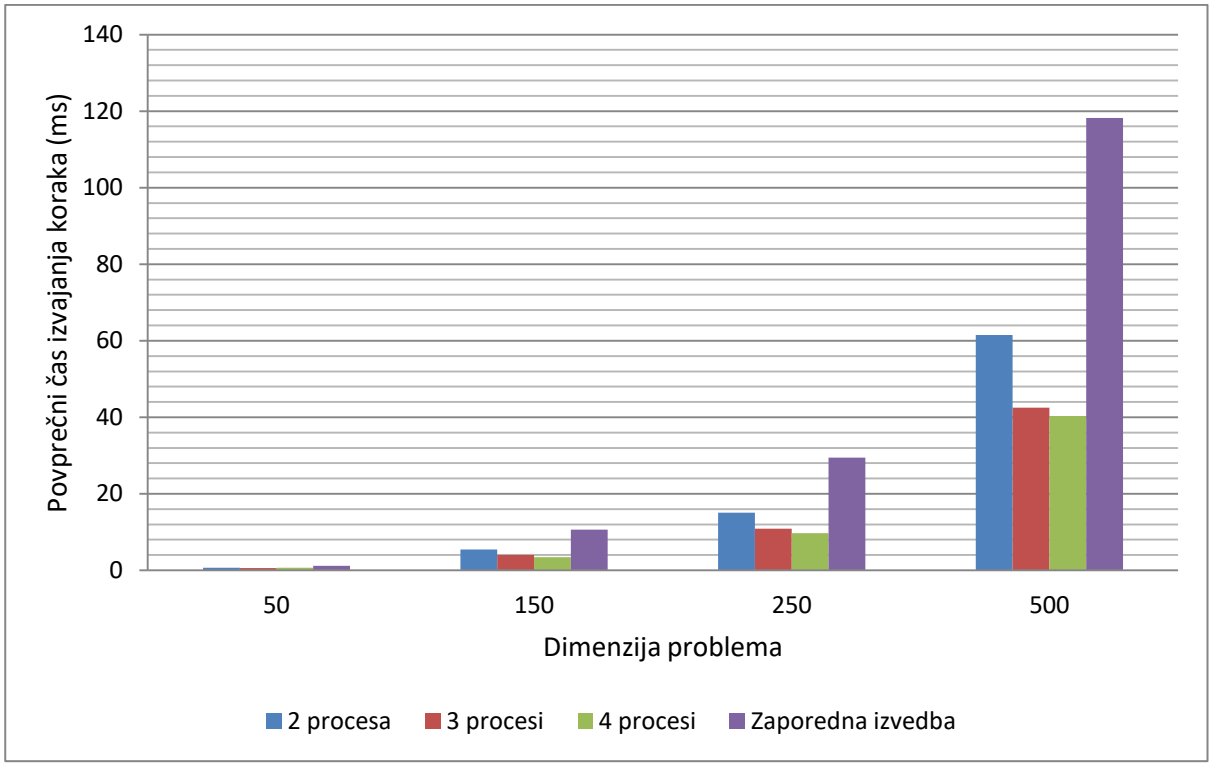
5.1.2 Meritve vzporednega pristopa MPI

V tem podpoglavju smo prikazali rezultate meritev pridobljenih pri vzporednemu izvajanju algoritma z uporabo vmesnika MPI. V sklopu naših meritev smo korake algoritma izvajali z uporabo dveh, treh in štirih procesov MPI. Tu še ponovno omenimo, da smo omrežje MPI simulirali z uporabo CPE, zaradi česar so bili stroški komunikacije MPI v sklopu naših meritev veliko manjši kot bi bili pri izvajanju algoritma v dejanskem omrežju MPI.



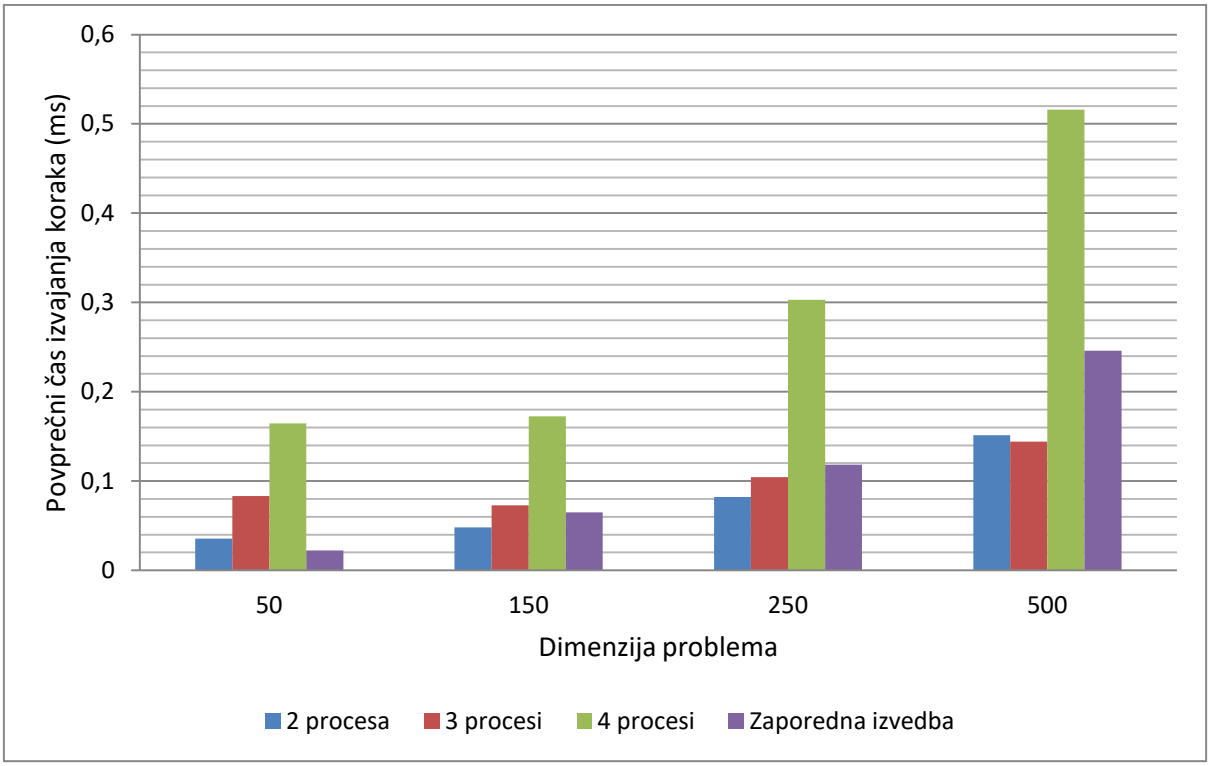
Slika 5.6 : Histogram meritev časa izvajanja koraka za generiranje populacije.

Na sliki 5.6 smo prikazali rezultate meritev pri vzporednemu izvajanju koraka za generiranje populacije. Prva stvar, ki jo lahko opazimo, je slaba skalabilnost in učinkovitost pristopa MPI. To lahko pripišemo predvsem relativno visokim stroškom komunikacije med procesi MPI, zaradi katerih smo pohitritve v času izvajanja koraka izmerili le pri uporabi dveh procesov MPI, saj so bili stroški komunikacije v tem primeru najnižji.



Slika 5.7: Histogram meritev časa izvajanja koraka za izračun kovariančne matrike.

Na sliki 5.7 smo prikazali rezultate meritev pri vzporednemu izvajanju koraka za izračun kovariančne matrike. V nasprotju s prejšnjim korakom generiranja populacije lahko pri tem koraku opazimo veliko boljšo skalabilnost, kar lahko predvsem pripišemo njegovi višji računski zahtevnosti. Skalabilnost pristopa je pri zelo majhnih dimenzijah problema še vedno slaba, vendar se ta s povečevanjem dimenzije problema (in s tem računske zahtevnosti koraka) izboljšuje. Pri majhnih dimenzijah problema se je zaradi tega v našem primeru za najboljšo izkazala uporaba dveh procesov MPI, pri velikih dimenzijah problema pa smo najboljše čase izvajanja dosegli pri uporabi štirih procesov MPI.

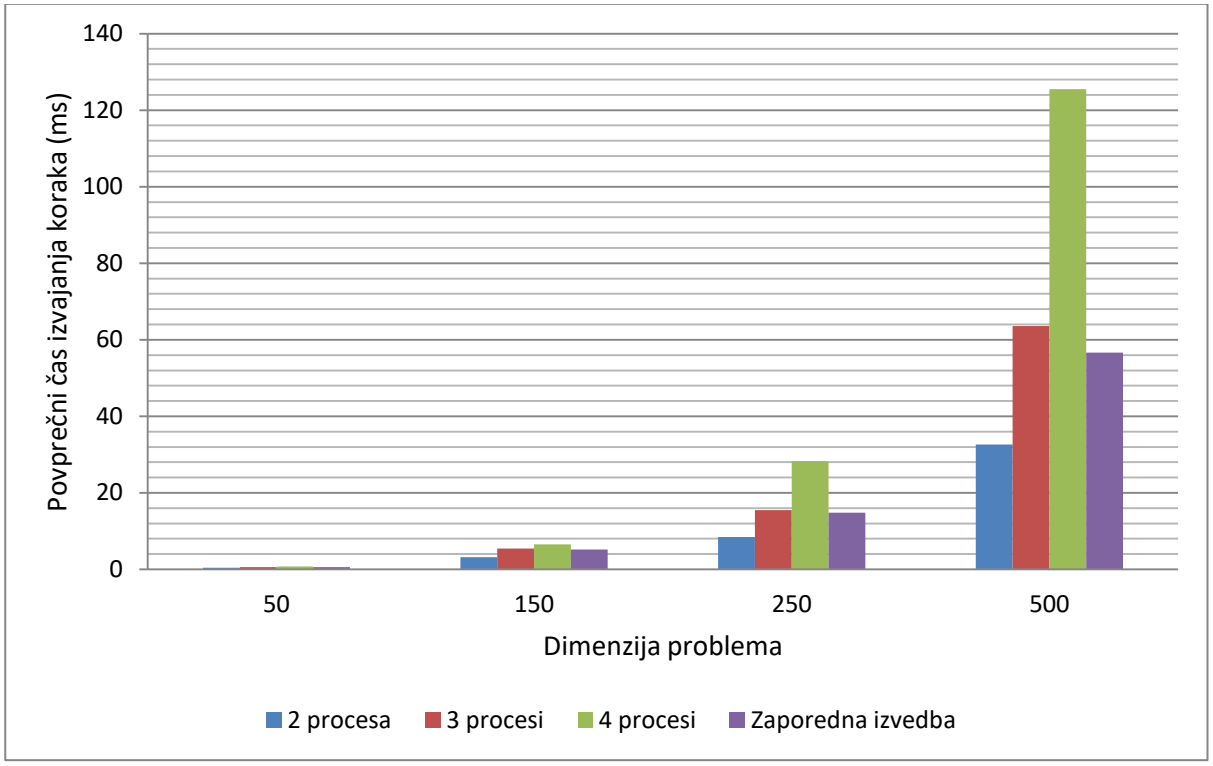


Slika 5.8: Histogram meritev časa izvajanja koraka za ovrednotenje populacije.

Rezultate meritev za korak ovrednotenja populacije smo prikazali na sliki 5.8. Kot lahko vidimo, uporabe MPI pri zelo majhnih dimenzijah problema zaradi nizke računske zahtevnosti koraka ne moremo opravičiti. Pri manjših dimenzijah problema (v našem primeru $N = 150$ in $N = 250$) se nam je najbolj obrestovala uporaba dveh procesov MPI, pri večjih dimenzijah problema pa smo najboljše rezultate izmerili z uporabo treh procesov MPI.

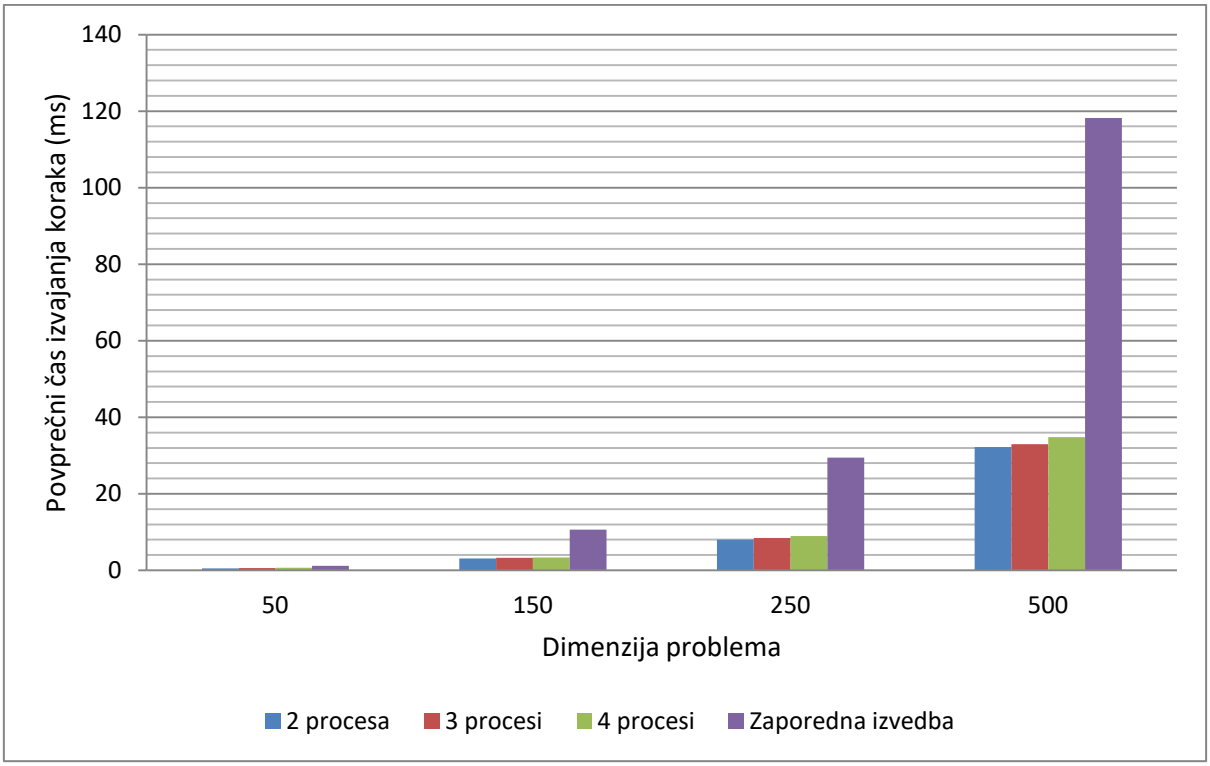
5.1.3 Meritve vzporednega pristopa MPI z večnitnostjo

V okviru tega poglavja smo predstavili meritve opravljene pri uporabi pristopa MPI, ki je izkoriščal še niti CPE. Pri izvajanju korakov algoritma smo uporabljali osem niti. Ker smo v našem primeru omrežje MPI simulirali z uporabo CPE, so si procesi morali niti medsebojno deliti, zaradi česar delovanje tega pristopa ni bilo optimalno. Kljub temu smo z rezultati meritev zadovoljni, te smo prikazali na slikah 5.9, 5.10 in 5.11.



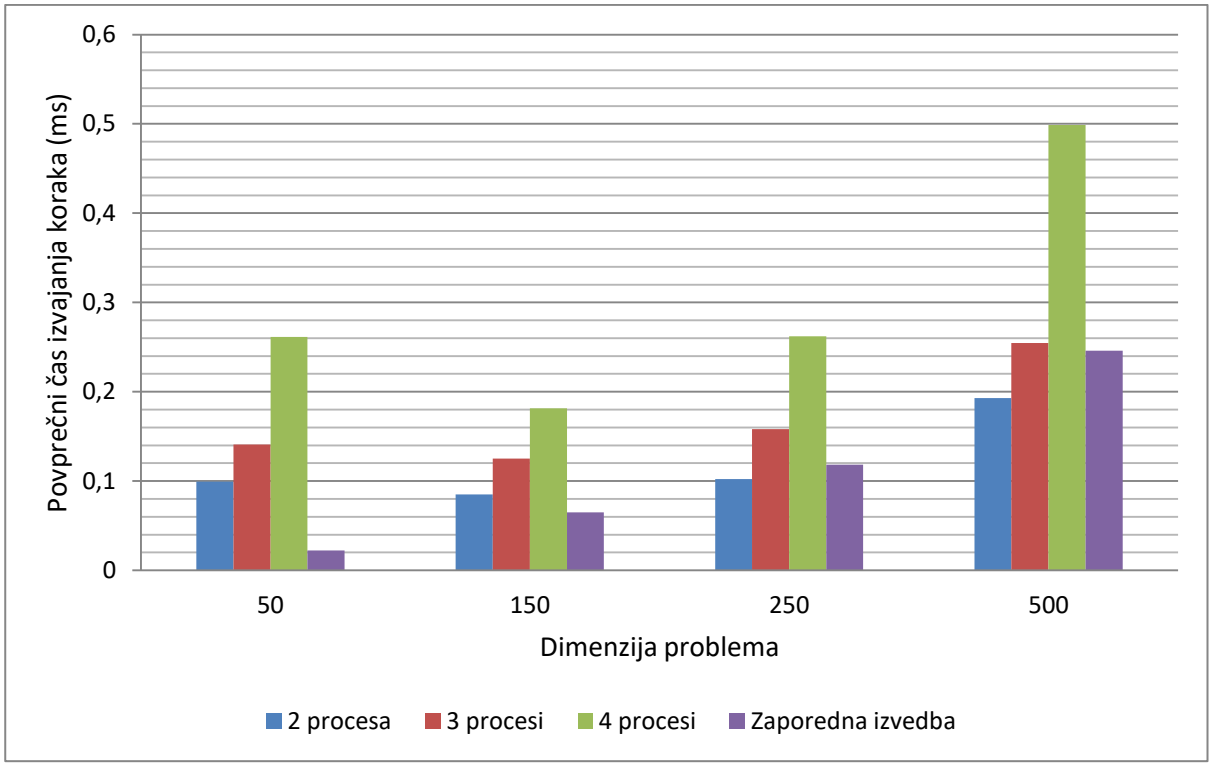
Slika 5.9: Histogram meritev časa izvajanja koraka za generiranje populacije.

Na sliki 5.9 smo prikazali rezultate meritev pri vzporednemu izvajanju koraka za generiranje populacije. Kot lahko vidimo, je skalabilnost pristopa podobno kot pri osnovnem pristopu MPI slaba. V nasprotju s slednjim pristopom smo tu izmerili nekoliko boljše čase izvajanja koraka, saj smo z uporabo večnitnosti še dodatno pohitrili izračune v koraku. Dejansko pohitritev v času izvajanja koraka pa smo enako kot pri osnovnem pristopu MPI izmerili le pri uporabi dveh procesov MPI, saj so učinkovitost oz. skalabilnost pristopa še vedno v veliki meri omejevali relativno visoki stroški komunikacije med procesi.



Slika 5.10: Histogram meritev časa izvajanja koraka za izračun kovariančne matrike.

Rezultate meritev pri izvajanju koraka za izračun kovariančne matrike smo prikazali na sliki 5.10. Kot lahko vidimo, so si časi izvajanja koraka pri uporabi dveh, treh in štirih proceosv MPI procesov zelo podobni. Če bi algoritem izvajali na dejanskem omrežju MPI, bi verjetno pri uporabi treh in štirih procesov izmerili boljše čase izvajanja kot pri uporabi dveh procesov, saj bi vsak računalnik v omrežju imel svoj lasten nabor niti CPE, zaradi česar bi bilo delovanje pristopa bolj optimalno. Kljub temu smo pri uporabi tega vzporednega pristopa izmerili boljše čase izvajanja kot pri uporabi osnovnega pristopa MPI, kar lahko pripišemo uporabi večnitnosti.

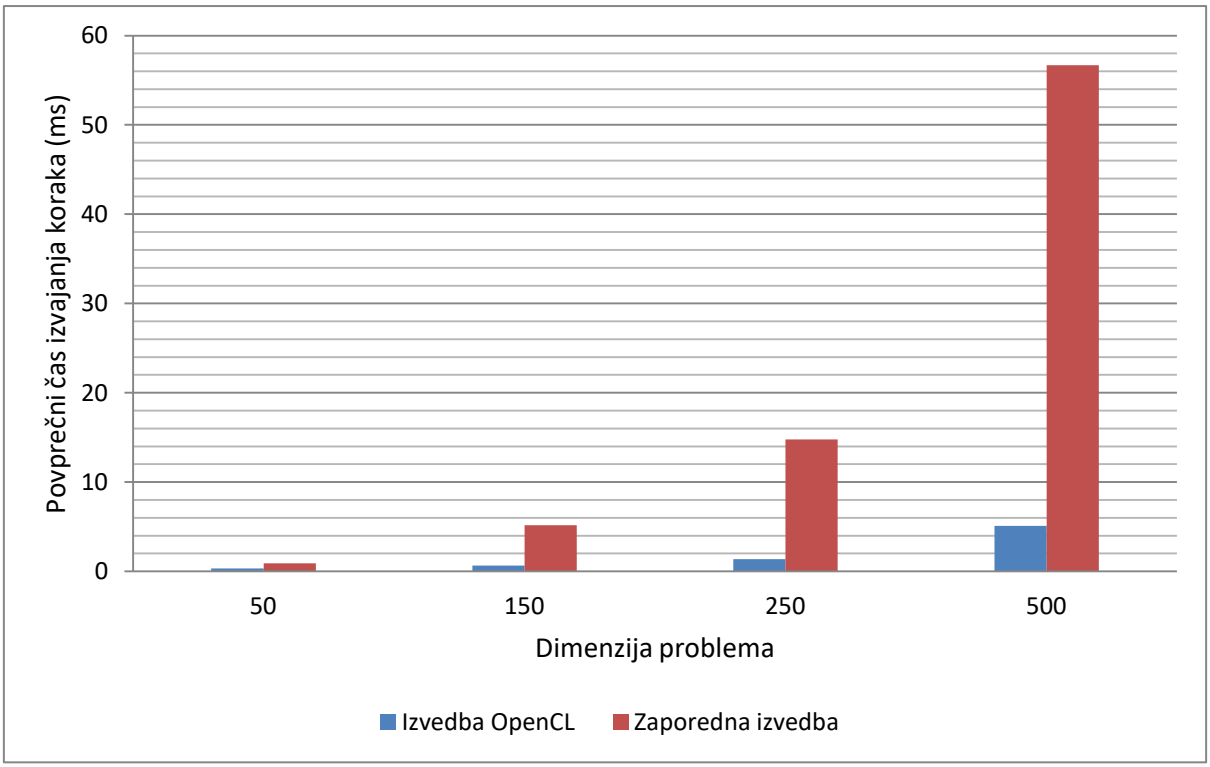


Slika 5.11: Histogram meritev časa izvajanja koraka za ovrednotenje populacije.

Rezultate meritev pri koraku za ovrednotenje populacije smo prikazali na sliki 5.11, iz katere je razvidno, da je skalabilnost pristopa pri majhnih dimenzijah problema zaradi nizke računske zahtevnosti koraka zelo slaba, s povečevanjem dimenzije problema pa se ta počasi izboljšuje. V sklopu naših meritev se je za najbolj učinkovito zopet izkazala uporaba dveh procesov, saj so stroški komunikacije pri njeni uporabi najnižji, vendar lahko opazimo, da postaja uporaba treh procesov s povečevanjem dimenzije problema vse bolj učinkovita. V nasprotju z osnovnim pristopom MPI pa so bili časi izvajanja koraka pri uporabi tega pristopa nekoliko slabši, saj smo z uporabo večnitnosti v korak vnesli le dodatne stroške komunikacije, ki jih pridobljene pohitritve v izračunih zaradi majhne računske zahtevnosti koraka ne morejo odtehtati.

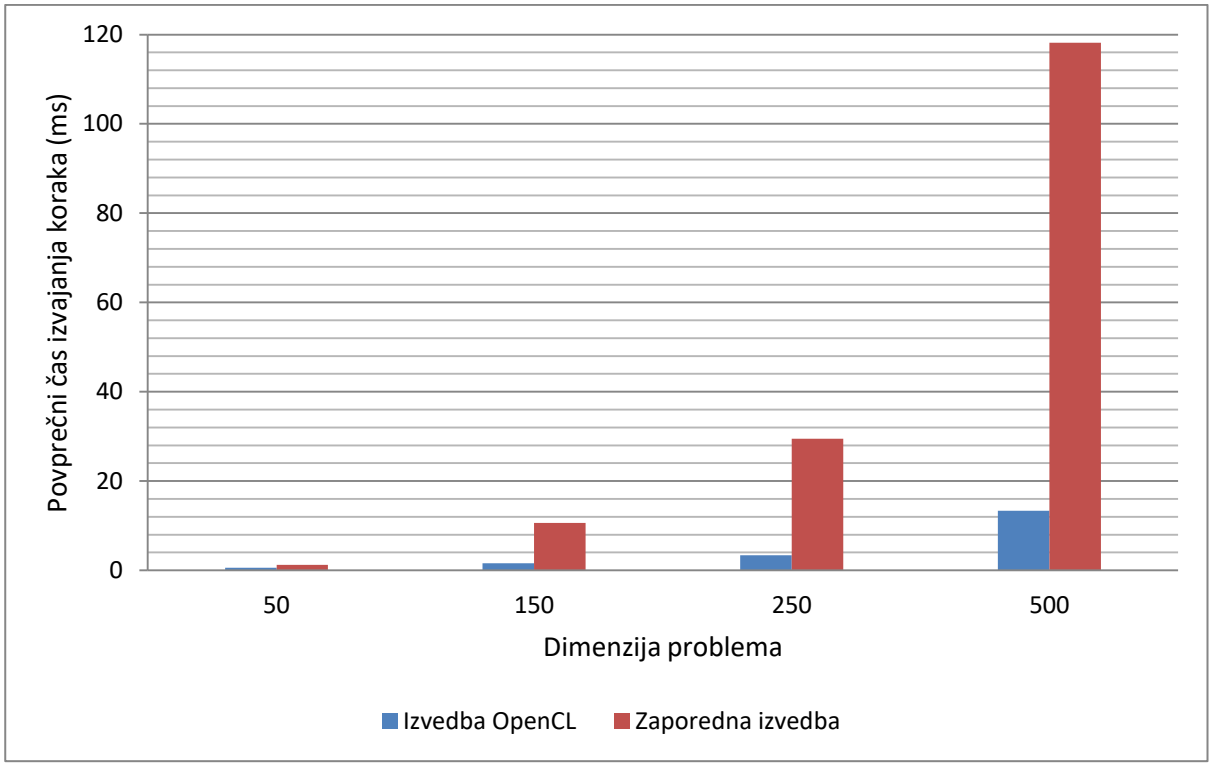
5.1.4 Meritve vzporednega pristopa OpenCL

Na koncu smo opravili še meritve časa izvajanja korakov algoritma z uporabo vmesnika OpenCL. Meritve smo tu opravljali nekoliko drugače, saj smo morali ščepce za pravilno delovanje izvajati s fiksnim številom delovnih enot, odvisnim od velikosti populacije in dimenzije problema.



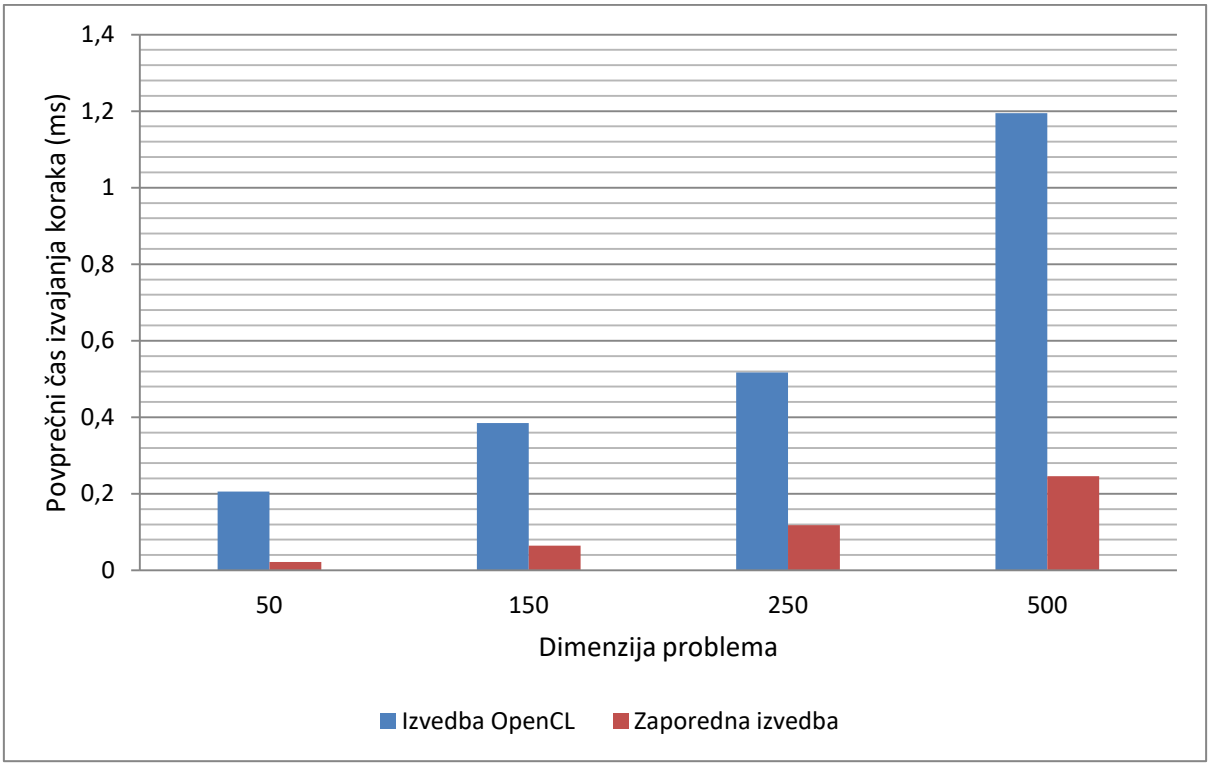
Slika 5.12: Histogram meritev časa izvajanja koraka generiranja populacije.

Pri opravljanju meritev smo zopet pričeli s korakom generiranja populacije, ki ga v ščepcu izvajamo z uporabo $\lambda \times N$ delovnih enot. Iz histograma na sliki 5.12 je razvidno, da lahko uporabo OpenCL pri vzporednemu izvajanju tega koraka opravičimo že pri zelo majhnih dimenzijah problema. To lahko pripišemo predvsem velikosti populacije, zaradi katere smo pri izvajanju ščepca uporabljali veliko število delovnih enot, kar je pripomoglo k dobri izkoriščenosti grafične kartice. To je razvidno tudi iz zgornjega histograma, saj faktorji pohitritve s povečevanjem dimenzije problema zaradi vse boljše izkoriščenosti grafične kartice le še naraščajo. Najvišji faktor pohitritve smo zaradi tega izmerili pri dimenziji problema $N = 500$, ta je znašal kar 11,1.



Slika 5.13: Histogram meritev časa izvajanja koraka izračuna kovariančne matrike.

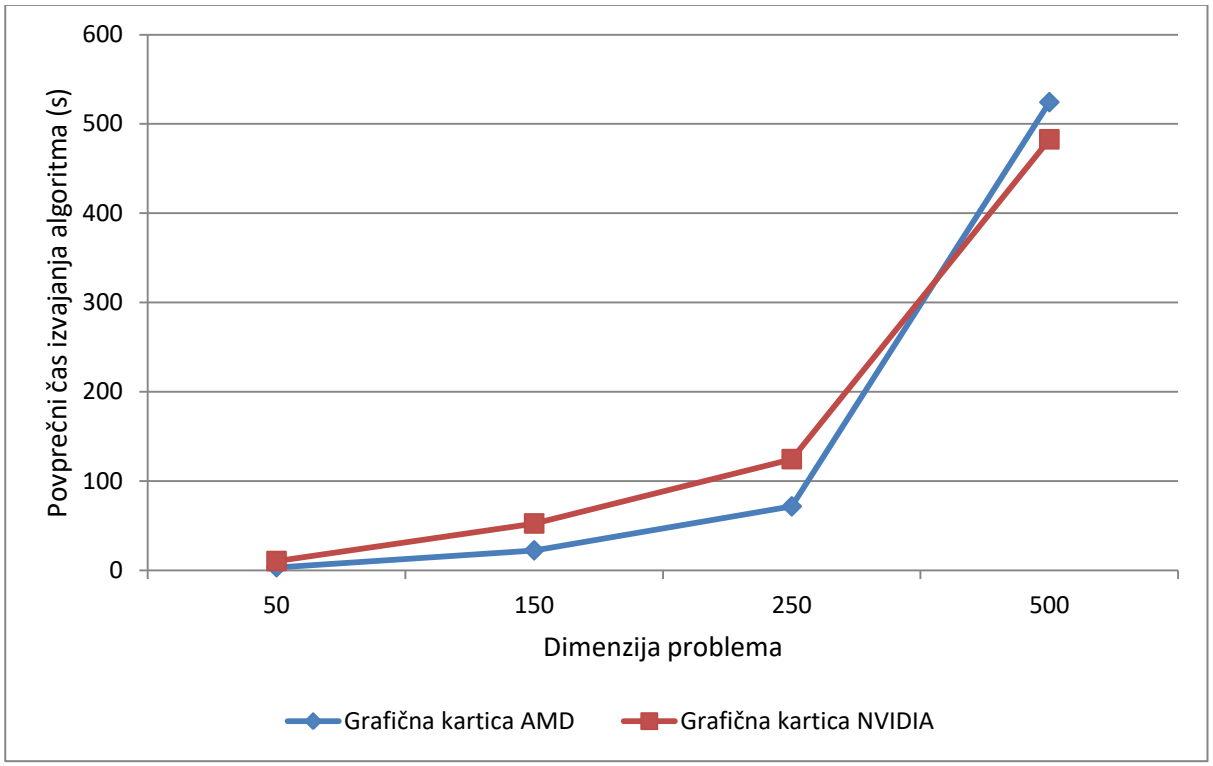
Na sliki 5.13 smo prikazali rezultate meritev za korak izračuna kovariančne matrike, ki ga izvajamo z uporabo $N \times N$ delovnih enot. Podobno kot pri prejšnjem koraku generiranja populacije lahko uporabo OpenCL opravičimo že pri zelo majhnih dimenzijah problema, saj je korak izračuna kovariančne matrike računsko zelo zahteven. Njegova uporaba se nam je najbolj obrestovala pri velikih dimenzijah problema, saj pri takih dimenzijah problema ščepec izvajamo z velikim številom delovnih enot, zaradi česar je izkoriščenost grafične kartice zelo dobra. Naš najvišji doseženi faktor pohitritve, ki smo ga izmerili pri dimenziji problema $N = 500$ je znašal 8,86.



Slika 5.14: Histogram meritev časa izvajanja koraka za ovrednotenje populacije.

Na sliki 5.14 smo prikazali še rezultate meritev pri vzporednemu izvajanju koraka ovrednotenja populacije. Ta korak je izmed vseh povzporejanih korakov algoritma računsko najmanj zahteven, poleg tega pri njegovem izvajanju uporabljamo le λ delovnih enot, zaradi česar je izkoriščenost grafične kartice pri izvajanju tega koraka zelo slaba. Kot je razvidno iz zgornje slike, koraka zaradi tega nismo uspeli pohitriti, saj so bile pridobljene pohitritve v izračunih zaradi preslabe izkoriščenosti grafične kartice preprosto premajhne, da bi lahko odtehtale stroške komunikacije med gostiteljem in napravo.

Nekaj vzorčnih meritev smo tudi to pot opravili na računalniku v lasti Instituta "Jozef Stefan". Ta vsebuje grafično kartico NVIDIA Tesla K80, ki je veliko zmogljivejša kot grafična kartica AMD na našem gostitelju. V sklopu meritev so nas predvsem zanimale razlike v učinkovitosti uporabe grafične kartice AMD in grafične kartice NVIDIA. Meritve smo opravljali pri velikosti populacije $\lambda = 250$ in pri dimenzijah problema $N = 50$, $N = 150$, $N = 250$ in $N = 500$. Rezultate meritev smo prikazali na sliki 5.15.

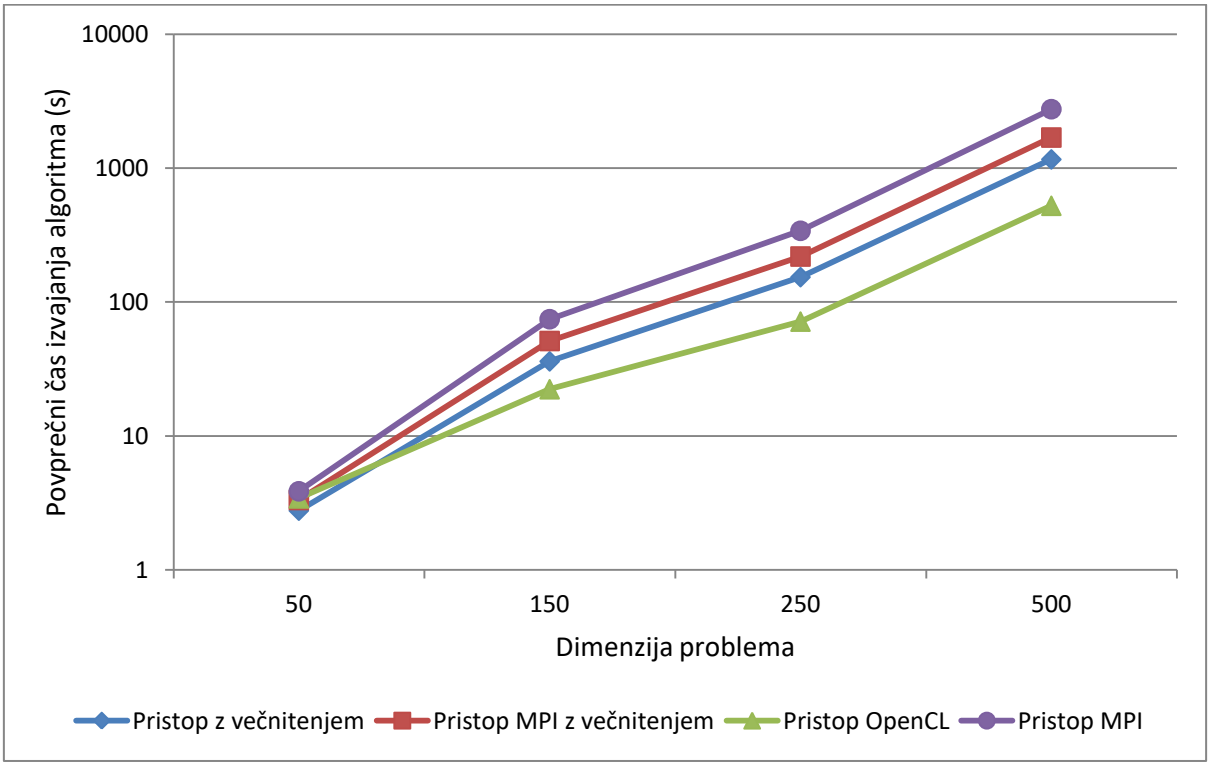


Slika 5.15: Primerjava časov izvajanja algoritma na grafičnih karticah AMD in NVIDIA.

Kot lahko razberemo iz slike 5.15 ima izbira grafične kartice velik vpliv na učinkovitost izvajanja algoritma z uporabo OpenCL. Uporaba grafične kartice NVIDIA se je kljub veliko višji zmogljivosti skoraj v vseh primerih izkazala za manj učinkovito kot uporaba grafične kartice AMD, kar lahko pripišemo predvsem dejstvu, da je OpenCL s strani AMD veliko bolje podprt kot s strani NVIDIE, ki je primarno usmerjena v uporabo svoje lastne platforme CUDA. Pri dimenziji problema $N = 500$ smo sicer pri uporabi grafične kartice NVIDIA izmerili boljše čase izvajanja kot pri uporabi grafične kartice AMD, vendar lahko to pripišemo njeni veliko višji zmogljivosti. Iz pridobljenih rezultatov lahko sklepamo, da bi pri uporabi enakovrednih grafičnih kartic AMD in NVIDIA, grafične kartice AMD zaradi boljše podpore OpenCL v vseh primerih prevladovale nad grafičnimi karticami NVIDIA.

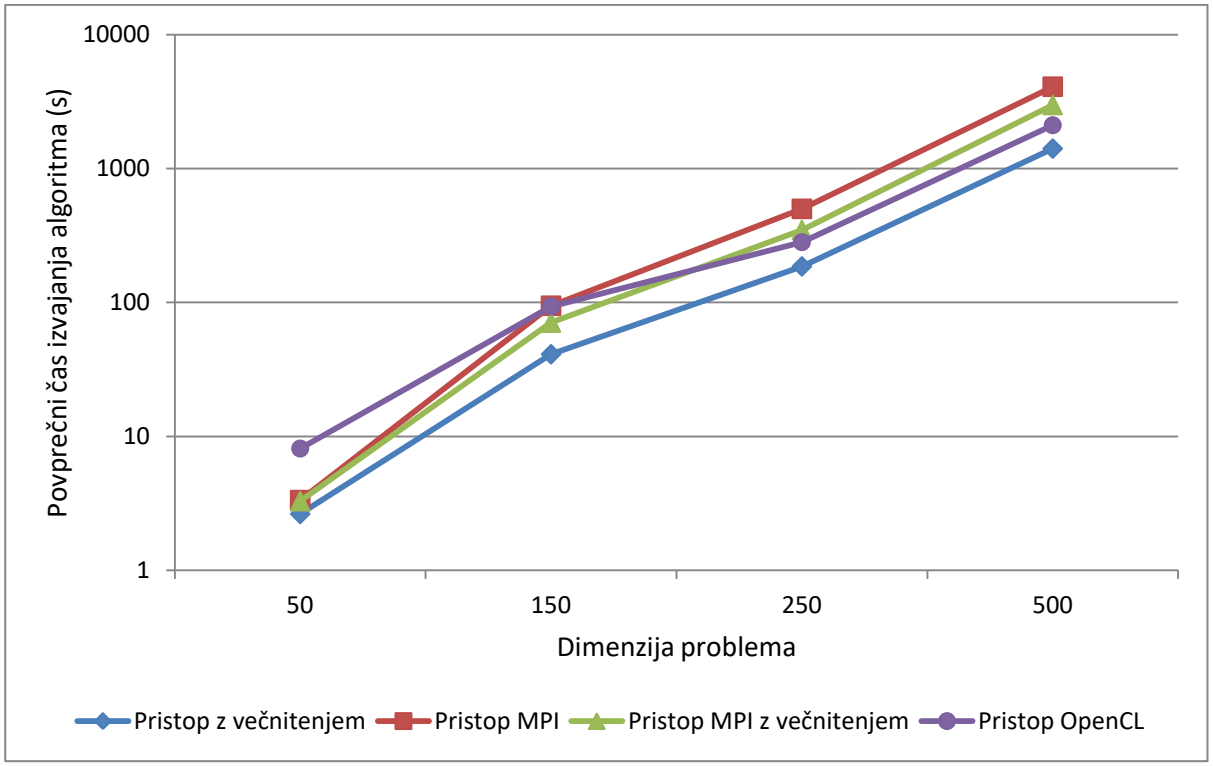
5.1.5 Primerjava učinkovitosti vzporednih pristopov

V tem podpoglavju smo prikazali še primerjavo učinkovitosti uporabe posameznih vzporednih pristopov pri različnih vhodnih parametrih algoritma. Prikazali smo le rezultate meritev pri velikostih populacije $\lambda = 250$ in $\lambda = 20$, saj so bili rezultati v sklopu teh najzanimivejši. Pri opravljanju meritev smo nastavitve delovanja posameznih vzporednih pristopov priredili tako, da smo dosegli najboljše možne čase izvajanja algoritma. To smo storili na podlagi rezultatov meritev, ki smo jih prikazali v sklopu prejšnjih podpoglavij.



Slika 5.16: Primerjava časov izvajanja algoritma pri uporabi različnih vzporednih pristopov in pri velikosti populacije $\lambda = 250$.

Na sliki 5.16 smo prikazali primerjave časov izvajanja algoritma pri velikosti populacije $\lambda = 250$. Najboljše rezultate smo dosegli z uporabo pristopa OpenCL, predvsem zaradi tega, ker smo lahko računsko moč grafične kartice zaradi visoke računske zahtevnosti reševanih problemov zelo dobro izkoristili. Zelo dobre rezultate smo dosegli tudi pri uporabi pristopa večnitnosti, za nekoliko manj učinkovito se je izkazala uporaba obeh pristopov MPI, predvsem zaradi visokih stroškov komunikacije, ki nastanejo pri njuni uporabi.



Slika 5.17: Primerjava časov izvajanja algoritma pri uporabi različnih vzporednih pristopov in pri velikosti populacije $\lambda = 20$.

Na sliki 5.17 smo prikazali še čase izvajanja algoritma pri velikosti populacije $\lambda = 20$. Zaradi majhnosti populacije je bila računska zahtevnost vseh povzporejanih korakov algoritma v sklopu teh meritev veliko nižja kot pri velikosti populacije $\lambda = 250$, zaradi česar so se ti slabše povzporejali. Pri vseh dimenzijah problema smo najboljše rezultate dosegli z uporabo pristopa z večnitnostjo. Uporaba pristopa OpenCL se nam pri majhnih dimenzijah problema zaradi nizke računske zahtevnosti korakov in relativno majhnega števila uporabljenih delovnih enot ni obrestovala. S povečevanjem dimenzije problema pa postaja njegova uporaba vse bolj učinkovita, saj z njenim povečevanjem povečujemo tudi število uporabljenih delovnih enot, kar pripomore k vse boljši izkoriščenosti grafične kartice. Pri največjih dimenzijah problema smo najslabše čase izvajanja izmerili pri uporabi obeh pristopov MPI, kar lahko tudi to pot pripišemo visokim stroškom komunikacije med procesi MPI.

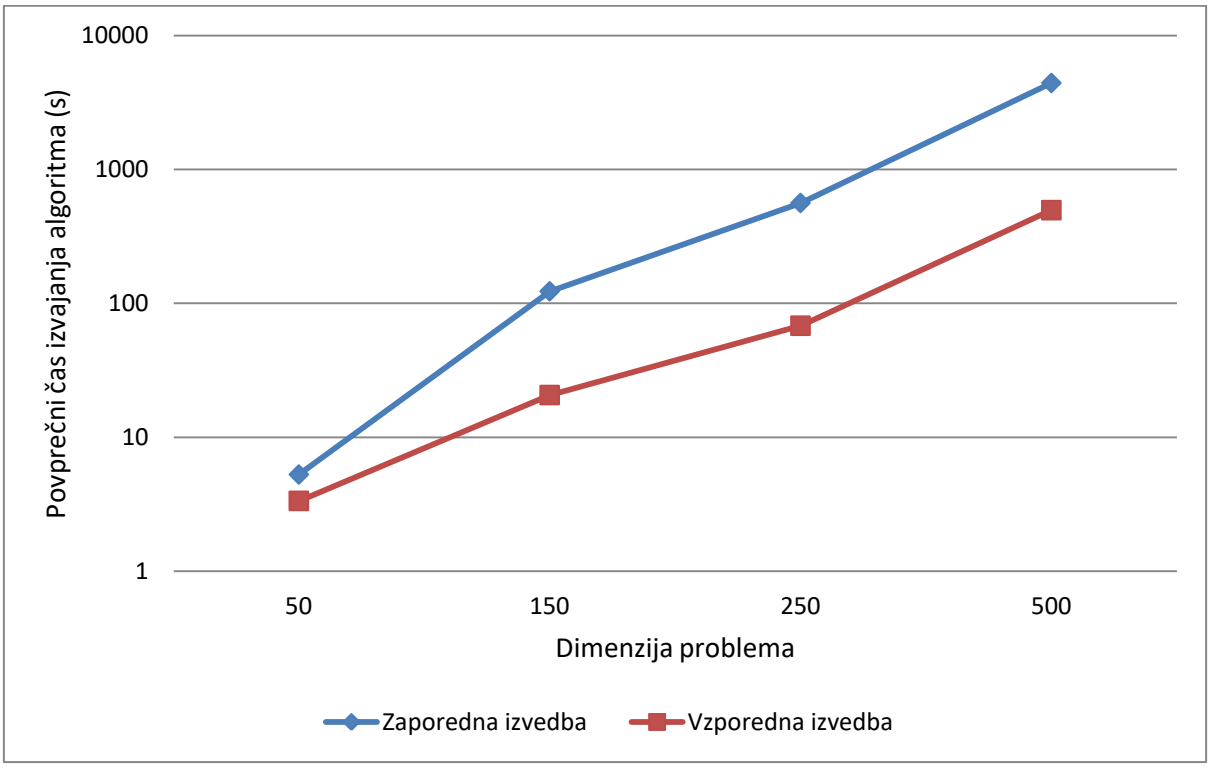
Iz pridobljenih rezultatov je razvidno, da je učinkovitost uporabe posameznih vzporednih pristopov odvisna predvsem od reševanih problemov. Tu še omenimo, da bi lahko na drugem gostitelju dobili povsem drugačne rezultate, saj je učinkovitost posameznih vzporednih pristopov v veliki meri odvisna še od strojne opreme gostitelja, kar smo tudi prikazali v prejšnjih podpoglavjih.

5.2 Primerjava zaporedne in vzporedne implementacije algoritma

V tem podpoglavju smo prikazali še primerjavo časa izvajanja zaporedne in vzporedne implementacije algoritma na našem gostitelju. Zanimal nas je predvsem vpliv dimenzije problema in velikosti populacije na dosežen faktor pohitritve algoritma (5.2).

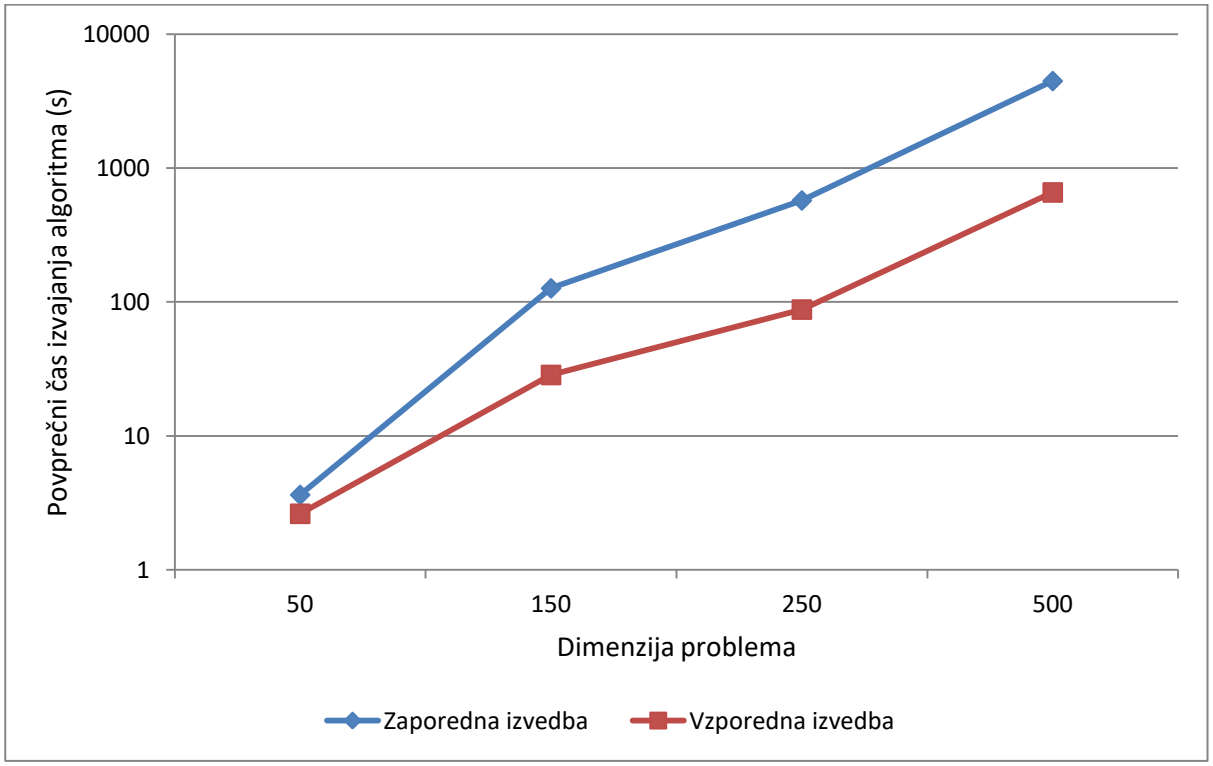
$$S = \frac{\check{c}as\ izvajanja\ zaporedne\ implementacije\ algoritma}{\check{c}as\ izvajanja\ vzporedne\ implementacije\ algoritma} \tag{5.2}$$

Meritve smo tudi to pot opravljali pri velikostih populacije $\lambda = 20$, $\lambda = 100$ in $\lambda = 250$ in pri dimenzijah problema $N = 50$, $N = 150$, $N = 250$ in $N = 500$. Pri vzporednemu izvajanju posameznih povzporejanih korakov algoritma smo uporabljali le najučinkovitejše pristope. Rezultate meritev smo prikazali na slikah 5.18, 5.19 in 5.20.



Slika 5.18: Graf primerjave časov izvajanja vzporedne in zaporedne implementacije algoritma pri velikosti populacije $\lambda = 250$.

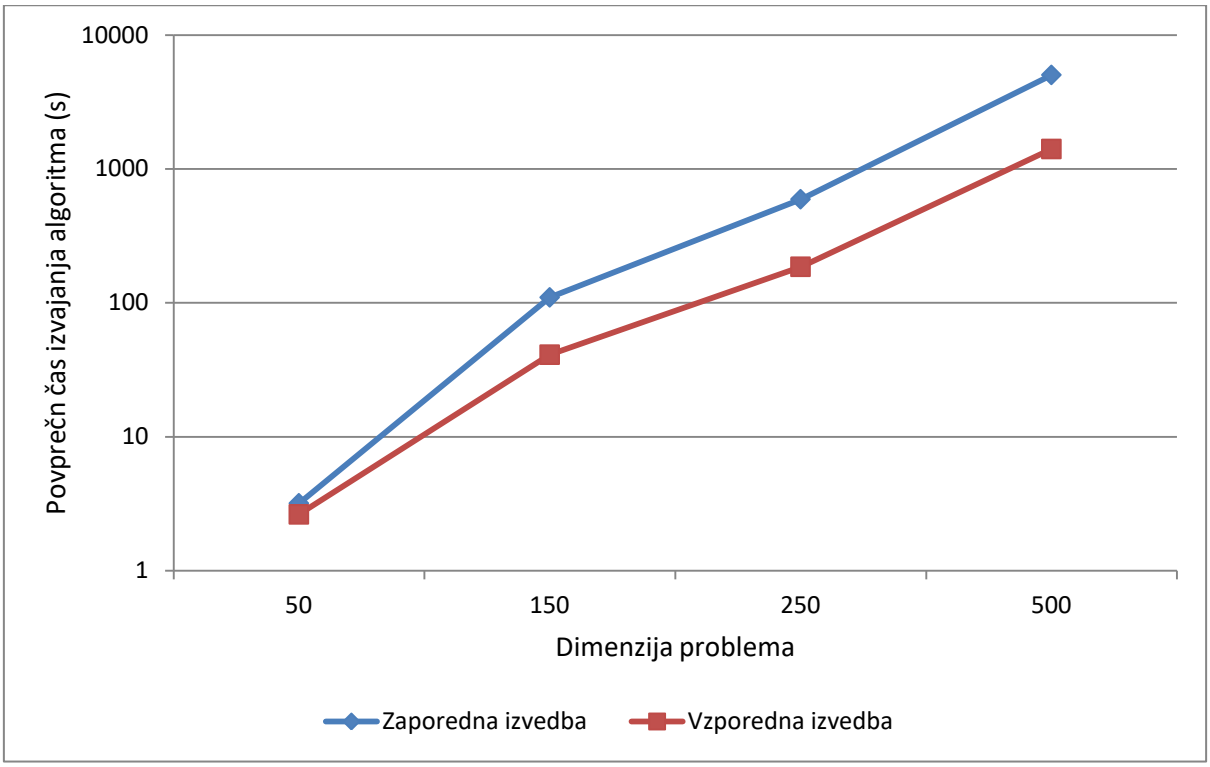
Prvi sklop meritev smo opravili pri velikosti populacije $\lambda = 250$. Pri vzporednemu izvajanju algoritma smo pri korakih generiranja populacije in izračuna kovariančne matrike uporabljali pristop OpenCL, pri koraku za ovrednotenje populacije pa pristop z večnitnostjo. Kot lahko vidimo iz slike 5.18, se nam vzporedno izvajanje algoritma splača že pri zelo majhnih dimenzijah problema. Pri največjih dimenzijah problema smo v skladu s pričakovanji izmerili najvišje faktorje pohitritve, saj se koraki algoritma zaradi visoke računske zahtevnosti pri takih dimenzijah problema najbolje povzporejajo. V sklopu naših meritev smo najvišji faktor pohitritve izmerili pri dimenziji problema $N = 500$, ta je znašal 8,90.



Slika 5.19: Graf primerjave časov izvajanja vzporedne in zaporedne implementacije algoritma pri velikosti populacije $\lambda = 100$.

Na grafu na sliki 5.19 smo prikazali rezultate meritev pri velikosti populacije $\lambda = 100$. Pri meritvah za dimenzijo problema $N = 50$ smo uporabljali izključno pristop z večnitnostjo, pri ostalih meritvah pa smo pri izvajanju korakov generiranja populacije in izračuna kovariančne matrike uporabljali pristop OpenCL, pri koraku ovrednotenja populacije pa pristop z večnitnostjo.

Faktorji pohitritve so bili v primerjavi z meritvami pri velikosti populacije λ = 250 nižji, kar lahko pripišemo manjši velikosti populacije, zaradi katere so bili koraki algoritma računsko manj zahtevni in so se slabše povzporejali. Naš najvišji doseženi faktor pohitritve, ki smo ga izmerili pri dimenziji problema $N = 500$ je znašal 6,78.



Slika 5.20: Primerjava časov izvajanja vzporedne in zaporedne implementacije algoritma pri velikosti populacije $\lambda = 20$.

Naš zadnji sklop meritev smo opravljali pri velikosti populacije λ = 20, rezultate teh smo prikazali na sliki 5.20. Pri vzporednemu izvajanju algoritma smo uporabljali izključno pristop z večnitnostjo, saj se je ta pri vseh dimenzijah problema izkazal za najučinkovitejšega. Faktorji pohitritve so bili izmed vseh naših meritev tu najnižji, saj je bila računska zahtevnost korakov zaradi zelo majhne velikosti populacije tudi relativno nizka. Naš najvišji doseženi faktor pohitritve, ki smo ga izmerili pri dimenziji problema $N = 500$ je znašal 3,57.

5.3 Splošne ugotovitve

Glede na pridobljene rezultate meritev smo spodaj podali nekaj splošnih ugotovitev:

- Pri splošni učinkovitosti izbranega vzporednega pristopa zelo veliko vlogo igrajo stroški komunikacije. Pristop z večnitnostjo je zaradi tega v splošnem zelo učinkovit, saj so stroški komunikacije pri njegovi uporabi zelo majhni, zaradi česar je pristop tudi zelo skalabilen. Poleg tega je povzporejanje algoritmov z uporabo večnitnosti v primerjavi z ostalimi vzporednimi pristopi relativno enostavno, a hkrati zelo učinkovito. Uporaba HTT se nam ponavadi splača, saj lahko z njeno uporabo dosežemo še nekoliko boljše čase izvajanja algoritmov.

- Skalabilnost pristopa MPI je pri računsko nezahtevnih problemih zelo slaba, saj so pohitritve izračunov pri takih problemih pogosto premajhne, da bi lahko odtehtale relativno visoke stroške komunikacije med procesi MPI. Zaradi tega se nam pri takih problemih pogosto najbolj splača le uporaba dveh procesov, saj so stroški komunikacije pri uporabi dveh procesov najnižji. Uporaba MPI je najbolj ustrezna pri reševanju računsko in časovno zelo zahtevnih problemov, saj so stroški komunikacije pri takih problemih skorajda zanemarljivi. V takih primerih lahko še dodatno izkoristimo ali niti CPE ali grafično kartico na vsakem izmed računalnikov v omrežju, podobno kot smo mi to storili pri vzporednemu pristopu MPI z večnitnostjo. S takim pristopom bi lahko pri takih problemih dosegli zelo visoke faktorje pohitritve.

- Uporaba OpenCL se nam pri časovno in/ali računsko nezahtevnih problemih ne obrestuje, saj pri takih problemih težko dobro izkoristimo grafično kartico, zaradi česar so pridobljene pohitritve v izračunih pogosto premajhne, da bi lahko odtehtale stroške komunikacije med gostiteljem in napravo. Njegovo uporabo lahko torej opravičimo le če uporabljamo veliko število delovnih enot in/ali če rešujemo računsko zahtevne probleme. Velik vpliv na učinkovitost uporabe OpenCL ima tudi izbira grafične kartice, saj je OpenCL s strani AMD veliko bolje podprt kot s strani NVIDIE.

- Pri povzporejanju časovno in računsko nezahtevnih korakov se nam najbolj splača uporaba pristopa z večnitnostjo, saj so stroški komunikacije pri njegovi uporabi najmanjši. Kljub temu se uporaba večnitnosti pri reševanju nekaterih nezahtevnih problemov vseeno odreže slabše kot zaporedna implementacija, ki je določenim problemom preprosto bolj pisana na kožo.

Poglavje 6 Sklepne ugotovitve

V okviru diplomske naloge smo algoritem CMA-ES uspešno pretvorili v modularno obliko in ga nato priredili za vzporedno izvajanje z uporabo treh različnih vzporednih pristopov. Z njihovo uporabo smo uspešno izboljšali čas izvajanja algoritma.

Z rezultati meritev smo v splošnem zadovoljni, saj so ti v skladu s pričakovanji, ki smo jih imeli ob pričetku izdelave diplomskega dela. V splošnem smo najbolj zadovoljni s pristopom večnitnosti, saj se je ta izkazal za zelo učinkovitega ne glede na reševani problem. Pri uporabi vzporednega pristopa OpenCL smo v skladu s pričakovanji pri računsko zahtevnih problemih izmerili najvišje faktorje pohitritve, pri vzporednemu pristopu MPI pa smo bili nekoliko razočarani nad nizko stopnjo skalabilnosti, čeprav smo kljub temu z njegovo uporabo uspešno izboljšali čas izvajanja algoritma. Pri uporabi tako osnovnega pristopa MPI kot pristopa MPI z večnitnostjo nekoliko obžalujemo, da izvajanja programa zaradi težavnosti konfiguracije nismo testirali na omrežju medsebojno povezanih računalnikov, saj bi tako lahko dobili boljšo idejo o dejanski učinkovitosti pristopa.

Prostora za izboljšave je še veliko. Posamezne vzporedne implementacije korakov bi lahko še dodatno priredili za bolj optimalno izvajanje. Pri programiranju grafične kartice bi lahko poleg vmesnika OpenCL uporabili tudi platformo CUDA, saj bi lahko z njeno uporabo korake algoritma priredili za učinkovitejše izvajanje na grafičnih karticah NVIDIA. Kljub vsemu pa menimo, da lahko ugotovitve pridobljene tekom izdelave diplomskega dela služijo vsaj kot osnova za povzporejanje ostalih evolucijskih algoritmov, saj so si ti v osnovi med seboj zelo podobni.

Literatura

[1] Evolutionary algorithm. Dostopno na: https://en.wikipedia.org/wiki/Evolutionary_algorithm.[Dostopano: 2017].

[2] Heuristic and meta-heuristic algorithms. Dostopno na: http://www.spatialanalysisonline.com/HTML/index.html?heuristic_and_meta-heuristic_a.htm. [Dostopano: 2017].

[3] Optimization problems. Dostopno na: https://en.wikipedia.org/wiki/Optimization_problem. [Dostopano: 2017].

[4] Test functions for optimization. Dostopno na: https://en.wikipedia.org/wiki/Test_functions_for_optimization. [Dostopano: 2017].

[5] CMA-ES algorithm. Dostopno na: https://www.lri.fr/~hansen/cmaesintro.html. [Dostopano: 2017].

[6] Stochastic optimization. Dostopno na: http://mathworld.wolfram.com/StochasticOptimization.html. [Dostopano: 2017].

[7] Rosenbrock function. Dostopno na: https://www.sfu.ca/~ssurjano/rosen.html. [Dostopano: 2017].

[8] Multi-core processor. Dostopno na: https://en.wikipedia.org/wiki/Multi-core_processor. [Dostopano: 2017].

[9] Message Passing Interface. Dostopno na: https://en.wikipedia.org/wiki/Message_Passing_Interface. [Dostopano: 2017].

[10] OpenCL. Dostopno na:
https://en.wikipedia.org/wiki/OpenCL. [Dostopano: 2017]

[11] MPJ Express. Dostopno na:
http://mpj-express.org/. [Dostopano: 2017].

[12] JOCL. Dostopno na:
http://www.jocl.org/. [Dostopano: 2017].

[13] Lehmer random number generator. Dostopno na:
https://en.wikipedia.org/wiki/Lehmer_random_number_generator.
[Dostopano: 2017].

[14] Box-Muller transform. Dostopno na:
https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform.
[Dostopano: 2017].

[15] Hyper-Threading. Dostopno na:
http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.
[Dostopano: 2017].