Temperaturna plošča

Rok Grmek, Matej Klemen

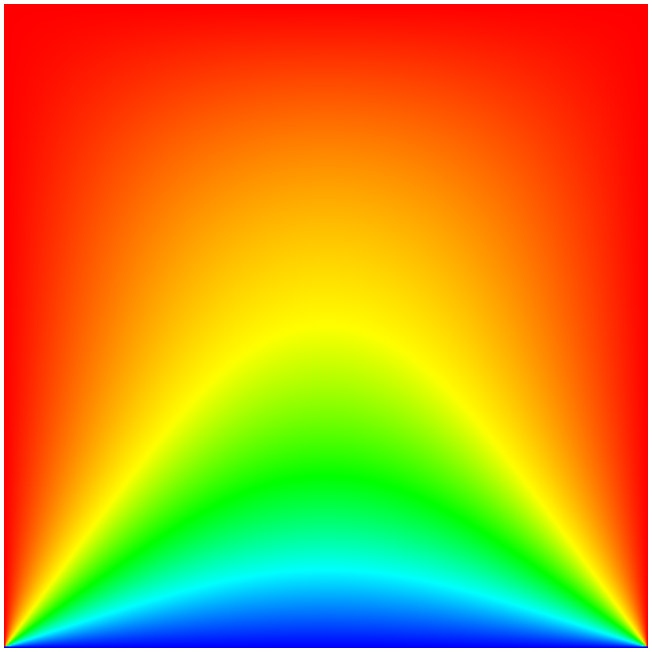
3. november 2017

Kazalo

Uvod

V okviru predmeta Porazdeljeni sistemi rešujeva problem temperaturne plošče. Na tem problemu bova tekom semestra spoznavala različne pristope za paralelno programiranje.

Problem temperaturne plošče je predstavljen s ploščo, ki je na treh straneh (zgoraj, levo in desno) segreta na 100 °C, na spodnji stranici pa ohlajena na 0 °C. Zanima nas, kako se v takem primeru toplota porazdeli po plošči (primer porazdelitve temperature je viden na sliki 1).



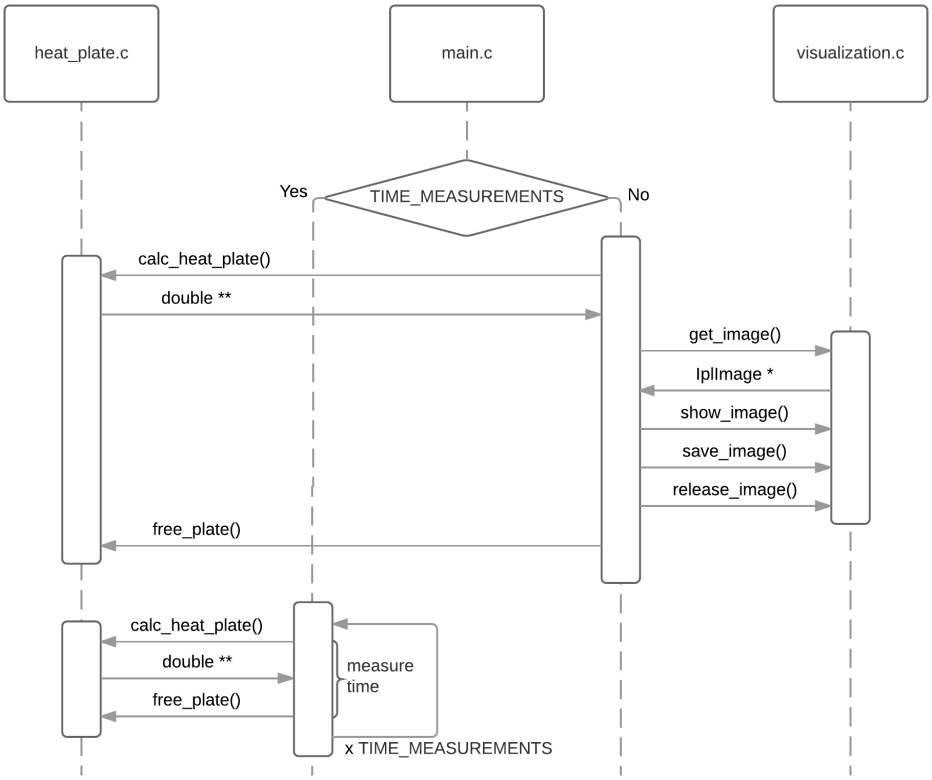
Slika 1: Primer temperaturne plošče, kjer rdeča barva prikazuje najvišjo, temno modra barva pa najnižjo temperaturo.

Iskala bova stacionarno rešitev enačbe $\nabla^2 W = f(x, y)$, pomagala pa si bova z metodo končnih diferenc. Ploščo bova najprej razdelila na mrežo točk in posameznim točkam določila začetne vrednosti. Nato bova v vsaki točki izračunala novo temperaturo na podlagi sosednjih točk. Postopek računanja novih točk se bo ponavljal, dokler rešitev ne skonvergira.

1 Serijski algoritem

1.1 Opis implementacije

Program ima dva načina delovanja. Če je definirana vrednost *TIME_MEASUREMENTS*, se bo program za podane argumente večkrat izvedel in izračunal nekaj uporabnih statistik. Ta način je namenjen predvsem opazovanju časa izvajanja programa. Sicer pa se bo program izvedel 1-krat in na koncu vizualiziral porazdelitev temperature na plošči. V nadaljevanju je opisan slednji način izvajanja, celotno delovanje programa pa je predstavljeno tudi z diagramom zaporedja, vidnim na sliki 2.



Slika 2: Diagram zaporedja, ki prikazuje delovanje programa.

Algoritem sprejme 3 argumente: višino plošče, širino plošče in mejno razliko med dvema iteracijama (v nadaljevanju označeno z ε), na podlagi katere se zaključi računanje. Plošči, ki jo določata vnešena višina in širina, algoritem na vseh štirih stranicah doda pas širine 1, ki služi za nastavljanje robnega pogoja temperature. Novi dimenziji sta torej:

$$višina := višina + 2 \tag{1}$$

$$širina := širina + 2 \tag{2}$$

Na začetku se alocirata dve tabeli dimenzij $višina \times širina$. Ena predstavlja trenutno stanje plošče, druga pa stanje plošče v prejšnji iteraciji. Alokaciji sledi inicializacija plošče - trem

stranicam (levi, desni, zgornji) algoritem nastavi temperaturo na 100 °C, eni (spodnji) pa na 0 °C. Vsem ostalim celicam plošče se zaporedno (*od zgoraj navzdol, od leve proti desni*) dodeli povprečje leve sosednje, zgornje sosednje, povsem desne ter povsem spodnje celice plošče. Eno izmed plošč algoritem izbere kot ploščo trenutnega, drugo pa kot ploščo prejšnjega stanja.

Nato sledi glavna zanka. V vsaki iteraciji gre algoritem skozi "dinamične" celice plošče (celice, ki niso del katerega izmed robov plošče) in za vsako tako celico c[i][j] v i-ti vrstici in j-tem stolpcu po naslednji formuli izračuna novo temperaturo:

$$c[i][j] := \frac{c'[i-1][j] + c'[i][j-1] + c'[i][j+1] + c'[i+1][j]}{4}, \tag{3}$$

kjer $c'[i][j]$ predstavlja temperaturo celice v i-ti vrstici in j-tem stolpcu v prejšnji iteraciji. Ob vsakem izračunu nove temperature algoritem izračuna še absolutno razliko med trenutno (novo) in prejšnjo temperaturo ter jo v primeru, da je to v trenutni iteraciji največja izračunana absolutna razlika temperatur, shrani. Ob koncu iteracije algoritem zamenja vlogi plošč (tista, ki je do sedaj predstavljala prejšnje stanje plošče, bo v naslednji iteraciji vsebovala novo stanje plošče in obratno), pred prehodom v naslednjo iteracijo pa preveri, če je temperaturna razlika v tej iteraciji že manjša od ε (v takem primeru se računanje zaključi).

Na koncu algoritem izpiše število izvedenih iteracij in zažene vizualizacijo končnega stanja temperaturne plošče. O vizualizaciji temperaturne plošče pa je več napisano v naslednjem poglavju.

1.2 Uporabljene knjižnice

Za vizualizacijo temperaturne plošče uporabljava knjižnico *OpenCV*. Najprej se pripravi prazna slika (*cvCreateImage*) z dimenzijami naše plošče, nato pa se za vsako točko na plošči pretvori temperaturo v 3 8-bitne kanale (rdeča, zelena, modra) in vrednosti prepiše na sliko. Če slika z največjo stranico presega *MAX_SIZE*, potem se jo še pomanjša (*cvResize*). Pripravljeno sliko se prikaže (*cvShowImage*) v oknu (*cvNamedWindow*, *cvMoveWindow*, *cvResizeWindow*) in shrani na disk (*cvSaveImage*). Na koncu se le še sprosti zaseden pomnilnik (*cvReleaseImage*). Primer vizualizacije je viden na sliki 1.

1.3 Rezultati

Program je bil testiran na sistemu, katerega specifikacije so navedene v tabeli 1. Da bi k izmerjenemu času čim manj pripomogli stroški režije operacijskega sistema, je bil sistem med testiranjem minimalno obremenjen z drugimi procesi.

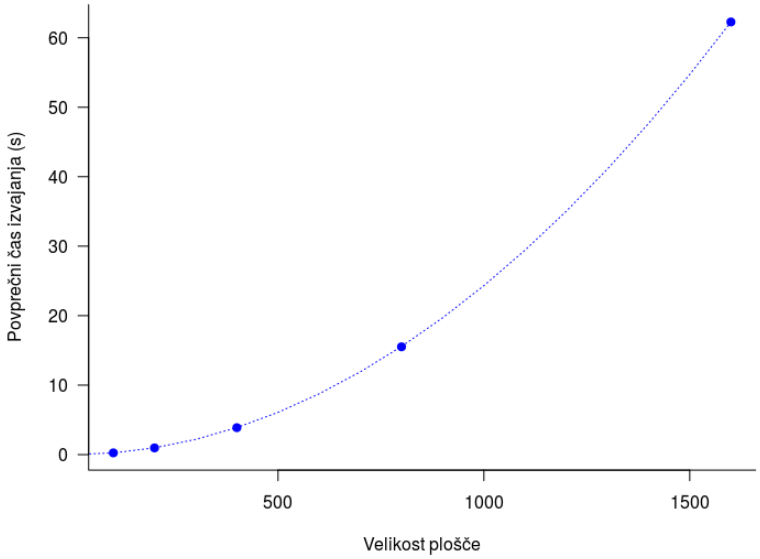
Tabela 1: Specifikacije testnega sistema.

Procesor:	Intel Core i5-4210U
Frekvenca procesorja:	1.70GHz
Število jeder:	2
Maksimalno število niti:	4
Velikost predpomnilnika:	3MB
Velikost in tip glavnega pomnilnika:	16GB DDR3
Grafična kartica:	NVIDIA GeForce 820M 2GB DDR3
Operacijski sistem:	Ubuntu 16.04

Pri testiranju sva se omejila na fiksno vrednost $\varepsilon = 0,01$ in na plošče kvadratne oblike, spreminjala pa sva zgolj velikost stranice. Za vsako izbrano velikost sva algoritem 100-krat zagnala in vsakič izmerila čas izvajanja. Iz meritev sva nato izračunala povprečni čas izvajanja in standardno napako (ta predstavlja razpršenost meritev okrog povprečnega časa). Rezultati so navedeni tabelarično v tabeli 2 in z grafom, prikazanim na sliki 3.

Tabela 2: Povprečni čas izvajanja programa in standardna napaka v odvisnosti od velikosti stranice.

Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
100	0,250	0,000
200	0,965	0,001
400	3,877	0,003
800	15,516	0,006
1600	62,272	0,062



Slika 3: Graf, ki prikazuje povprečni čas izvajanja programa v odvisnosti od velikosti stranice.

Iz rezultatov je vidno, da je povprečni čas izvajanja kvadratno odvisen od velikosti stranice plošče, kar ustreza tudi teoretični časovni zahtevnosti $O(k \cdot h \cdot w)$, kjer je k število iteracij, h višina, w pa širina plošče. V našem primeru je namreč število iteracij konstantno (določeno z ε) in velja zveza $h = w$. Točke na grafu se zelo lepo prilegajo funkciji $t(x) = 2,432 \cdot 10^{-5} \cdot x^2$, iz te pa lahko razberemo, da algoritem na testnem sistemu potrebuje približno $2,432 \cdot 10^{-5}$ s za izračun posamezne točke (v vseh iteracijah skupaj).

2 Paralelizacija s knjižnico "Pthreads"

2.1 Ideja paralelizacije

Do vključno inicializacije plošč poteka vse enako kot v serijskem algoritmu, za tem pa se delo razdeli med $NUM\_THREADS$ niti. Vsaka nit v posamezni iteraciji izračuna svoj delež vrstic. Pred računanjem se meje vrstic določijo na podlagi spodnjih enačb (*index* predstavlja zaporedno številko niti iz intervala $[0, NUM\_THREADS - 1]$, ki jo dobi posamezna nit podano kot argument).

$$spodnja\ meja := 1 + \left\lfloor \frac{index \cdot (h-2)}{NUM\_THREADS} \right\rfloor \tag{4}$$

$$zgornja\ meja := 1 + \left\lfloor \frac{(index+1) \cdot (h-2)}{NUM\_THREADS} \right\rfloor \tag{5}$$

V zgornjih enačbah $(h-2)$ predstavlja število vrstic, katerih temperatura ni fiksno nastavljena. Te razdeliva kar se da enakomerno med niti, na koncu pa mejam še prištejeva 1, saj je temperatura prve vrstice fiksno nastavljena na 100 °C in se računanje začne v drugi vrstici. V takem primeru delitve se število dodeljenih vrstic med posamenzimi nitmi razlikuje za kvečjemu eno vrstico.

Ko niti poračunajo svoj del plošče, morajo počakati na preostale niti, saj je nadaljnje izvajanje programa odvisno od izračunov vseh niti v trenutni iteraciji. To je doseženo s postavitvijo t.i. prepreke (angl. *barrier*), ki nitim ne pusti opravljati nadaljnjega dela, dokler vse niti ne pridejo do mesta, kjer je postavljena.

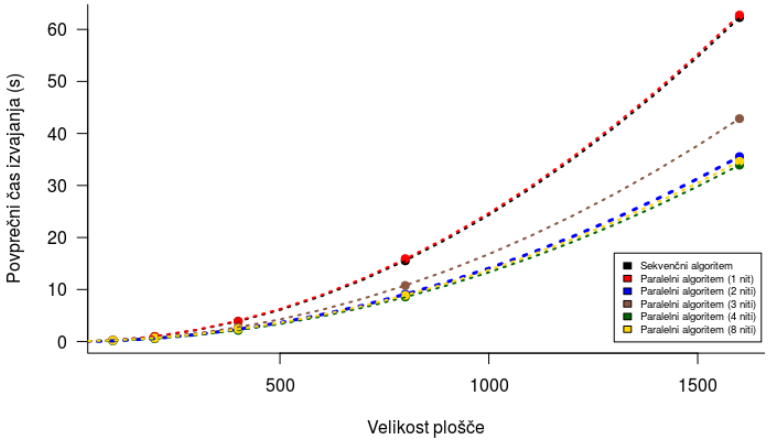
Sledi postopek sinhronizacije največje izračunane absolutne razlike temperature v trenutni iteraciji. Vsaka nit je ta podatek izračunala na svojem delu plošče in ga shranila globalno, v tem delu pa posamezna nit preveri, če so vsi izračunani lokalni maksimumi pod mejo ε. V takem primeru se računanje zaključi, niti se združijo in program poteka od tu naprej enako kot pri serijskem algoritmu. V nasprotnem primeru se niti počakajo (še ena prepreka) in nato hkrati začnejo novo iteracijo.

2.2 Rezultati

Program je bil testiran na istem sistemu (specifikacije so navedene v tabeli 1) in pri enakih pogojih kot serijski algoritem. Za fiksno vrednost $\varepsilon = 0,01$ sva izmerila čase pri različnih velikostih plošče in za različno število niti. Rezultati so vidni v tabeli 3 in na grafu 4.

Tabela 3: Povprečni čas izvajanja paralelnega programa in standardna napaka glede na velikost stranice in število uporabljenih niti z uporabo knjižnice Pthreads.

Število uporabljenih niti	Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
1	100	0,253	0,000
	200	0,974	0,000
	400	3,890	0,002
	800	15,935	0,071
	1600	62,760	0,052
2	100	0,167	0,002
	200	0,578	0,006
	400	2,282	0,020
	800	9,130	0,044
	1600	35,545	0,030
3	100	0,199	0,000
	200	0,729	0,000
	400	2,740	0,000
	800	10,778	0,002
	1600	42,850	0,005
4	100	0,163	0,001
	200	0,558	0,001
	400	2,149	0,000
	800	8,557	0,001
	1600	33,917	0,005
8	100	0,212	0,000
	200	0,680	0,001
	400	2,414	0,001
	800	8,874	0,003
	1600	34,636	0,023

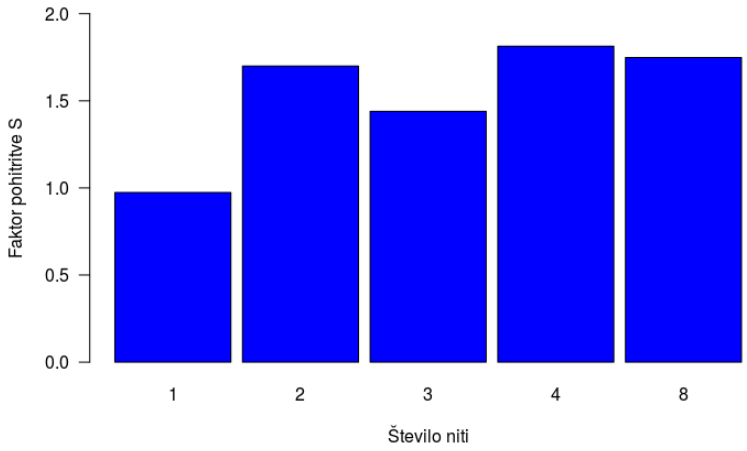


Slika 4: Povprečni čas izvajanja paralelnega programa glede na velikost plošče ob uporabi različnega števila niti (Pthreads).

Za lažjo analizo meritev definiramo **faktor pohitritve** S kot:

$$S = \frac{T_s}{T_p}, \tag{6}$$

kjer je T_s čas izvajanja sekvenčnega programa, T_p pa čas izvajanja paralelnega programa. Največja dosežena pohitritev je bila 1,813-krat, in sicer v primeru štirih niti (toliko je tudi logičnih jeder na testnem sistemu). Ta pohitritev je sicer le malo boljša od tiste v primeru dveh niti, ampak je tak rezultat pričakovan, saj imamo le dve fizični jedri. "Hyper-threading" nam ne prinese bistvene pohitritve, saj je naše računanje zahtevno tudi iz vidika pomnilniških dostopov, tukaj pa si po dve niti delita eno fizično jedro, kar pomeni, da uporabljata tudi isti predpomnilnik in prihaja do številnih zgrešitev. Zanimiv je še primer treh niti, pri katerem dosežemo celo manjšo pohitritev kot v primeru dveh niti. To si lahko razlagamo tako. Naj bo hitrost posamezne niti enaka številu izračunanih celic plošče na enoto časa (merjeno v času, ko nit računa). Če primerjamo primer štirih niti s primerom dveh niti vidimo, da je hitrost posamezne niti manjša, ko uporabljamo štiri niti, ampak je čas, potreben za posamezno iteracijo v tem primeru vseeno manjši, ker imajo posamezne niti dodeljeno le četrtino (in ne polovico) plošče. V primeru treh niti je hitrost primerljiva s primerom štirih niti, ker si dve od treh niti delita eno fizično jedro in je skupna hitrost zaradi čakanja ob prepreki na koncu vsake iteracije odvisna od hitrosti najpočasnejše niti. Čas, potreben za posamezno iteracijo, pa je tukaj še večji od tistega v primeru dveh (in tudi štirih) niti, ker ima vsaka nit dodeljeno tretino plošče. Drugače povedano, ena od niti je hitrejša od ostalih dveh in zaradi enakomerne delitve dela prej konča z računanjem posamezne iteracije, nato pa neučinkovito čaka preostali dve niti. Tudi v primeru večjega števila niti (npr. 8) nam faktor pohitritve rahlo pada, saj čas izvajanja narašča še zaradi režije in preklapljanja med nitmi, hkrati pa že izkoriščamo vse razpoložljive vire sistema. Vse to je vidno tudi na grafu 5.

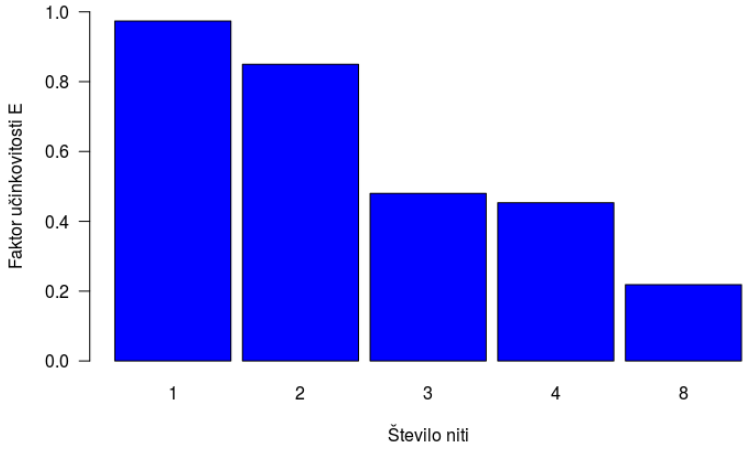


Slika 5: Faktor pohitritve v odvisnosti od števila uporabljenih niti za fiksno velikost plošče 800×800 (Pthreads).

Smiselno je preveriti še **učinkovitost** oziroma izkoristek uporabljenih niti, E, ki jo izračunamo kot:

$$E = \frac{S_N}{N}, \tag{7}$$

kjer je S_N faktor pohitritve in N število niti. Učinkovitost je najboljša v primeru ene same niti (0,974). Rahlo slabšo, a še vedno zadovoljivo učinkovitost dosežemo tudi za dve niti (0,849), pri večjem številu niti pa ta precej pade, saj smo omejeni na zgolj dve fizični jedri. Učinkovitost v odvisnosti od števila niti je vidna na grafu 6.



Slika 6: Učinkovitost v odvisnosti od števila uporabljenih niti za fiksno velikost plošče 800×800 (Pthreads).

3 Paralelizacija s knjižnico "OpenMP"

3.1 Ideja paralelizacije

Paralelizacija poteka na podoben način kot paralelizacija s knjižnico Pthreads (za katero je ideja paralelizacije opisana v poglavju 2.1). Na začetku vsake iteracije se delo statično razdeli med vse niti - vsaka nit dobi $\frac{\check{s}tevilo\ vrstic\ -\ 2}{\check{s}tevilo\ niti}$ vrstic. Če $\check{s}tevilo\ vrstic - 2$ ni deljivo s $\check{s}tevilo\ niti$ (ostane $0 < r < \check{s}tevilo\ niti$), potem se prvim r nitim dodeli še po 1 vrstico.

Zatem vsaka nit poračuna svoj del plošče, na koncu pa niti sinhronizirajo maksimalne absolutne razlike med starimi in novimi temperaturami, ki so jih izračunale na svojem delu plošče v trenutni iteraciji. Če je največja izmed teh temperatur manjša od meje konvergence ε, se izračun konča, sicer pa se začne nova iteracija in ponovi postopek.

V primerjavi s paralelizacijo s knjižnico Pthreads je bilo tukaj potrebno manj dela. Če zanemarimo vključevanje knjižnice OpenMP in nastavitev števila niti, ki jih program uporablja, je bila paralelizacija dosežena zgolj z naslednjo vrstico:

```
#pragma omp parallel for schedule(static) reduction(max: max_diff).
```

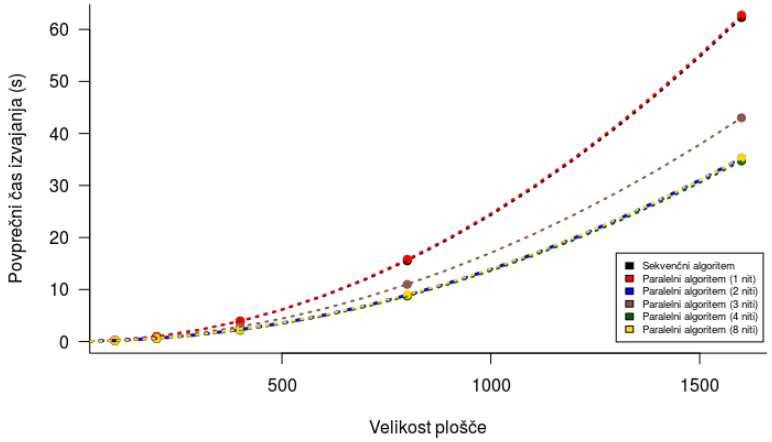
To navodilo prevajalniku pove, naj poskrbi, da se delo porazdeli tako, kot je bilo navedeno v prvem odstavku tega poglavja ter da se niti na koncu počakajo in sinhronizirajo svoje lokalne podatke.

3.2 Rezultati

Program je bil testiran na sistemu, opisanem v tabeli 1. Izmerjeni časi so zelo podobni časom, izmerjenim na programu, ki je bil paraleliziran s knjižnico Pthreads. Podobni rezultati so bili pričakovani, saj sta ideji paralelizacije pri obeh knjižnicah skoraj enaki - minimalna razlika se pojavi zgolj pri delitvi dela med niti. Poleg tega je lahko minimalni časovni pribitek ali pa odbitek posledica različno učinkovitih implementacij paralelnih konstruktov v posamezni knjižnici. Rezultati meritev so predstavljeni v tabeli 4 in na grafu 7.

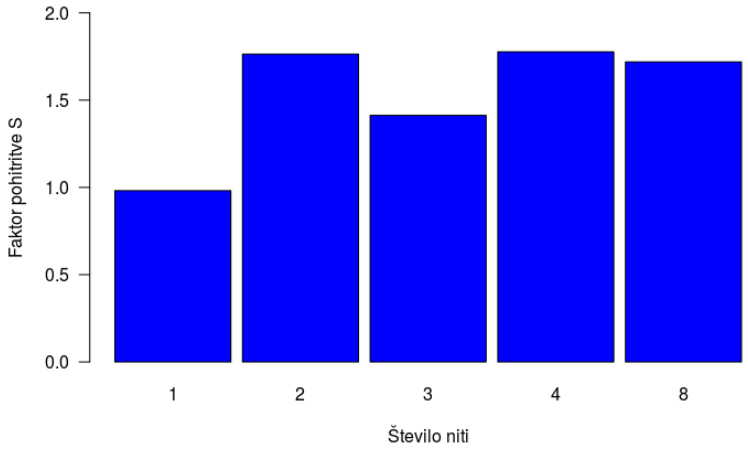
Tabela 4: Povprečni čas izvajanja paralelnega programa in standardna napaka glede na velikost stranice in število uporabljenih niti z uporabo knjižnice OpenMP.

Število uporabljenih niti	Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
1	100	0,253	0,000
	200	0,973	0,000
	400	3,983	0,007
	800	15,808	0,021
	1600	62,756	0,038
2	100	0,146	0,000
	200	0,554	0,000
	400	2,508	0,025
	800	8,798	0,001
	1600	35,274	0,002
3	100	0,184	0,000
	200	0,690	0,000
	400	2,913	0,001
	800	10,982	0,018
	1600	43,008	0,069
4	100	0,140	0,000
	200	0,539	0,000
	400	2,148	0,001
	800	8,734	0,006
	1600	34,699	0,006
8	100	0,193	0,001
	200	0,643	0,001
	400	2,284	0,002
	800	9,025	0,002
	1600	35,343	0,105



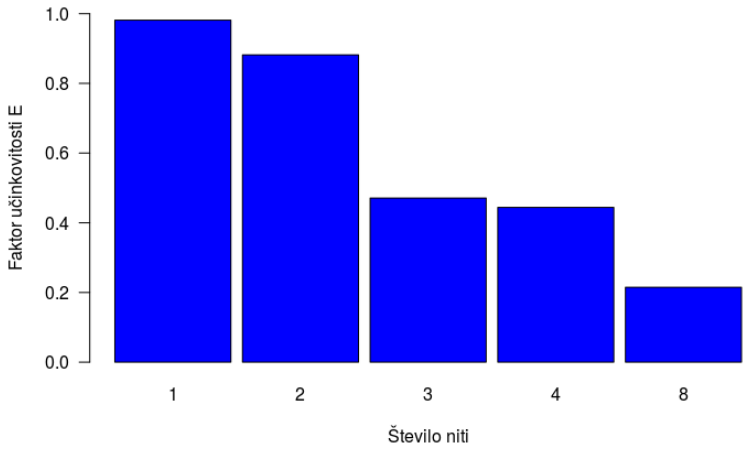
Slika 7: Povprečni čas izvajanja paralelnega programa glede na velikost plošče ob uporabi različnega števila niti (OpenMP).

Za eno od plošč sva za različna števila niti izračunala faktor pohitritve po enačbi 6. Največja pohitritev (1,777-krat) je bila dosežena z uporabo štirih niti (toliko je tudi logičnih jeder). Vse vrednosti so grafično predstavljene na sliki 8, razlaga posameznih vrednosti pa je na voljo v poglavju 2.2, kjer so opisani popolnoma skladni rezultati paralelizacije programa s knjižnico Pthreads.



Slika 8: Faktor pohitritve v odvisnosti od števila uporabljenih niti za fiksno velikost plošče 800×800 (OpenMP).

Po enačbi 7 sva izračunala še učinkovitost v odvisnosti od števila niti, rezultati pa so vidni na sliki 9. Učinkovitost v primeru ene niti je bila tokrat 0,982, v primeru dveh niti pa 0,882. Ostale vrednosti so bistveno nižje, saj ima testni sistem le dve fizični jedri (rezultati so spet skladni s tistimi iz poglavja 2.2, kar je zaradi enake ideje paralelizacije popolnoma smiselno).



Slika 9: Učinkovitost v odvisnosti od števila uporabljenih niti za fiksno velikost plošče 800×800 (OpenMP).

4 Paralelizacija s knjižnico "OpenCL"

4.1 Ideja paralelizacije

Algoritem, pripravljen s knjižnico OpenCL je sestavljen iz dveh delov. En del se izvaja na gostitelju (CPE), drugi pa na napravi, ki je podprta z OpenCL platformo (v našem primeru je to GPE).

Program, ki se bo izvajal na napravi, je opisan kot procedura, ki se bo izvedla za posamezen delovni element (angl. work-item). Pri problemu temperaturne plošče nam delovni element predstavlja enkratno računanje nove temperature za eno od točk na plošči, postopek pa je sledeč. Najprej se na podlagi globalnega indeksa delovnega elementa preveri, če pripadajoča točka leži na notranji strani roba plošče. V takem primeru se iz sosednjih točk druge plošče (ki predstavlja stanje temperaturne plošče v prejšnji iteraciji) izračuna novo temperaturo. Če razlika med novo izračunano temperaturo in prejšnjo temperaturo še vedno presega dano mejo ε, potem ponastavimo globalno zastavico *all_difs_below_eps* na 0 in s tem signaliziramo, da obstaja vsaj ena točka, ki zahteva še nekaj iteracij računanja (na začetku vsake iteracije gostitelj postavi zastavico na 1, računanje pa se lahko zaključi, če nobena od točk ne ponastavi zastavice na 0).

Preden pa se lahko računanje na napravi sploh začne, mora gostitelj pripraviti okolje. Ta najprej prebere datoteko s programom (*kernel.cl*), ki se bo izvajal na napravi. Nato se opravi poizvedba o razpoložljivih OpenCL platformah in napravah. Za želeno napravo se pripravita kontekst in ukazna vrsta, iz prej prebrane datoteke pa se v okviru konteksta pripravi objekt tipa *cl_program*. Ta se za izbrano napravo prevede, na koncu pa se pripravi še ščepec (objekt tipa *cl_kernel*), katerega bomo pozneje poganjali na napravi. Gostitelj poskrbi še za alokacijo in inicializacijo ene plošče v svojem pomnilniku, dve kopiji inicializirane plošče pa pripravi v globalnem pomnilniku naprave (na teh bo naprava računala). Poleg plošč se v globalnem pomnilniku naprave rezervira še mesto za zastavico *all_difs_below_eps*. Za smiseln zagon ščepca potrebujemo še dva argumenta - *local_item_size* in *global_item_size*. Prvi določa velikost delovne skupine (po koliko delovnih elementov naj se izvaja na eni računski enoti) in je v našem programu definiran kot konstanta *WORKGROUP_SIZE* (pri določanju te vrednosti moramo biti pozorni na zgornjo mejo, saj je ta odvisna od naprave). Drugi označuje celotno število delovnih elementov in je izračunan na podlagi velikosti plošče in velikosti delovne skupine (število vseh delovnih elementov mora biti večkratnik velikosti delovne skupine, hkrati pa želimo imeti vsaj toliko delovnih elementov kot je vseh točk na plošči). S tem imamo vse pripravljeno za računanje. Gostitelj na začetku vsake iteracije postavi zastavico *all_difs_below_eps* v globalnem pomnilniku naprave na 1, nato požene računanje nove plošče na napravi, zamenja vlogi plošč, poveča števec iteracij in preden začne z novo iteracijo preveri stanje zastavice. Kadar ostane zastavica po računanju temperatur nastavljena na 1, se računanje lahko zaključi. Po zadnji iteraciji se končni rezultat (plošča iz globalnega pomnilnika naprave) prenese v pomnilnik gostitelja, vsi preostali viri pa se počistijo.

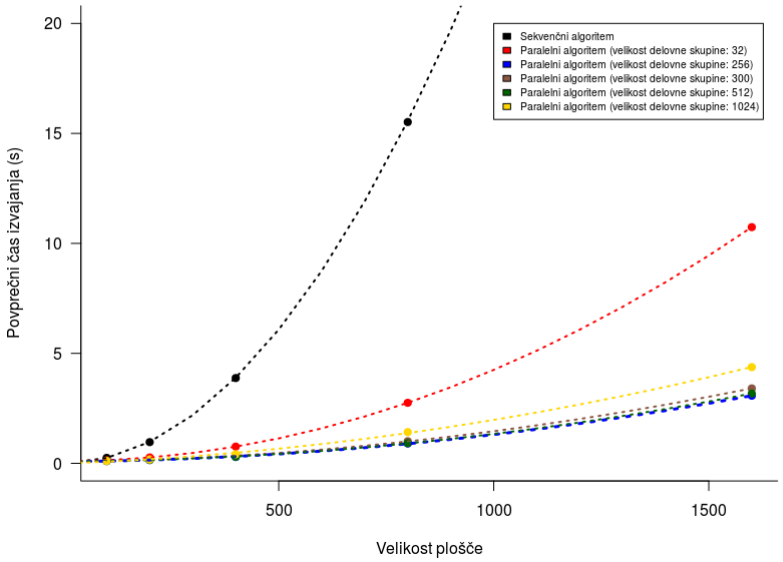
4.2 Rezultati

Program je bil testiran na sistemu, opisanem v tabeli 1. Splača se omeniti še nekaj podrobnosti o testnem okolju, ki za potrebe prejšnjih testiranj niso bile pomembne:

- pri testiranju je bila uporabljena verzija 1.2 knjižnice OpenCL,
- največja dovoljena velikost delovne skupine je na testnem sistemu 1024,
- velikost snopa na testnem sistemu je 32.

Meritve so bile opravljene pri petih različnih velikostih delovne skupine: 32, 256, 300, 512 in 1024. Te velikosti so bile izbrane zato, da bi iz meritev videli, če je vedno smiselno nastaviti kar največjo možno velikost delovne skupine, ali je to odvisno od primera uporabe.

Rezultati meritev so prikazani v tabeli 5 in na grafu na sliki 10.



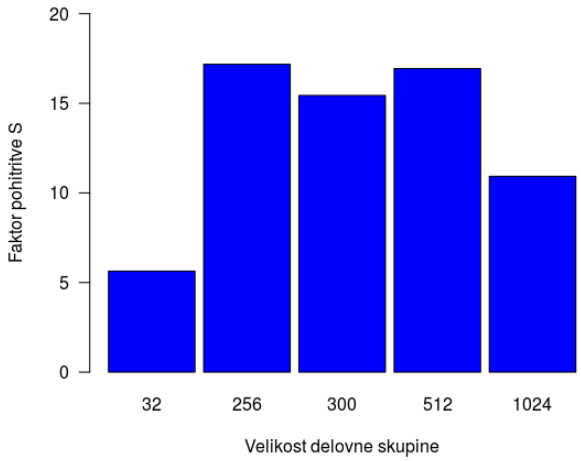
Slika 10: Povprečni čas izvajanja paralelnega programa glede na velikost plošče ob uporabi različnih velikosti delovne skupine (OpenCL). Graf je zaradi preglednosti "odrezan"pri povprečnem času izvajanja 20 sekund, zato je potrebno omeniti, da povprečni čas sekvenčnega algoritma izven vidnega monotono narašča in je pri velikosti plošče 1600 enak 62.272 sekund.

Tabela 5: Povprečni čas izvajanja paralelnega programa in standardna napaka glede na velikost stranice in velikost delovne skupine pri fiksni meji konvergence $\epsilon = 0.01$ z uporabo knjižnice OpenCL.

Velikost delovne skupine	Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
32	100	0,139	0,000
	200	0,268	0,001
	400	0,761	0,000
	800	2,752	0,000
	1600	10,738	0,001
256	100	0,111	0,000
	200	0,151	0,000
	400	0,290	0,000
	800	0,903	0,001
	1600	3,074	0,000
300	100	0,105	0,000
	200	0,161	0,000
	400	0,332	0,001
	800	1,005	0,000
	1600	3,407	0,000
512	100	0,107	0,000
	200	0,155	0,000
	400	0,296	0,000
	800	0,916	0,000
	1600	3,171	0,000
1024	100	0,110	0,000
	200	0,176	0,000
	400	0,413	0,000
	800	1,420	0,000
	1600	4,373	0,000

Kot je bilo pričakovano, je algoritem, paraleliziran na grafični kartici, pri vseh preizkušenih velikostih delovne skupine boljši (doseže krajši povprečni čas izvajanja) kot sekvenčni algoritem. Pri manjših velikostih plošče je dosežena pohitritev sicer nekoliko nižja kot bi si mogoče želeli, ampak je bilo to pričakovano, saj je v tem primeru relativno malo računanja glede na vso potrebno režijo, grafične kartice pa svojo moč pokažejo predvsem pri računsko zahtevnih opravilih. To dokazujejo tudi rezultati na večjih dimenzijah plošče. Največja dosežena pohitritev glede na sekvenčni algoritem je bila dosežena pri dimenziji plošče 1600 × 1600 - sekvenčni algoritem je v povprečju za ta izračun potreboval 62,272 sekund, paralelni algoritem na grafični kartici pa pri najboljši nastavitvi velikosti delovne skupine zgolj 3,074 sekunde.

Slika 11 prikazuje faktor pohitritve (določen z enačbo 6) v odvisnosti od velikosti delovne skupine. Največja pohitritev je bila dosežena pri velikosti delovne skupine 256 (približno **17**-kratna), najmanjša pa pri velikosti delovne skupine 32 (približno **6**-kratna).



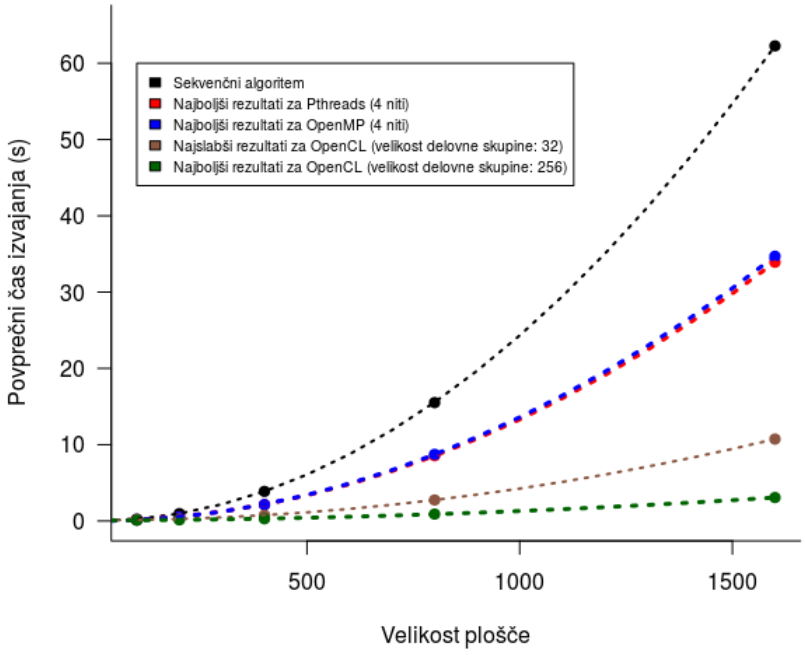
Slika 11: Faktor pohitritve pri različnih velikostih delovne skupine ter fiksni velikosti plošče 800×800 (OpenCL).

Iz rezultatov je vidno, da nastavljanje velikosti delovne skupine na največjo možno velikost ni nujno najboljša ideja. Najkrajši povprečni časi izvajanja namreč niso bili izmerjeni pri velikosti plošče 1024 (kolikor je največja dovoljena velikost skupine na grafični kartici na testnem sistemu), pač pa pri velikosti skupine 256. Izjema je zgolj velikost temperaturne plošče 100 × 100, kjer pa je izmerjeni povprečni čas nekaj tisočink sekunde večji od najboljšega. OpenCL omogoča tudi možnost, da ne določimo velikosti skupine eksplicitno, pač pa pustimo, da se ta določi samodejno, vendar s tem nisva nič pridobila. Časi so bili približno enaki, nekje pa celo slabši kot pri ročno nastavljeni velikosti delovne skupine 256. Za najboljše rezultate, je torej bolje eksperimentalno določiti velikost delovne skupine kot pa se zanašati, da bo sistem res našel najboljšo nastavitev.

Posebno zanimive so še meritve pri velikosti delovne skupine 300. Kljub temu, da je velikost

delovne skupine večja, sistem pri teh nastavitvah za skoraj vse testirane dimenzije plošče doseže slabše rezultate kot pri velikosti delovne skupine 256. Razlog za to je, da mora biti velikost delovne skupine za optimalno izkoriščanje virov večkratnik velikosti snopa. Še en razlog za slabše dosežene povprečne čase je bil opisan v prejšnjem odstavku (večanje velikosti delovne skupine ne prinese nujno krajšega časa izvajanja).

Za konec pa poglejmo še, kako dobro se je algoritem, paraleliziran na grafični kartici, obnesel v primerjavi s prejšnjimi metodami paralelizacije (več o načinu paralelizacije s knjižnicama Pthreads in OpenMP je napisano v poglavjih 2 in 3). Na sliki 12 so prikazani povprečni časi izvajanja za različne velikosti temperaturne plošče, doseženi z različnimi načini paralelizacije. Za boljšo predstavo o doseženih izboljšavah so na grafu tudi povprečni časi, doseženi s sekvenčnim algoritmom. Paralelizacija na grafični kartici doseže tako v najboljšem kot tudi v najslabšem primeru krajše povprečne čase izvajanja kot vsi prejšnji poizkusi paralelizacije. Temu je tako, ker ima naš problem visoko stopnjo podatkovnega paralelizma (z dovolj velikim številom računskih enot bi lahko tudi celotno ploščo izračunali v enem koraku), to je pa posebej ugodno za paralelizacijo z grafično kartico.



Slika 12: Primerjava povprečnih časov izvajanja sekvenčnega algoritma in različnih verzij paralelnega algoritma pri različnih dimenzijah temperaturne plošče.

5 Paralelizacija s knjižnico "OpenMPI"

5.1 Ideja paralelizacije

Paralelni programi, pripravljeni s knjižnico OpenMPI, se v celoti izvajajo vzporedno (ob zagonu navedemo želeno število procesov). Delitev dela je organizirana na podlagi identifikacijskih številk procesov, na enak način pa lahko po potrebi poskrbimo še za vejitve, če želimo, da eden od procesov deluje kot gospodar.

V našem primeru temperaturne plošče smo se odločili, da procese navidezno razporedimo v mrežo $\sqrt{p} \times \sqrt{p}$, vsakemu pa se dodeli pripadajoč blok plošče. Takšna delitev dela zahteva (v primerjavi z delitvijo na vodoravne pasove) nekoliko več organizacije pri izmenjevanju robov plošče, vendar se v celoti gledano pošilja manjša količina podatkov. Za ploščo velikosti $s \times s$ in p procesov se izkaže, da pri delitvi na pasove izmenjamo $2 \cdot s \cdot (p-1)$ točk na iteracijo, z mrežo blokov pa izmenjamo le $4 \cdot s \cdot (\sqrt{p}-1)$ točk na iteracijo, kar prinese pri večjem številu procesov kar veliko razliko. Po drugi strani nas takšna delitev rahlo omejuje, saj lahko izkoristimo le toliko virov, za kolikor se takšna delitev izide (1, 2, 4, 9, 16...). V programu zato takoj na začetku obdržimo le $\lfloor \sqrt{p} \rfloor^2$ procesov.

Na vsakem od obdržanih procesov se nato pripravita dve celotni plošči, za tem pa se izračunajo še meje, v katerih bo dan proces računal. Taka rešitev je sicer precej pomnilniško potratna (vsak proces ima shranjeno celotno ploščo, potrebuje pa le svoj del), ampak se je v praksi izkazala za boljšo kot npr. inicializacija plošče na enem procesu in pošiljanje delov plošče med ostale ali pa porazdeljena inicializacija delov plošče (naša implementacija inicializacije namreč vsebuje nekaj odvisnosti med sosednjimi točkami, kar prinese potrebo po dodatni komunikaciji med procesi).

V tej točki je vse pripravljeno in začnemo z računanjem. Vsak proces izračuna svoj del in lokalno največjo izračunano razliko. Za tem se opravi skupna redukcija, kjer kot rezultat vsak proces dobi globalno maksimalno razliko (ko je ta dovolj majhna se zaključi računanje), hkrati pa ta ukaz služi tudi kot prepreka na kateri se procesi časovno sinhronizirajo. Zabeležimo še novo število iteracij in zamenjamo vlogi plošč, za tem pa sledi izmenjevanje robov. Vsak proces najprej neblokirajoče pošlje robove vsem sosedom, nato pa blokirajoče prejema želene podatke. Takoj, ko posamezen proces prejme vse podatke, lahko brez prepreke nadaljuje z računanjem naslednje iteracije, po računanju pa se procesi zaradi skupne redukcije spet sinhronizirajo. Proces izmenjevanja robov, bi lahko zaradi velikih stroškov vzpostavljanja komunikacije izboljšali tako, da bi izmenjali širši rob, nato pa bi nekaj iteracij računali brez izmenjave robov (v takem primeru bi sicer imeli nekaj več računanja, ampak je to cenejše od komunikacije).

Ob koncu se izračunane dele plošče pošlje iz posameznih procesov na prvega (gospodar), ta pa podatke sprejme, organizira v celoto in nato ploščo shrani kot datoteko. Za tem procesi le

še počistijo rezerviran pomnilnik.

5.2 Rezultati

5.2.1 Rezultati na lokalnem testnem sistemu

Program smo lokalno testirali na enakem testnem sistemu kot pri prejšnjih testiranjih (specifikacije so navedene v tabeli 1), rezultati meritev pa so predstavljeni v tabeli 6.

Za lažjo primerjavo smo meritve časov opravili za popolnoma enak nabor parametrov kot pri testiranju s knjižnicama Pthreads in OpenMP, vendar je tukaj pomembno dejstvo, da se program zaradi posebne delitve dela vedno izvaja le na $\lfloor\sqrt{p}\rfloor^2$ procesih (p je zahtevano število procesov). V tabeli 6 smo zato poleg zahtevanega števila procesov navedli še število aktivnih procesov za primere, ko se ti dve števili razlikujeta.

Smiselno lahko torej obravnavamo le primere, kjer smo zahtevali 1 in 4 procese. Ti rezultati so zelo podobni rezultatom, ki smo jih pri enakih parametrih dobili s paralelizacijo s knjižnicama Pthreads in OpenMP, kar je bilo tudi pričakovano. Medprocesna komunikacija znotraj enega vozlišča je pri MPI najverjetneje implementirana kar z uporabo deljenega pomnilnika in je zato časovna zakasnitev pri izmenjevanju robov zelo majhna.

Pri ostalih primerih (2, 3 in 8 zahtevainh procesov) so izmerjeni časi seveda primerljivi s časi meritev, pri katerih smo že sami zahtevali toliko procesov, kolikor jih je bilo pri računanju dejansko aktivnih. Nekaj več časa se je očitno porabilo le na račun upravljanja z dodatnimi neaktivnimi procesi.

Tabela 6: Povprečni čas izvajanja paralelnega programa (na lokalnem testnem sistemu) in standardna napaka glede na velikost stranice in število uporabljenih procesov z uporabo knjižnice OpenMPI.

Število uporabljenih procesov	Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
1	100	0,312	0,000
	200	1,040	0,004
	400	3,961	0,013
	800	15,547	0,013
	1600	65,138	0,104
2 (1)	100	0,350	0,000
	200	1,153	0,003
	400	4,393	0,004
	800	17,424	0,014
	1600	72,692	0,044
3 (1)	100	0,357	0,000
	200	1,162	0,003
	400	4,400	0,004
	800	17,432	0,016
	1600	72,757	0,074
4	100	0,219	0,000
	200	0,603	0,000
	400	2,140	0,000
	800	8,388	0,003
	1600	34,598	0,012
8 (4)	100	0,302	0,000
	200	0,723	0,000
	400	2,417	0,004
	800	9,316	0,016
	1600	38,208	0,072

5.2.2 Rezultati na omrežju grid

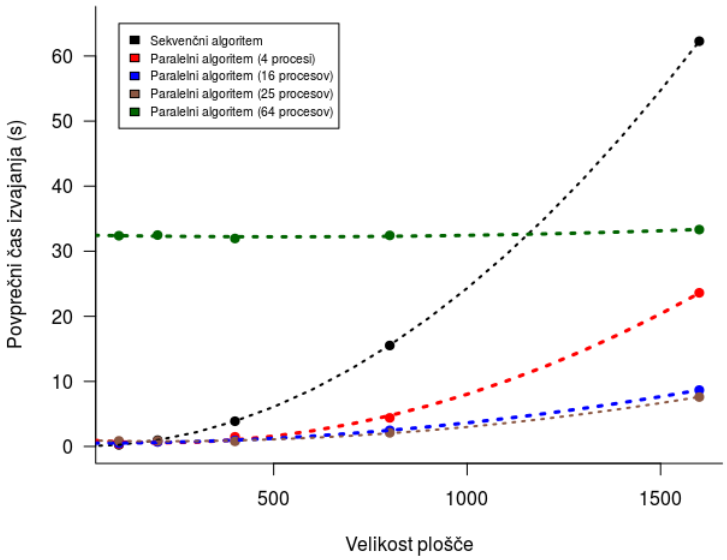
Program, paraleliziran s knjižnico OpenMPI, sva testirala tudi na omrežju grid, ki ga upravlja konzorcij SLING (slovenska iniciativa za nacionalni grid).
Ta uporabnikom omogoča preprost dostop do razpršenega omrežja računskih virov za namen paralelnega računanja in obdelave podatkov. Iniciativo vodi Arnes, med člani skupine SLING pa so še organizacije, kot so Institut Jožef Stefan, Agencija republike Slovenije za okolje, Comtrade, itd..

Program sva testirala na 4, 16, 25 in 64 jedrih. Več kot s 64 jedri programa nisva testirala predvsem zato, ker se čas čakanja na dodelitev virov na gridu močno poveča, če zahtevamo več virov. Rezultati testiranja so vidni v tabeli 7 in na sliki 13.

Vidno je, da za majhno velikost plošče večje število procesov doseže slabše rezultate kot manjše število procesov. Razlog je v tem, da z večanjem števila procesov povečujemo čas komunikacije med procesi, količina računanja pa ostaja enaka. Prejšnjo poved dokazuje dejstvo, da se nam pri večjih dimenzijah plošče uporaba večjega števila procesov obrestuje (do neke meje) - primer tega je čas izvajanja za velikost plošče 1600×1600, kjer je čas izvajanja z uporabo 16 in 25 procesov krajši kot tisti z uporabo 4 procesov.

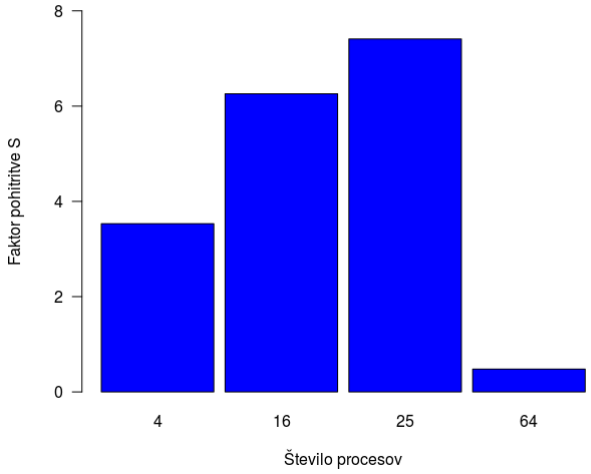
Tabela 7: Povprečni čas izvajanja paralelnega programa (na omrežju SLING) in standardna napaka glede na velikost stranice in število uporabljenih procesov z uporabo knjižnice OpenMPI.

Število uporabljenih procesov	Velikost stranice [št. točk]	Povprečni čas izvajanja [s]	Standardna napaka [s]
4	100	0,352	0,003
	200	0,717	0,009
	400	1,483	0,011
	800	4,394	0,038
	1600	23,611	0,045
16	100	0,473	0,013
	200	0,702	0,001
	400	0,932	0,015
	800	2,479	0,018
	1600	8,661	0,024
25	100	0,858	0,053
	200	0,855	0,001
	400	0,788	0,018
	800	2,094	0,062
	1600	7,601	0,053
64	100	32,383	0,131
	200	32,483	0,092
	400	31,955	0,052
	800	32,432	0,126
	1600	33,325	0,083



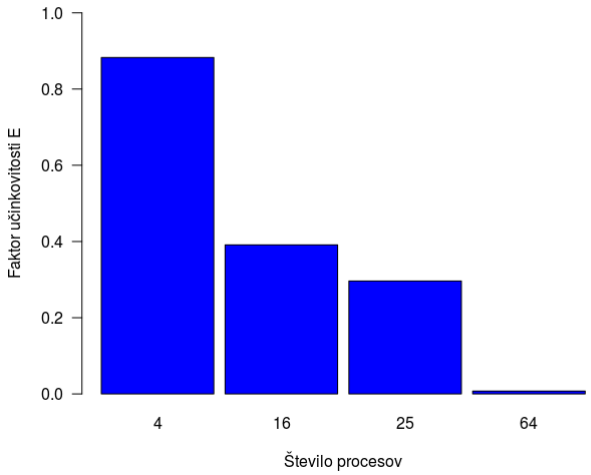
Slika 13: Povprečni čas izvajanja paralelnega programa in standardna napaka glede na velikost stranice in število uporabljenih procesov pri fiksni meji konvergence $\epsilon = 0.01$ z uporabo knjižnice OpenMPI.

Za ploščo velikosti 800×800 smo izračunali faktor pohitritve (enačba 6) za različna števila procesov (vidno na grafu 14). Pohitritev najprej z večanjem števila procesov narašča, nato pa strmo pade, ko prečkamo mejo 32 - toliko je namreč procesorskih jeder na enem od vozlišč v omrežju. Ko mejo prečkamo, pridobimo velik časovni pribitek zaradi visoke latence pri komunikaciji med procesi. To je še lepše prikazano na grafu 13, kjer je vidno, da časovni pribitek ni v veliki meri odvisen od velikosti plošče (od skupne količine podatkov, ki se med računanjem prenaša), ampak je pretežno odvisen od števila zaporednih začetkov prenosa (torej od števila iteracij, ki je v našem primeru približno konstantno za vse velikosti plošče pri fiksnem epsilonu). Vidimo torej, da bi se nam za računanje na omrežju v veliki meri izplačalo implementirati že prej predlagan postopek izmenjevanja nekoliko širšega roba. Tako smo najboljši rezultat dosegli s tem, da smo prenose obdržali na lokalnem nivoju in za 25 procesov dosegli 7,410-kratno pohitritev (25 je namreč največje število, ki ni večje od 32 in ima za koren celo število).



Slika 14: Faktor pohitritve v odvisnosti od števila uporabljenih procesov za fiksno velikost plošče 800×800 (OpenMPI).

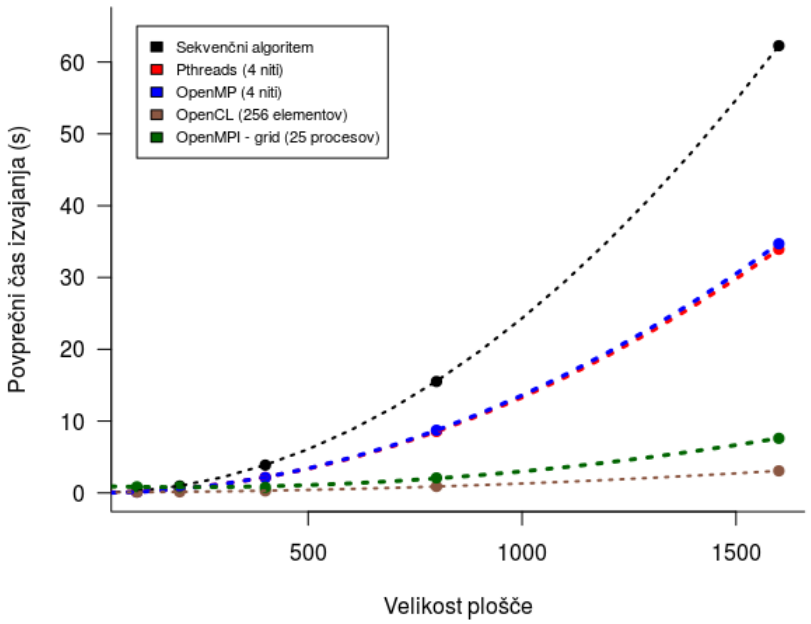
Prej izračunane faktorje pohitritve smo preračunali (enačba 7) še v učinkovitost. Največja učinkovitost (0,883) je bila dosežena pri najmanjšem številu pognanih procesov. Iz grafa na sliki 15 se lepo vidi, da z večanjem števila uporabljenih procesov večamo stroške komunikacije, kar pri fiksni velikosti plošče (kjer je količina računanja vedno enaka) pomeni nižjo učinkovitost na posamezen proces. Ta problem je najlažje rešiti tako, da ob večanju procesov poskušamo latenco prikriti s tem, da poleg tega tudi več računamo (torej povečamo velikost plošče).



Slika 15: Učinkovitost v odvisnosti od števila uporabljenih procesov za fiksno velikost plošče 800×800 (OpenMPI).

6 Pregled uporabljenih metod paralelizacije

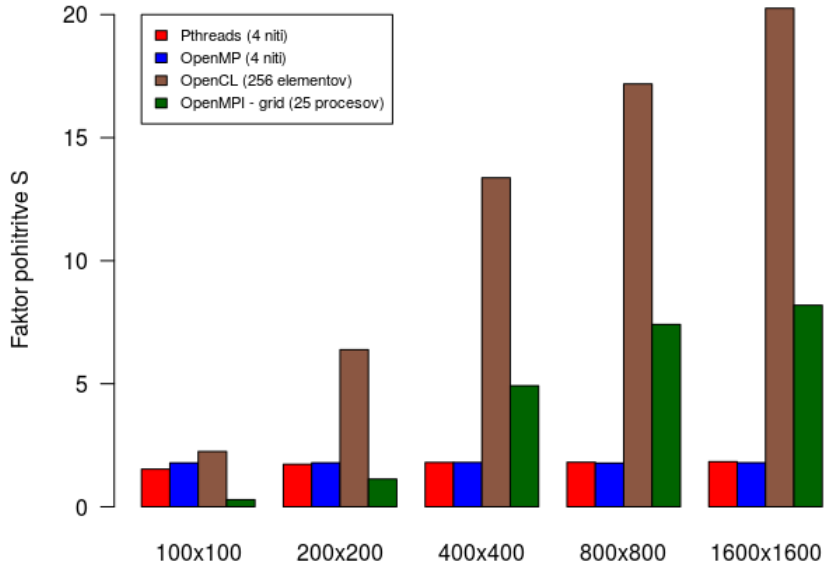
V posameznih poglavjih so bile opisane posamezne metode paralelizacije, to poglavje pa združuje najboljše dosežene rezultate. Graf na sliki 16 prikazuje čas izvajanja sekvenčnega algoritma ter najboljše dosežene čase izvajanja s posameznimi pristopi paralelizacije.



Slika 16: Čas izvajanja sekvenčnega algoritma ter najboljši (najkrajši) doseženi časi izvajanja s posameznimi pristopi paralelizacije.

Pri majhnih velikostih plošče (konkretno je najmanjša velikost plošče, za katero so bile opravljene meritve, tukaj 100×100) so si (povprečni) časi izvajanja precej podobni. To je posledica dejstva, da je tukaj relativno malo računanja v primerjavi s potrebno režijo za vzpostavitev paralelizacije. Zato paralelni programi ves tisti čas, ki so ga pridobili na račun paralelnega izvajanja, izgubijo zaradi režije. Poleg tega je povprečni čas izvajanja pri majhnih velikostih plošče že v primeru sekvenčnega algoritma precej kratek (približno 0,250 sekunde), zato izboljšave niso tako očitne. Najkrajši povprečni čas izvajanja doseže paralelizacija s knjižnico OpenCL, 0,111 sekunde.

Z večanjem velikosti plošče povprečni čas izvajanja hitro narašča (kvadratno v odvisnosti od velikosti stranice). Tukaj se začnejo kazati prednosti uporabe paralelnih pristopov. Kot primer velja omeniti, da pri velikosti plošče 1600×1600 sekvenčni algoritem rabi v povprečju 62,272 sekund, paralelni algoritem na grafični kartici (paraleliziran s pomočjo OpenCL) pa zgolj 3,074 sekunde. Naraščanje faktorja pohitritve z večanjem plošče se še lepše vidi na sliki 17.



Slika 17: Faktor pohitritve, dosežen s posameznimi pristopi paralelizacije za različne velikosti plošče.

Izmed uporabljenih pristopov paralelizacije sta si paralelizacija s knjižnico Pthreads in paralelizacija s knjižnico OpenMP zelo podobni - povprečni časi izvajanja so si zelo blizu, iz tega pa sledi, da so si tudi pohitritve zelo podobne. Majhne razlike so najverjetneje posledica različno učinkovitih implementacij paralelnih konstruktov v posameznih knjižnicah.

Najboljše rezultate je za vse (izbrane) velikosti plošče dosegla paralelizacija na grafični kartici (s knjižnico OpenCL). Na 1600×1600 je dosegla kar 20,258-kratno pohitritev (glede na sekvenčni algoritem). Kljub temu pa to ne pomeni, da bo temu vedno tako - kateri pristop bo deloval najbolje, je močno odvisno od tipa problema. Problem temperaturne plošče ima visoko stopnjo podatkovnega paralelizma in je zato primeren za paralelizacijo na GPE. Če bi naš problem zahteval številne vejitve programskega toka in manj vzporednega računanja, bi pristop z OpenCL dosegel veliko slabše rezultate.

Poleg paralelizacije na grafični kartici, smo precej dobre rezultate dosegli tudi na omrežju grid, kjer smo testirali paralelizacijo s knjižnico MPI. Nekoliko nas je presenetilo dejstvo, da je latenco pri komunikaciji na omrežju zelo težko prikriti. Najboljši rezultat smo namreč dosegli, ko smo število procesov omejili tako, da se je algoritem v celoti izvajal na enem vozlišču. V primeru večjega števila vozlišč se je pri vseh časih pojavil pribitek (približno 30 sekund), so si pa ti časi precej podobni za različne velikosti plošče, kar nakazuje, da bi lahko z ekstremnim večanjem problema dosegli dober rezultat tudi na tak način.

Običajno prav tak pristop (večanje problema) uporabljamo, ko želimo za večje število

vzporednih procesov obdržati enako učinkovitost. Veljati mora naslednja zveza:

$$T_s(n) \geq C \cdot T_{overhead}(n, p), \tag{8}$$

kjer $T_s(n)$ predstavlja čas, ki ga za izvajanje potrebuje serijski algoritem (v odvisnosti od velikosti problema), $T_{overhead}(n, p)$ je skupen odvečni čas, ki ga pridobimo s paralelizacijo (v odvisnosti od velikosti problema in števila paralelnih enot), C je pa konstanta, ki zajema zahtevo po konstantni učinkovitosti. Zgornje čase lahko bolj jasno zapišemo kot:

$$T_s(n) = \sigma(n) + \varphi(n), \tag{9}$$

$$T_{overhead}(n, p) = (p - 1) \cdot \sigma(n) + p \cdot \kappa(n, p), \tag{10}$$

kjer je σ čas izvajanja tistega dela algoritma, ki ga ne moremo paralelizirati, φ čas izvajanja dela algoritma, ki ga lahko paraleliziramo in κ dodaten čas, potreben za komunikacijo in sinhronizacijo med paralelnimi procesi.

V našem primeru bomo za velikost problema uporabili kar velikost stranice plošče, število iteracij pa naj bo konstantno. Na ta način lahko iz obravnave izpustimo še del algoritma, kjer se plošča inicializira, razdeli med procese in na koncu spet združi. Delež časa, ki se porabi za to, lahko namreč enostavno približamo k 0 s tem, ko povečamo zahtevano število iteracij ($\varepsilon \to 0$). To si lahko privoščimo, saj ob tem nimamo nobenih dodatnih pomnilniških zahtev. Če torej upoštevamo le del algoritma, kjer ponavljajoče računamo temperature, bo veljalo naslednje:

$$\sigma(n) = O(0), \tag{11}$$

$$\varphi(n) = O(n^2), \tag{12}$$

$$\kappa(n, p) = O(\frac{n}{\sqrt{p}} + \log p). \tag{13}$$

V enačbi 8 lahko zdaj uporabimo zgornje časovne zahtevnosti in dobimo naslednjo zvezo:

$$n^2 \geq C \cdot (n\sqrt{p} + p \log p). \tag{14}$$

Iz te lahko izpeljemo pogoj, ki nam pove, vsaj kakšna naj bo nova velikost problema, če želimo ob večanju števila paralelnih enot obdržati enako učinkovitost:

$$n^2 - c\sqrt{p} \cdot n - Cp \log p \geq 0, \tag{15}$$

$$n \leq \frac{C\sqrt{p} - \sqrt{C^2 p + 4Cp \log p}}{2} \quad \text{ali} \quad n \geq \frac{C\sqrt{p} + \sqrt{C^2 p + 4Cp \log p}}{2}. \tag{16}$$

Izkaže se, da lahko za naše potrebe prvi del pogoja izpustimo, saj nam ta k rešitvi nikoli ne doda takih n-jev, ki bi bili večji ali enaki 1, zanimajo pa nas le ti. Tako dobimo naslednjo zahtevo:

$$n \geq f(p) \quad \text{za} \quad f(p) = \frac{C\sqrt{p} + \sqrt{C^2 p + 4Cp \log p}}{2}, \tag{17}$$

radi pa bi izvedeli še kakšnega reda bo pomnilniška zahtevnost za posamezno paralelno enoto (v odvisnosti od p). V našem primeru poznamo odvisnost od velikosti problema:

$$M(n) = O(n^2), \tag{18}$$

iz te pa sledi:

$$M(f(p)) = O(p \log p). \tag{19}$$

To pomeni, da sistem ni popolnoma raztegljiv, saj moramo z večanjem števila procesov (ob želji, da dosežemo enako učinkovitost) toliko povečati velikost problema, da nam narašča tudi pomnilniška zahtevnost na posamezni procesni enoti. Na tem mestu lahko spet omenimo, da je tukaj še nekaj prostora za izboljšave, saj ima v naši implementaciji vsak proces shranjeno celotno ploščo (in ne le svoj del), ampak za praktične potrebe ta izboljšava ni bila potrebna. Tudi iz vidika raztegljivosti nam izboljšava ne bi prinesla popolne raztegljivosti, saj bi bila v tem primeru pomnilniška zahtevnost v odvisnosti od števila procesnih enot enaka $M(f(p)) = O(\log p)$ in ne konstantna, kot bi si želeli.

7 Zaključek