Paralelizacija nevronskih mrež

Kristian Žarn, Maks Vrščaj

Porazdeljeni sistemi 2016/17, Fakulteta za računalništvo in informatiko

27. januar 2017

Kazalo

1 Predstavitev problema

Za paralelizacijo sva si izbrala algoritem iz področja umetne inteligence, in sicer strojno učenje na podlagi nevronskih mrež. Bolj konkretno gre za večnivojski perceptron z gradientnim spustom. To je eden izmed mnogih algoritmov za nadzorovano učenje. Uporabljena literatura na to temo se nahaja na [3].

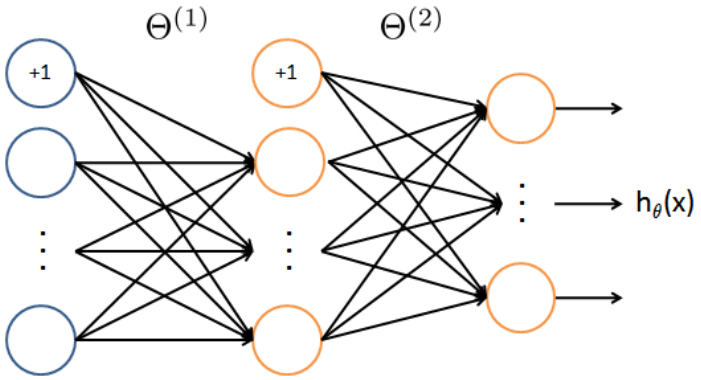
Problemska domena, na kateri sva preverila delovanje algoritma je razpoznavanje števk. Vhod v algoritem je slika števke, izhod pa ena izmed oznak 0 do 9. Gre torej za problem klasifikacije. S strojnim učenjem takšne probleme rešujemo na naslednji način. Pridobiti je potrebno bazo označenih primerov. V najnem primeru so to slike števk z pripadajočimi oznakami 0 do 9. Tu sva uporabila bazo MNIST, ki je dostopna na [2]. Množico primerov nato razdelimo na učni in testni del. Učno množico uporabimo za grajenje modela, s testno množico pa ocenimo kako dobra je napoved modela.

Najbolj časovno zahtevna naloga algoritma je seveda učenje, zato je to glavni izziv za paralelizacijo. Ena izmed idej za paralelizacijo je opisana v članku [4].

2 Delovanje Nevronskih mrež

2.1 Predikcija

Preprost primer nevronske mreže je prikazan na sliki 1. Celotna mreža je sestavljena iz posameznih perceptronov. Perceptron ima lahko mnogo vhodov in en izhod. Vsakemu vhodu pripada utež, naloga perceptrona pa je, da poračuna uteženo vsoto svojih vhodov in aktivira svoj izhod, če je vsota večja od mejne vrednosti.



Slika 1: Primer nevronske z enim skritim nivojem.

Naučeni model je predstavljen z utežmi za povezavah. Sama predikcija je zato preprosta. Na vhod postavimo nove vrednosti, nato pa poračunamo utežene vsote po vseh nivojih, na izhodu pa se pojavi rezultat klasifikacije (izhod ima seveda lahko več nevronov, v primeru klasifikacije v več razredov). Temu postopku pravimo propagacija naprej (angl. forward propagation).

2.2 Učenje

Rezultat učenja so uteži na povezavah. Z njimi skušamo minimizirati cenilno funkcijo, ki je definirana na podlagi razlike med napovedjo in resnično oznako. Za minimizacijo funkcije lahko uporabimo različne nelinearne optimizacijske postopke. V tej nalogi sva se odločila za gradientni spust, zaradi njegove preproste implementacije. Gre za iterativni postopek, opisuje pa ga spodnja psevdokoda 1.

Glavni problem je računanje parcialnih odvodov cenilne funkcije, ki jih optimizacijski postopki potrebujejo za delovanje. To nalogo opravlja algoritem imenovan propagacija nazaj (angl. backpropagation). Algoritem opisuje spodnja psevdokoda 2.

Algoritem 1 Gradient Descent

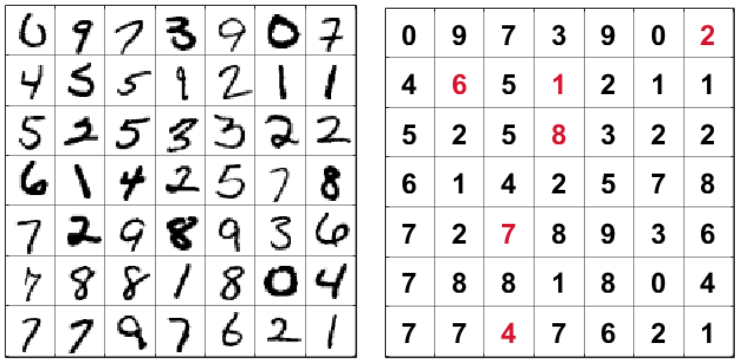
Vhod: Vektor zacetnih parametrov Θ; število iteracij N
Izhod: Cena J, novi parametri Θ' ki optimizirajo cenilno funkcijo
1: Določi stopnjo učenja α (angl. learning rate)
2: **for** iter = 1 do N **do**
3: Izracunaj vektor parcialnih odvodov $\Delta := Backpropagation(\Theta, X, Y)$
4: Posodobi parametre $\Theta := \Theta - \alpha\Delta$
5: **end for**
6: $\Theta' := \Theta$

Algoritem 2 Backpropagation

Vhod: Uteži po nivojih Θ^l; učna množica primerov X; oznake Y
Izhod: Gradient uteži po nivojih Δ^l
1: Inicializacija $\Delta^l = 0$
2: **for** i = 1 do length(X) **do**
3: Propagacija naprej z vhodom $X(i)$, da dobimo aktivacije nevronov po nivojih a^l
4: Izracunaj razliko na zadnjem nivoju, $\delta^L := a^L - Y(i)$
5: Izracunaj razlike na preostalih nivojih, $\delta^{L-1}, \delta^{L-2}, \ldots, \delta^2$ na podlagi pripadajočega a^l in Θ^l
6: Posodobi parcialne odvode po nivojih, $\Delta^l := \Delta^l + \delta^{l+1}a^l$
7: **end for**

2.3 Primer delovanja

Na sliki 2 je prikazano delovanje programa na manjšem številu primerov iz baze. Napačno klasificirane številke so označene z rdečo in jih je relativno malo.



0	9	7	3	9	0	2
4	6	5	1	2	1	1
5	2	5	8	3	2	2
6	1	4	2	5	7	8
7	2	7	8	9	3	6
7	8	8	1	8	0	4
7	7	4	7	6	2	1

(a) (b)

Slika 2: Na levi strani je prikazanih nekaj primerov števil iz baze MNIST, na desni strani pa je rezultat njihove klasifikacije. Napačne predikcije so označene z rdečo.

3 Serijski algoritem

3.1 Implementacija

Najin model nevronske mreže je prikazan na sliki 1. Sestavljen je iz vhodnega, skritega in izhodnega nivoja. V bazi MNIST so slike resolucije $28x28$. Vsak piksel predstavlja en vhod, zato vhodni nivo vsebuje 784 enot. Na skritem nivoju sva spreminjala število enot od 5 do 50. Izhodni nivo ima deset izhodnih enot, ki nam določajo deset razredov, v katere klasificiramo.

3.2 Meritve

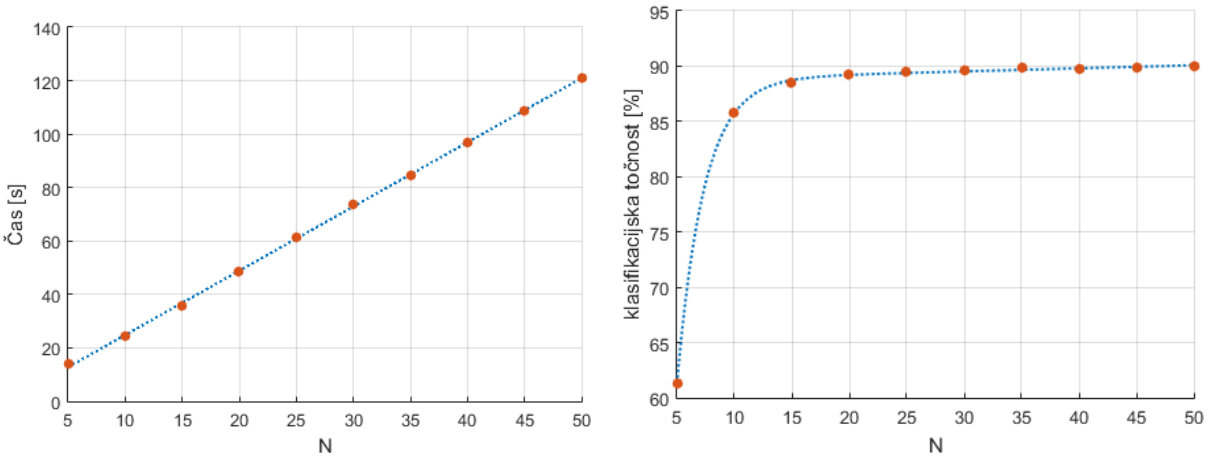
Pri merjenju hitrosti učenja nevronske mreže sva učno množico pustila nespremenjeno. Meritve so pokazale, da večanje učne množice v tem primeru ne prispeva veliko k točnosti, zato sva se omejila na 10000 primerov. Vhod je torej matrika velikosti $10000x784$, kjer vsaka vrstica predstavlja posamezno sliko številke.

Čas izvajanja sva merila v odvisnosti od števila nevronov N na skritem nivoju. To število sva spreminjala od 5 do 50 s korakom 5. Za vsak N sva pognala učenje 15-krat. Število iteracij gradientnega spusta sva nastavila na 150 z regularizacijskim faktorjem $\lambda = 0.1$. Poleg časa izvajanja sva izmerila tudi klasifikacijsko točnost na testni množici, ki je sestavljena iz 10000 primerkov.

Meritve sva izvajala na prenosniku s štirijedrnim procesorjem (2.0Ghz - 2.9GHz) in 4GB RAM-a. Rezultati so prikazani v tabeli 1. Grafa za hitrost izvajanja in klasifikacijske točnosti v odvisnosti od števila skritih vozlišč sta prikazana na sliki 3.

N	Čas [s]	Čas SE [s]	Točnost [%]	Točnost SE [%]
5	13,94	0,06	61,32	1,89
10	24,37	0,03	85,81	0,38
15	35,60	0,05	88,46	0,08
20	48,65	0,21	89,21	0,04
25	61,30	0,24	89,48	0,03
30	73,56	0,45	89,67	0,04
35	84,66	0,21	89,81	0,04
40	96,95	0,18	89,80	0,04
45	108,69	0,16	89,89	0,04
50	121,19	0,43	89,98	0,04

Tabela 1: Tabela prikazuje čase izvajanja in točnost klasifikacije, pri povečevanju števila nevronov na skritem nivoju.



(a) (b)

Slika 3: (a): Čas izvajanja učenja v odvisnosti od števila vozlišč na skritem nivoju. (b): Klasifikacijska točnost v odvisnosti od števila vozlišč na skritem nivoju.

3.3 Analiza meritev

Rezultati meritev so pokazali, da se čas izvajanja učeja v odvisnosti od števila vozlišč na skritem nivoju povečuje linearno 3a, to je v skladu s teoretično časovno zahtevnostjo $\mathcal{O}(n)$.

Teoretična časovna zahtevnost učenja je linearna, če spreminjamo samo velikost skritega nivoja, pri konstantni velikosti učne množice ter vhodnega in izhodnega nivoja. Vse računske operacije z vrednostmi, ki pripadajo nevronom so namreč neposredno odvisne samo med sosednjima nivojema. Ker se tu spreminja samo velikost skritega nivoja, lahko sklepamo, da število operacij raste linearno, torej je časovna zahtevnost linearna.

Slika 3b prikazuje vpliv skritih nevronov na klasifikacijsko točnost. Klasifikacijska točnost se pri povečevanju števila vozlišč na skritem nivoju asimptotično približuje vrednosti okrog 90%. Opazimo lahko, da so pridobitve pri 30 in več nevronih minimalne. Najboljša dosežena točnost pri testiranju je bila 89.98% pri 50 vozliščih. Tukaj je največja omejitev predvsem gradientni spust, ki v tej preprosti obliki konvergira zelo počasi.

4 Pthreads

4.1 Paralelizacija

Najbolj časovno zahteven del učenja je posodabljanje gradienta. Vsi učni primeri morajo namreč skozi algoritem Backpropagation, da se poračuna njihov prispevek h gradientu. Algoritem sva paralelizirala tako, da vsaki niti pripiševa enak del učne množice. Nato vsaka nit poračuna gradiente parametrov za svoj del množice. Glavna nit čaka, da ostale zaključijo z delom, nato pa sešteje delne rezultate v končni rezultat. Ta del ustreza paralelizaciji zanke v algoritmu 2. Paralelizirala sva tudi posodabljanje parametrov znotraj gradientnega spusta. Vsaka nit dobi svoj del vektorja, kateremu prišteje gradient. Ta del ustreza posodobitvi parametrov znotraj for zanke v algoritmu 1. Niti med seboj ne potrebujejo posebne komunikacije, saj lahko svoje delo opravijo neodvisno. Glavna nit je edina, ki mora čakati, da se pomožne končajo.

4.2 Meritve

Testni sistem je enak kot v prvem delu, torej prenosnik s štirijedrnim procesorjem (2.0Ghz - 2.9GHz) in 4GB RAM-a. Procesor ima omogočen HyperThreading, torej lahko vsako jedro upravlja z dvema nitma. Časovne meritve so podane v tabeli 2 in grafično na sliki 4

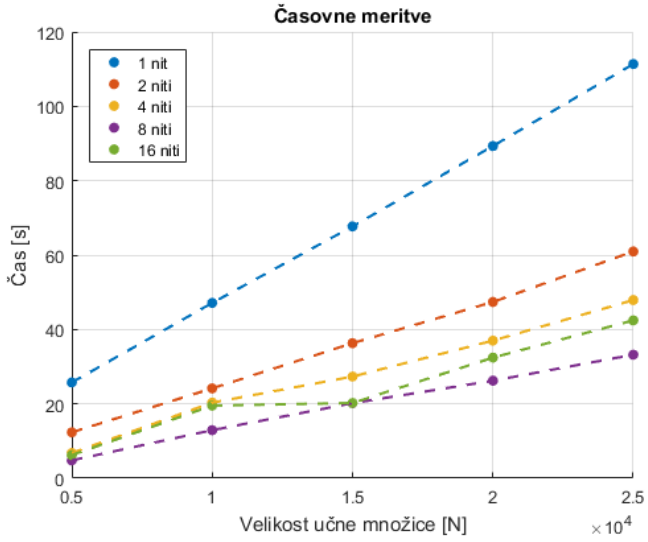
Velikost problema predstavlja število primerkov N v učni množici. Spreminjala sva jo od 5000 do 25000 s korakom po 5000. Število skritih nevronov sva fiksirala na 25, število iteracij gradientnega spusta pa na 100. Za število niti T, pa sva izbrala vrednosti 1, 2, 4, 8 in 16. Za vsako kombinacijo sva izvedla 10 poskusov.

P \ N	5k	10k	15k	20k	25k
1	25,76 ± 0,36	47,13 ± 0,36	67,71 ± 0,46	89,32 ± 0,74	111,37 ± 0,56
2	12,35 ± 0,06	24,16 ± 0,32	36,31 ± 0,71	47,40 ± 0,90	60,98 ± 1,37
4	6,78 ± 0,05	20,32 ± 2,27	27,34 ± 2,91	37,00 ± 1,02	47,91 ± 0,86
8	4,80 ± 0,03	12,92 ± 0,49	20,19 ± 0,29	26,22 ± 0,25	33,21 ± 0,05
16	6,21 ± 0,66	19,52 ± 0,02	20,23 ± 2,33	32,43 ± 3,05	42,44 ± 3,88

Tabela 2: Časovne meritve učenja, pri različnih velikostih problema N in številu niti T.

4.3 Analiza meritev

Iz grafa meritev na sliki 4 lahko opazimo precejšnje zmanjšanje časa izvajanja, kar potrjujeta tudi pohitritev in učinkovitost, ki sta prikazani v tabeli 3 in grafu 5. Iz slike 5a lahko opazimo, da pohitritev raste skoraj linearno, dokler število niti ne preseže število jeder. Pohitritev je največja $(5, 3)$ pri 8 nitih, pri večjem številu niti, pa se zmanjša ali pa ostane nespremenjena. To je seveda smiselno, saj lahko procesor upravlja z največ 8 niti.



Slika 4: Grafični prikaz časovnih meritev v odvisnosti od velikosti problema. Prikazani so grafi pri različnem številu niti.

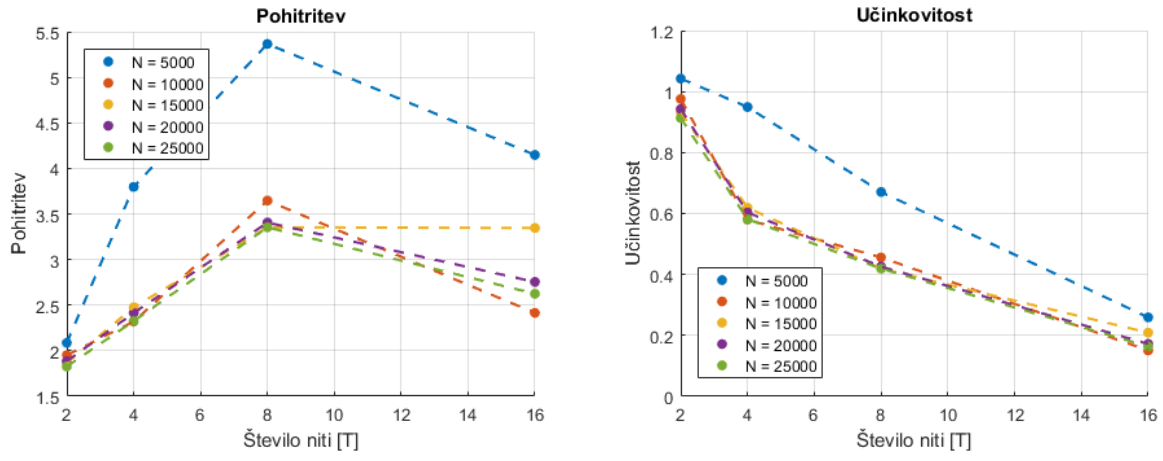
T \ N	5k	10k	15k	20k	25k
2	2,0	1,9	1,8	1,8	1,8
4	3,7	2,3	2,4	2,4	2,3
8	5,3	3,6	3,3	3,4	3,3
16	4,1	2,4	3,3	2,7	2,6

(a) pohitritev

T \ N	5k	10k	15k	20k	25k
2	1,04	0,97	0,93	0,94	0,91
4	0,94	0,57	0,61	0,60	0,58
8	0,67	0,45	0,41	0,42	0,41
16	0,25	0,15	0,20	0,17	0,16

(b) učinkovitost

Tabela 3: Tabelarični prikaz pohitritve in učinkovitosti. N je velikost problema, T pa število niti.



(a) (b)

Slika 5: (a): Pohitritev algoritma glede na število niti. (b): Učinkovitost niti glede na njihovo število. V obeh primerih so narisani grafi pri različnih velikostih vhoda N.

Učinkovitost nam pove kakšen delež procesorske moči je bil uporabljen za reševanje problema. Pričakovano je da z večanjem števila niti pada, zaradi režije. To se pokaže tudi v najnem primeru. V najnem primeru je pri največji pohitritvi, učinkovitost 67%, za večje množice pa malo nad 40%.

5 OpenMP

5.1 Paralelizacija

OpenMP je programski vmesnik, ki v programu omogoči večnitenje. Njegova naloga je, podobno kot pri Pthreads, da delo razdeli med več niti. Ideje za paralelizacijo kode so zato med tema knjižnicama podobne.

Učenje sva paralelizirala na naslednji način. Najprej glavna nit za vse preostale rezervira prostor, kjer bodo niti poračunale prispevke h gradientu. Vsaki niti je nato dodeljen enak delež učnih primerov. Posamezna nit poračuna gradiente za svoj del množice. Ko so vsi delni rezultati poračunani, jih niti paralelno seštejejo v končni rezultat. Paralelizirala sva tudi posodabljanje parametrov in regularizacijo gradienta ter cenilne funkcije.

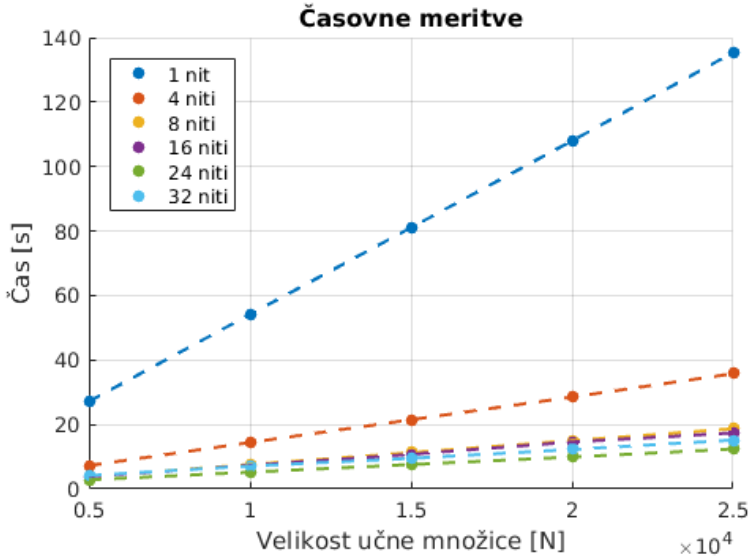
5.2 Meritve

Za testiranje sva uporabila strežnik na fakulteti, ki je opremljen z dvema procesorjema Intel Xeon E5-2620 in 64GB RAM-a. Skupno je na voljo 12 jeder in 24 niti.

Pri testiranju sva uporabila fiksno število skritih nevronov (25) in iteracij gradientnega spusta (100). Spreminjala sva velikost učne množice N, in sicer od 5000 do 25000 s korakom po 5000. Za število niti T, pa sva izbrala vrednosti 1, 4, 8, 16, 24 in 32. Za vsako kombinacijo parametrov je bilo izvedenih 10 poskusov. Časovne meritve so zbrane v tabeli 4 in grafično prikazane na sliki 6.

T \ N	5000	10000	15000	20000	25000
1	27,143 ± 0,003	54,143 ± 0,007	81,064 ± 0,018	107,983 ± 0,028	135,349 ± 0,013
4	7,240 ± 0,004	14,399 ± 0,006	21,476 ± 0,006	28,532 ± 0,007	35,754 ± 0,011
8	3,849 ± 0,002	7,567 ± 0,002	11,288 ± 0,007	14,995 ± 0,005	18,690 ± 0,005
16	3,872 ± 0,016	7,319 ± 0,028	10,748 ± 0,063	14,575 ± 0,034	17,373 ± 0,034
24	2,836 ± 0,005	5,212 ± 0,007	7,576 ± 0,006	9,905 ± 0,008	12,387 ± 0,012
32	4,161 ± 0,009	7,127 ± 0,014	9,503 ± 0,018	12,268 ± 0,024	15,144 ± 0,019

Tabela 4: Časovne meritve učenja, pri različnih velikostih učne množice N in številu niti T.



Slika 6: Prikaz časovnih meritev v odvisnosti velikosti učne množice. Prikazani so grafi pri različnem številu niti.

5.3 Analiza meritev

Iz slike 6 je razvidno občutno zmanjšanje časa izvajanja, že pri štirih nitih. Tudi pri osmih je čas izvajanja opazno krajši. Časi za 16, 24 in 32 niti pa so zelo skupaj, najboljši rezultat pa je bil dosežen z 24-imi niti. Pohitritve in učinkovitosti pozametnih testov so zbrane v tabeli 5 in prikazane grafično na sliki 7.

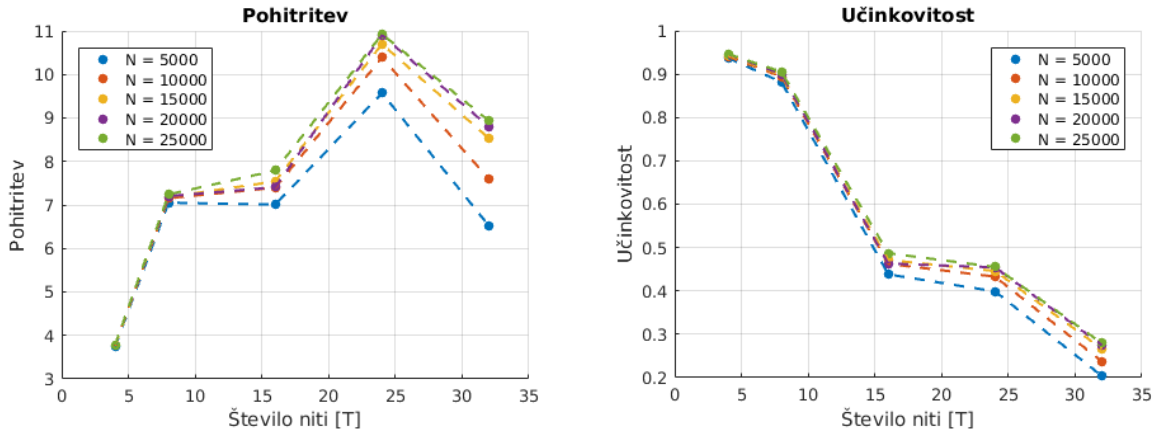
T \ N	5k	10k	15k	20k	25k
4	3,7	3,8	3,8	3,8	3,8
8	7,1	7,2	7,2	7,2	7,2
16	7,0	7,4	7,5	7,4	7,8
24	9,6	10,4	10,7	10,9	10,9
32	6,5	7,6	8,5	8,8	8,9

(a) pohitritev

T \ N	5k	10k	15k	20k	25k
4	0,94	0,94	0,94	0,95	0,95
8	0,88	0,89	0,90	0,90	0,91
16	0,44	0,46	0,47	0,46	0,49
24	0,40	0,43	0,45	0,45	0,46
32	0,20	0,24	0,27	0,28	0,28

(b) učinkovitost

Tabela 5: Tabelarični prikaz pohitritve in učinkovitosti. N je velikost učne množice, T pa število niti.



(a) (b)

Slika 7: (a): Pohitritev algoritma glede na število niti. (b): Učinkovitost niti glede na njihovo število. V obeh primerih so narisani grafi pri različnih velikostih učne množice N.

Pri vseh velikostih učne množice dobimo dobre pohitritve za 4 in 8 niti. Zanimivo je, da se pri 16-ih nitih pohitritev skoraj nič ne poveča v primerjavi z 8-imi. Točnega razloga za to nisva odkrila, ugibamo pa lahko, da niti niso bile dobro razporejene med jedra procesorja, ali pa OpenMP ni generiral maksimalno dovoljeno število niti. Največjo pohitritev sva dosegla z 24-imi nitmi in znaša 10.9, pri večjem številu pa pohitritev začne padati, ker je niti preveč, da bi sočasno tekle na procesorju.

Iz grafa na sliki 7a lahko opazimo še eno zanimivost. Pohitritve namreč rastejo z večanjem učne množice (krivulje so vedno višje). S tem ko povečamo velikost učne množice, povečamo tudi delo ki ga dobi posamezna nit. S tem se poveča tudi razmerje med koristnim delom in režijo med nitmi. Večanje učne množice torej koristno vpliva na pohitritev.

Učinkovitost, je prikazana na sliki 7b, pove pa nam kakšen delež procesorske moči je bil koristno uporabljen. Zelo dobra učinkovitost, preko 90%, je dosežena pri 4-ih in 8-ih nitih, nato pa začne hitro padati. Pri največji pohitritvi znaša učinkovitost 46%. Glede na to, da v tem primeru teče 24 niti, strežnik pa ima 12 pravih jeder, je rezultat soliden.

5.4 Primerjava z Pthreads

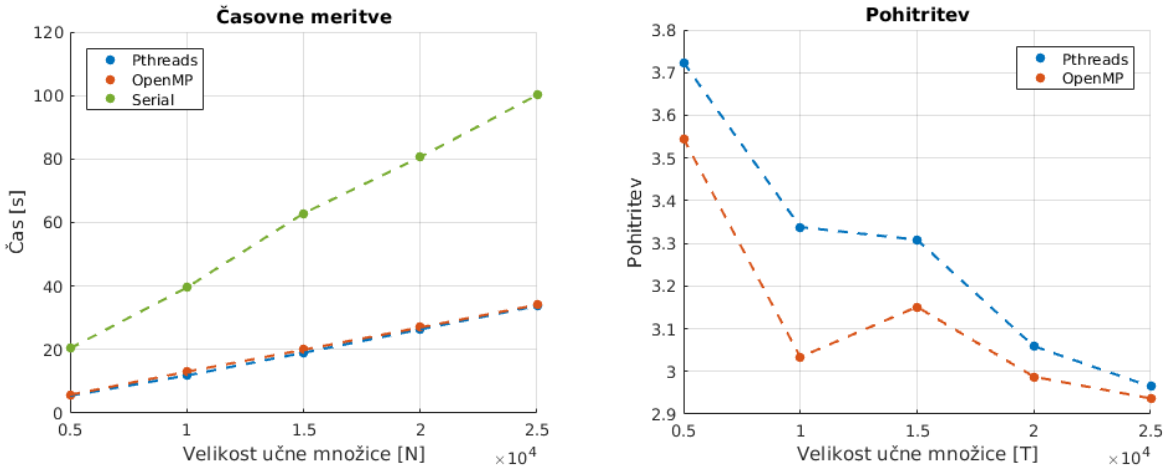
Poglejmo si še kako se OpenMP obnese v primerjavi z Pthreads. Tukaj sva meritve izvajala na prenosniku s štirijedrnim procesorjem (2.0Ghz - 2.9GHz) z omogočenim HyperThreading-om, ter 4GB RAM-a. Pri testiranju sva uporabila enake parametre, kot so opisani v poglavju 5.2. Število niti sva nastavila na 8, ker je to maksimalno število niti s katerimi lahko naenkrat upravlja procesor.

N	5000	10000	15000	20000	25000
Serial	20,389 ± 0,121	39,527 ± 0,082	62,887 ± 0,809	80,625 ± 0,802	100,134 ± 0,148
Pthreads	5,475 ± 0,181	11,841 ± 0,435	19,008 ± 0,416	26,357 ± 0,597	33,777 ± 0,373
OpenMP	5,751 ± 0,064	13,029 ± 0,446	19,956 ± 0,500	26,987 ± 0,619	34,097 ± 0,487

Tabela 6: Primerjava časovnih meritev za serijski program, Pthreads in OpenMP.

N	5000	10000	15000	20000	25000
Pthreads	3,7	3,3	3,3	3,1	3,0
OpenMP	3,5	3,0	3,2	3,0	2,9

Tabela 7: Primerjava pohitritev za Pthreads in OpenMP.



(a) (b)

Slika 8: (a): Časovne meritve za serijski program, Pthreads in OpenMP. (b): Pohitritev za Pthreads in OpenMP v odvisnosti od velikosti učne množice.

Rezultati časovnih meritev so prikazani v tabeli 6 in na sliki 8a, pohitritve pa v tabeli 7 in na sliki 8b. Pthreads implementacija je nekoliko hitrejša, vendar pa so razlike v hitrosti med obema knjižnicama minimalne. Tudi iz tabele pohitritev je razvidno, da se razlika pri pohitritvi giblje nekje med 0.1 in 0.3.

Iz grafa na sliki 8b lahko tokrat opazimo, da pohitritev z večanjem učne množice pada. To je ravno v nasprotju z rezultati v poglavju 5.3, kjer je pohitritev z večanjem učne množice naraščala. Razlog za to je zelo verjetno procesor v prenosniku, ki lahko prilagaja svojo frekvenco delovanja glede na temeraturo. Za majhne učne množice, je učenje krajše, zato se procesor manj segreje in lahko deluje pri večji frekvenci, pri večjih pa je ravno obratno.

5.5 XeonPhi

Učenje nevronske mreže sva testirala tudi na koprocesorju Xeon Phi. Vse parametre učenja sva fiksirala. Število skritih nevronov na 25, iteracij gradientnega spusta na 100, velikost učne množice na 10000. Spreminjala sva število niti T, kjer sva vrednosti nastavila na 1, 20, 40, 60, 80, 120, 236.

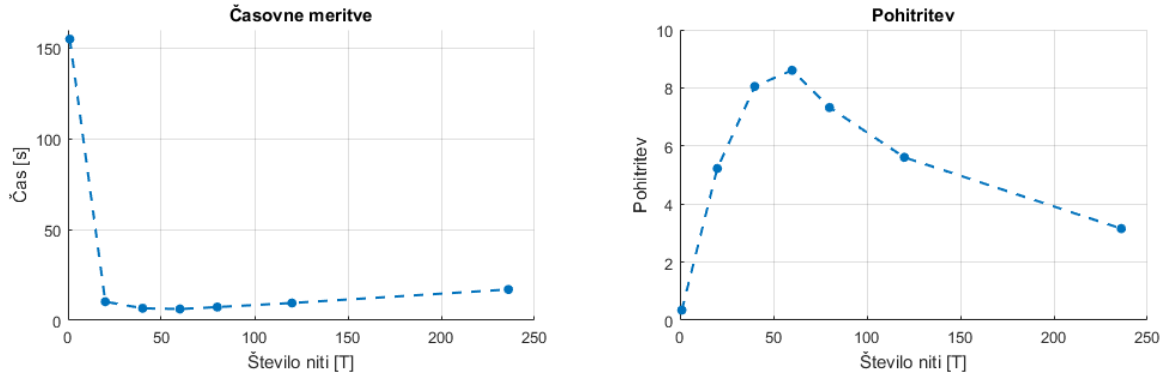
Rezultati časovnih meritev so prikazani v tabeli 8, pohitritev glede na serijski program, pa v tabeli 9. Grafični prikaz je na sliki 9.

1	20	40	60	80	120	236
154,80 ± 0,97	10,26 ± 0,10	6,66 ± 0,03	6,23 ± 0,03	7,32 ± 0,09	9,56 ± 0,53	17,00 ± 0,16

Tabela 8: Časovne meritve učenja, pri različnem številu niti T.

1	20	40	60	80	120	236
0,35	5,23	8,05	8,61	7,32	5,61	3,15

Tabela 9: Pohitritev, pri različnem številu niti T.



(a) (b)

Slika 9: (a): Časovne meritve za XeonPhi (b): Pohitritev za XeonPhi v odvisnosti od števila niti.

Največjo pohitritev dobimo pri 60-ih nitih, ta znaša 8.6, nato pa pohitritev začne padati, saj niti niso več dobro izkoriščene. Če primerjamo najboljša časa izvajanja s procesorjem, lahko opazimo, da je na koprocesorju čas izvajanja skoraj sekundo slabši (5.21s na procesorju in 6.23 na koprocesorju).

6 OpenCL

6.1 Paralelizacija

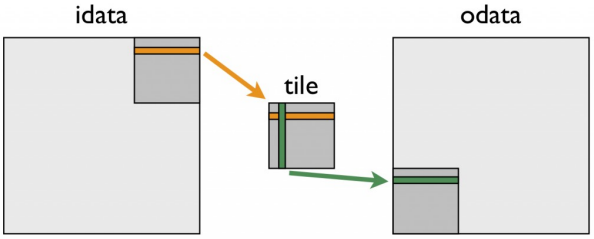
OpenCl je programski vmesnik, ki omogoča programiranje različnih heterogenih platform. V tej nalogi smo ga uporabili za programiranje grafičnih kartic.

Za razliko od Pthreads in OpenMP so tu bile potrebne večje spremembe v kodi. Algoritem učenja sva najprej preoblikovala tako, da sva se znebila eksplicitne zanke ki se sprehodi čez učne primere in obdela vsakega posebej. To zanko sva nadomestila z matričnimi operacijami kot so množenje, transponiranje, odštevanje, itd. Omenjene operacije so namreč lahko na grafičnih karticah realizirane zelo učinkovito.

Množenje matrik sva implementirala z uporabo lokalnega pomnilnika. Množenje poteka po blokih velikosti $16x16$. Niti znotraj bloka sodelujejo tako, da najprej prenesejo ustrezne podatke iz glavnega v lokalni pomnilnik. V naslednjem koraku poračunajo svojo delno vsoto in nato postopek ponavljajo dokler ne obdelajo vseh blokov. Pomembno je še da alociramo lokalni pomnilnik z enim odvečnim stolpcem (torej

16x17). S tem se izognemo konfliktom pri prenosih iz lokalnega pomnilnika (angl. bank conflict). Ta preprosta izboljšava je pohitrila celotno izvajanje algoritma za več kot dvakrat.

Z uporabo lokalnega pomnilnika sva implementirala tudi transponiranje matrik. Pri naivnem transponiranju beremo zaporedne lokacije v pomnilniku, pišemo pa v nezvezne lokacije, kar močno upočasni delovanje. Ponovno si lahko pomagamo z bloki. Posamezen blok najprej preberemo v lokalni pomnilnik, nato pa stolpce v lokalnem pomnilniku zapišemo v vrstice globalnega pomnilnika. S tem smo transponirali blok in pisali v zaporedne lokacije v pomnilniku. Postopek je prikazan na sliki 10.



Slika 10: Transponiranje matrike z uporabo lokalnega pomnilnika.

Preostale operacije, delujejo neodvisno nad elementi vektorjev, zato je njihova implementacija enostavna. Te operacije so funkcija Sigmoid in njen odvod, odštevanje, množenje po elementih, regularizacija gradienta in posodabljanje parametrov.

Celotni program deluje tako, da gostitelj najprej prenese vse vhodne podatke na grafično kartico in pripravi ščepce, ki se bodo izvajali. Nato v zanki, ki opravlja nalogo gradientnega spusta, gostitelj kliče pripravljene ščepce v ustreznem vrstnem redu. Na koncu se poračunani parametri prenesejo nazaj na gostitelja.

6.2 Meritve

Za testiranje sva uporabila strežnik na fakulteti. Opremljen je z dvema procesorjema Intel Xeon E5-2620 (skupno 12 jeder in 24 niti) in ima 64GB RAMA-a. Poleg tega je opremljen še z dvema karticama Tesla K20m.

Pri testiranju sva uporabila samo eno izmed kartic. Število skritih nevronov sva fiksirala na 25. Spreminjala sva velikost učne množice N, in sicer od 10000 do 60000 s korakom po 10000. Vsako velikost množice sva testirala z različnim številom iteracij gradientnega spusta I, in sicer od 50 do 200 s korakom po 50. Za vsako kombinacijo parametrov je bilo izvedenih 10 poskusov. Rezultati meritev so prikazani v tabeli 10 in prikazani grafično na sliki 13.

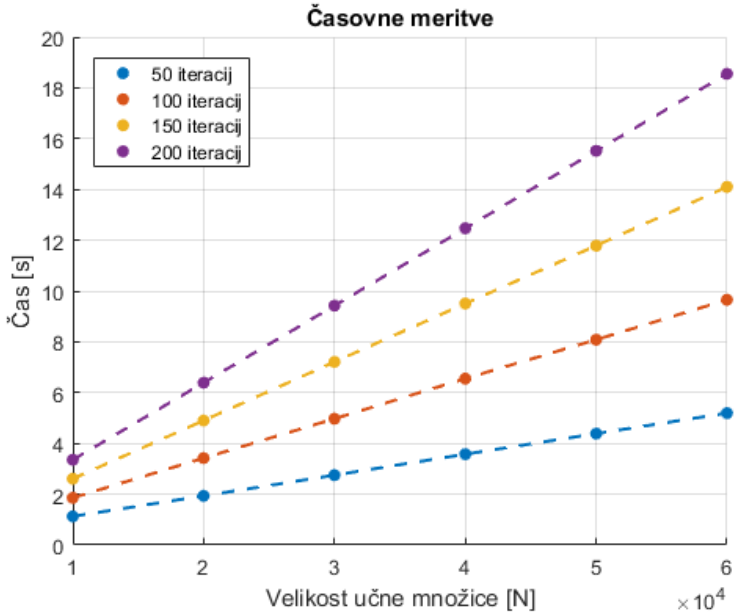
Da sva lahko izračunala pohitritve, sva za enake parametre na strežniku pognala tudi serijski algoritem. Te meritve so zbrane v tabeli 11.

I \ N	10k	20k	30k	40k	50k	60k
50	1,126 ± 0,014	1,936 ± 0,008	2,750 ± 0,008	3,575 ± 0,008	4,386 ± 0,009	5,192 ± 0,012
100	1,857 ± 0,005	3,422 ± 0,010	4,972 ± 0,007	6,552 ± 0,012	8,083 ± 0,006	9,653 ± 0,012
150	2,609 ± 0,005	4,901 ± 0,007	7,208 ± 0,008	9,505 ± 0,008	11,786 ± 0,009	14,100 ± 0,010
200	3,354 ± 0,007	6,390 ± 0,006	9,424 ± 0,009	12,467 ± 0,009	15,518 ± 0,006	18,552 ± 0,009

Tabela 10: Časovne meritve za OpenCL, za različne učne množice N in števila iteracij I.

I \ N	10k	20k	30k	40k	50k	60k
50	26,81 ± 0,01	53,65 ± 0,02	80,41 ± 0,07	107,54 ± 0,12	134,03 ± 0,06	160,81 ± 0,13
100	53,62 ± 0,02	107,16 ± 0,04	160,86 ± 0,12	217,43 ± 0,53	267,69 ± 0,18	321,23 ± 0,09
150	80,41 ± 0,02	160,74 ± 0,07	241,17 ± 0,32	321,55 ± 0,18	401,19 ± 0,16	482,36 ± 0,15
200	107,13 ± 0,04	213,80 ± 0,19	321,63 ± 0,15	427,86 ± 0,32	535,29 ± 0,18	641,85 ± 0,16

Tabela 11: Časovne meritve serijskega programa, za različne učne množice N in števila iteracij I.



Slika 11: Časovne meritve izvajanja na grafični kartici v odvisnosti od velikosti učne množice.

6.3 Analiza meritev

Iz slike 13 lahko opazimo da z večanjem velikosti množice in števila iteracij, čas izvajanja raste linearno, kar je tudi pričakovano. Bolj zanimive so pohitritve, ki so zbrane v tabeli 12 in prikazane na sliki 12.

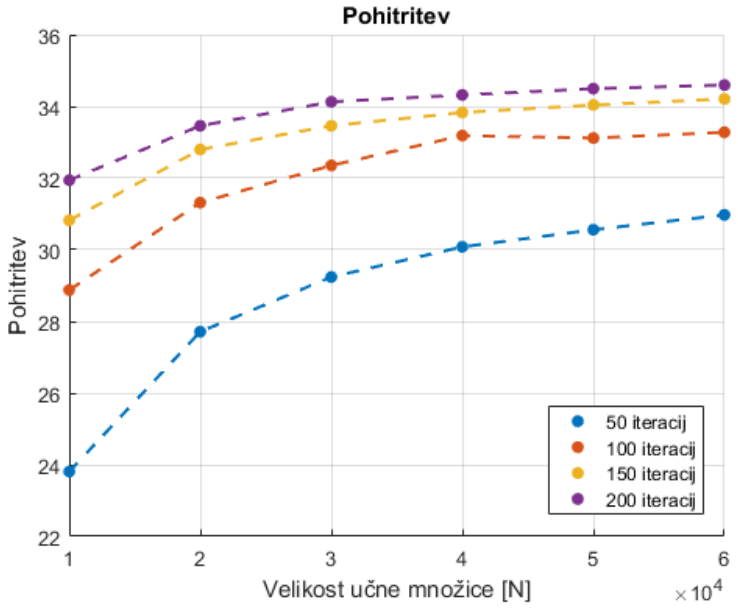
I \ N	10k	20k	30k	40k	50k	60k
50	23,8	27,7	29,2	30,1	30,6	31,0
100	28,9	31,3	32,4	33,2	33,1	33,3
150	30,8	32,8	33,5	33,8	34,0	34,2
200	31,9	33,5	34,1	34,3	34,5	34,6

Tabela 12: Pohitritev algoritma z OpenCL, za različne učne množice N in števila iteracij I.

Najprej lahko opazimo, da z večanjem učne množice pohitritev narašča in se približuje neki asimptotični vrednosti. Podatkov mora biti dovolj, da odtehtamo odvečno delo, ki ga imamo s kopiranjem podatkov na grafično kartico in pripravo ščepcov.

Podobno velja tudi za število iteracij. Večje število iteracij poveča razmerje med koristnim delom na grafični kartici in odvečnim delom gostitelja.

Največja pohitritev je bila dosežena pri učni množici velikosti 60000 z 200 iteracijami gradientnega spusta. V tem primeru znaša pohitritev 34.6. Rezultat je v primerjavi z večnitnim algoritmom na procesorju in koprocesorju XeonPhi precej boljši.



Slika 12: Pohitritev algoritma v odvisnosti od velikosti učne množice.

7 MPI

7.1 Paralelizacija

MPI je standardiziran sistem, ki omogoča paralelno procesiranje na več-računalniških sistemih. Za razliko od sistemov z deljenim pomnilnikom so tukaj pomnilniki nepovezani. Vsak procesor ima dostop samo do svojega pomnilnika, zato procesorji med seboj komunicirajo s pošiljanjem sporočil.

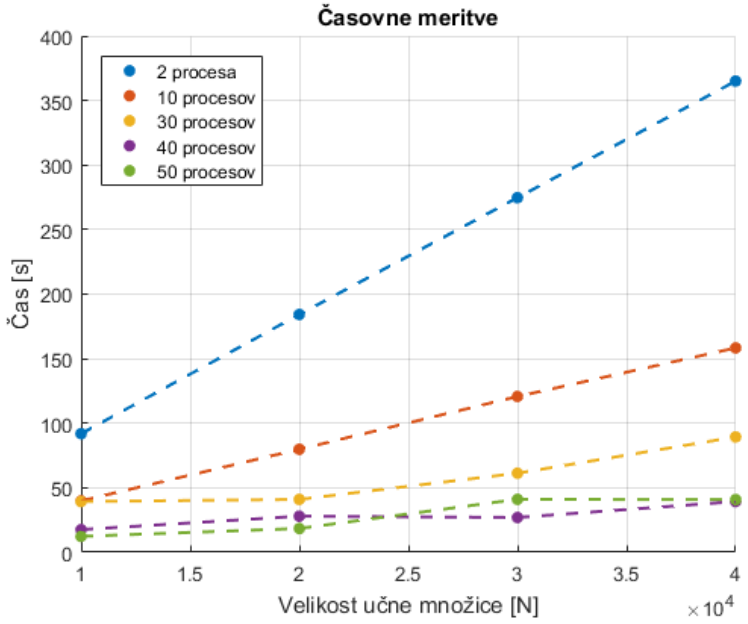
Paralelizacija ni zahtevala večjih sprememb v kodi, potrebno jo je bilo prilagodit klicem funkcij MPI. Vsak procesor dobi svoj del učne množice z raztrosom (angl. scatter), ki jo je zaradi lažjega operiranja z MPI funkcijami treba predstaviti kot eno-dimenzionalni vektor, s čimer je bilo potrebno spremeniti način indeksiranja v programu. Po končanem računanju delnih matrik se vsi rezultati seštejejo (angl. reduction) na glavnem procesorju z indeksom 0, ki za naslednjo iteracijo gradientnega spusta izračuna vektor parametrov nevronske mreže in jih pošlje (angl. broadcast) vsem ostalim procesorjem.

7.2 Meritve

Za testiranje sva uporabila Arnesov grid Sling. Število skritih nevronov sva fiksirala na 25. Spreminjala sva velikost učne množice N od 10000 do 40000 s korakom 10000. Število iteracij gradientnega spusta sva fiksirala na 100, vsako učenje na učni množici je bilo izvedeno 10krat. Zaradi težav grida Sling z izvajanjem programa na 1 procesorju sva meritve izvedla začenši z 2 procesorjema. Rezultati meritev so prikazani v tabeli 13.

P \ N	10k	20k	30k	40k
2	91,87 ± 0,17	184,23 ± 0,21	274,73 ± 0,45	365,041 ± 1,66
10	39,86 ± 0,29	79,82 ± 1,44	120,71 ± 0,82	158,338 ± 0,31
30	39,33 ± 0,90	40,99 ± 0,01	61,29 ± 0,11	89,381 ± 0,54
40	17,60 ± 0,81	27,99 ± 0,40	26,99 ± 1,11	39,338 ± 0,50
50	12,36 ± 1,01	18,48 ± 0,63	41,07 ± 5,07	40,857 ± 1,70

Tabela 13: Časovne meritve za MPI, za različne učne množice N in števila procesov P.



Slika 13: Časovne meritve izvajanja na gruči v odvisnosti od velikosti učne množice.

7.3 Analiza meritev

Iz slike 13 lahko razberemo da z večanjem velikosti množice in števila iteracij, čas izvajanja raste linearno tako kot sva pričakovala, vendar če natančno pogledamo je na določenih poskusih prihajalo do odstopanj od pričakovanj. Recimo pri meritvah za velikost učne množice 10000 z 10 in 30 procesorjev vidimo, da sta skoraj enaki. Najverjetneje je razlog temu bila obremenjenost mreže na Slingu.

P \ N	10k	20k	30k	40k
10	2,3	2,3	2,3	2,3
30	2,3	4,5	4,5	4,1
40	5,2	6,6	10,2	9,3
50	7,4	9,9	6,7	8,9

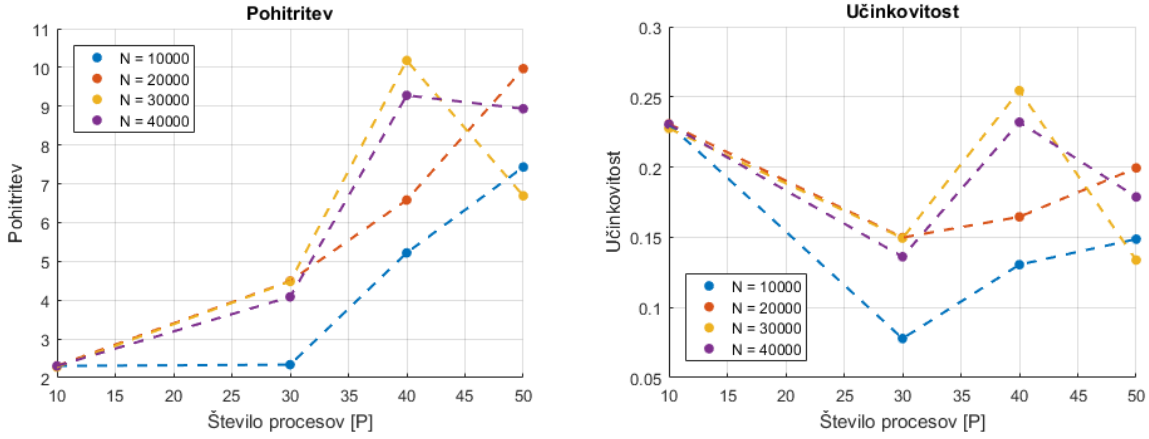
(a) pohitritev

P \ N	10k	20k	30k	40k
10	0,23	0,23	0,23	0,23
30	0,08	0,15	0,15	0,14
40	0,13	0,16	0,25	0,23
50	0,15	0,20	0,13	0,18

(b) učinkovitost

Tabela 14: Tabelarični prikaz pohitritve in učinkovitosti. N je velikost učne množice, P pa število procesov.

Vidimo lahko, da z večanjem števila procesorjev pohitritev načeloma narašča. Problemi z obremenjenostjo mreže na Slingu so razvidni na sliki 14a. Največja pohitritev je bila dosežena pri učni množici s 30000 primeri in 40 procesorji in je približno 10kratna. Ustrezna učinkovitost pri tej meritvi je bila 25%, kot je razvidno iz 14. Učinkovitost na sliki 14b je relativno majhna, kar je tudi pričakovano saj je bila paralelizacija implementirana tako, da veliko dela opravi glavni procesor.



(a) (b)

Slika 14: (a): Pohitritev algoritma glede na število procesov. (b): Učinkovitost procesov glede na njihovo število. V obeh primerih so narisani grafi pri različnih velikostih učne množice N.

8 Sklepne ugotovitve

Zaključimo lahko, da je učenje nevronskih mrež primeren algoritem za paralelizacijo. Večji del učenja lahko namreč implementiramo z množenjem matrik, kar je precej ugodno. To potrjujejo tudi rezultati, saj je pohitritev bila dosežena v vseh uporabljenih tehnologijah za paralelizacijo.

Največja pohitritev je bila dosežena na GPE (Grafična procesna enota) z OpenCL programskim vmesnikom in znaša $34,6$. V tem primeru se je tudi pokazalo, da večanje učne množice ugodno vpliva na pohitritev, limitira pa proti vrednosti 35.

Glede na te rezultate lahko povzamemo, da je GPE najbolj primerna za paralelizacijo nevronskih mrež. V zadnjem času so precej aktivno raziskovalno področje konvolucijske nevronske mreže, ki se sicer nekoliko razlikujejo od klasičnih nevronskih mrež. Popularnejše implementacije (npr. Caffe [1]) podpirajo učenje na GPE.

Literatura

[1] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[2] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST dataset of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Online; accessed 26-October-2016].

[3] Michael A. Nielsen. Neural Networks and Deep Learning. `http://neuralnetworksanddeeplearning.com/`, 2015. [Online; accessed 26-October-2016].

[4] Stefano Scanzio, Sandro Cumani, Roberto Gemello, Franco Mana, and Pietro Laface. Parallel implementation of artificial neural network training. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4902–4905. IEEE, 2010.