Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gašper Kolar, Luka Prijatelj

Simulacija jate ptic

končno poročilo

poraZdeljeni sistemi
univerzitetni študijski program
prve stopnje
računalništvo in informatika

Ljubljana, 2017

Kazalo

Poglavje 1

Uvod

Za temo seminarske naloge pri predmetu smo si izbrali simulacijo jate ptic(ang. Flocking simulation).

Vsako iteracijo je potrebno izračunati novo pozicijo za vsako ptico glede na bližnje ptice. Nova pozicija se izračuna glede na 3 preprosta pravila. Ta pravila so ločenost (angl. Separation), usmerjenost (angl. Allignment) in povezanost (angl. Cohesion). Pravilo ločenosti skrbi, da ptice ne letijo preblizu druga drugi. Pravilo usmerjenosti poskrbi, da ptice v jati letijo v isto smer. Pravilo povezanosti pa poskrbi, da se ptice držijo v jati in ne odletijo vsaka v svojo smer. Stanje vsake ptice je opisano z štirimi komponentami - vektor pozicije (X in Y koordinati) ter vektor smeri oz. hitrosti (X in Y koordinati).

Iz grobe psevdokode 1 algoritma je razvidno, da je osnovna serijska implementacija zelo enostavna. Vsako iteracijo se, glede na trenutno stanje, izračuna novo stanje. Novo stanje za vsako ptico se izračuna le glede na lokalne ptice. To so ptice, ki so znotraj določenega radija oddaljenosti od ptice za katero računamo novo stanje. Iskanje lokalnih ptic ima $O(N)$ časovno zahtevnost, kjer je N enak številu vseh ptic v jati. Računanje novega stanje za določeno ptico pa ima časovno zahtevnost enako $O(M)$. Tu je M enak številu lokalnih ptic, ki pa ni nikoli večje od N tako, da je časovna zahtevnost računanja novega stanja za določeno ptico prav tako $O(N)$. Novo stanje

pa je potrebno izračunati za vse ptice v jati tako, da je časovna zahtevnost celotnega problema enaka $O(N^2)$.

Iz psevdokode 1 se prav tako vidi, da v pomnilniku hranimo le dve tabeli ptic, ki jih vsako iteracijo zamenjamo. Prostorska zahtevnost algorima je torej $O(N)$.

Algorithm 1 Groba psevdo koda serijskega algoritma

```
1: N ← Stevilo ptic
2: trenutnoStanje ← ptice[N]
3: naslednjeStanje ← ptice[N]
4: loop
5:     for i ← 0; i < N; i++ do
6:         lokalnePtice ← poisciLokalnePtice(trenutnoStanje[i], trenutnoStanje);
7:         naslednjeStanje[i] ← novoStanje(trenutnoStanje[i], lokalnePtice);
8:     izrisiStanje(naslednjeStanje);
9:     trenutnoStanje ← naslednjeStanje;
```

Poglavje 2

Serijska implementacija

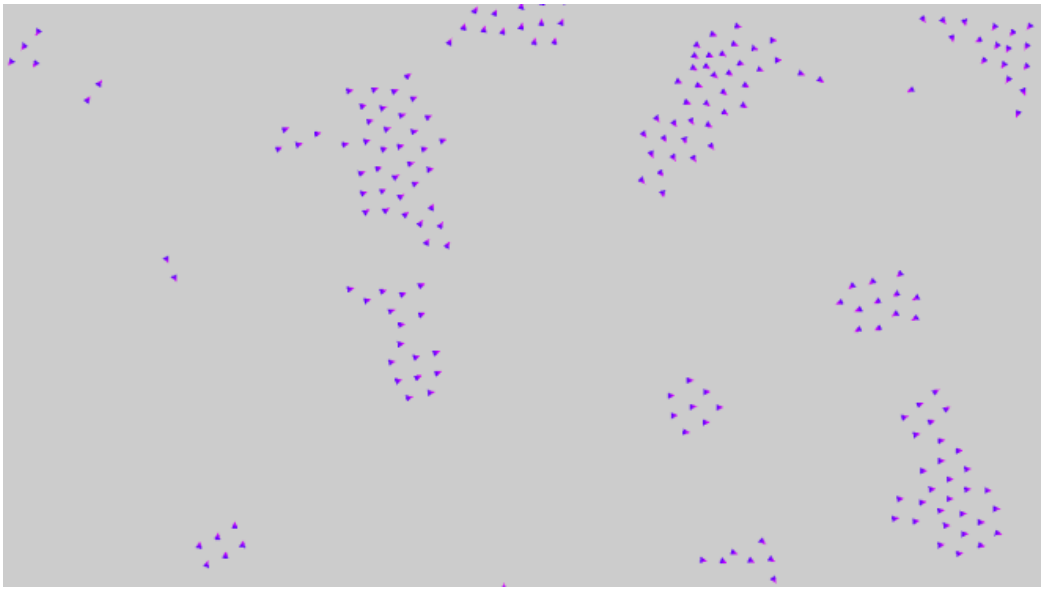
2.1 Osnovna implementacija

Razvoj osnovne implementacije ni predstavljal večjih izzivov. Za razvoj smo uporabili Microsoftov Visual Studio 2015 integrirano razvojno okolje, ki se je izkazalo za zelo koristno, posebno pri odpravljanju hroščev. Nekoliko težav smo imeli le z grafičnim izrisovanjem jate ptic. Saj smo se avtorji tokrat privič srečali z grafično knjižnico OpenGL. Poleg OpenGL smo uporabili še GLFW knižnjico, ki poskrbi za kreiranje grafičnega okna. Končen rezulatat je prikazan na Sliki 2.1. Po približno 10 sekundah leta so se ptice, iz popolnoma naključnih pozicij združile v jate.

Glavna podatkovna struktura programa je *ptica*. *ptica* zajema vse podatke ki so potrebni za opis ene ptice v prostoru. To sta torej dva dvodimenzionalna vektorja *pozicija* in *smer*. Struktura *jata* pa združi vse ptice v tabelo *ptic*.

Serijska implementacija potrebuje kot vhodni podatek le število ptic. Glede na število ptic se ustvari ustrezno velika *jata*.

Ker rezultat vsake iteracije tudi izrišemo na ekran smo se oločili, da bo glavna merska enota število okvirjev na sekundo (ang. Frames per second). Rezultati meritev so tabelirani v Tabela 2.1. Ker smo bili nad rezultati nekoliko razočarani smo se odločili algoritem pohitriti z mrežo.



Slika 2.1: Slika je bila zajeta po približno 10 sekundah prostega leta. Ptice so se združile v jate.

2.2 Implementacija z mrežo

Zaradi relativno poraznih rezultatov smo se odločili, da serijski algoritem pohitrimo. To smo naredili z implementacijo nove podatkovne strukture mreža (ang. Grid). Iz popravljene psevdokode 2 algoritma je razvidno, da se vsako iteracijo ptice najprej razporedi v mrežo na to pa se izračunajo nova stanja ptic. Mreža je ilustrirana na Sliki 2.2. Velikost celice je določena z doložino lokalnosti. To je maksimalna razdalja med pticama, ki še vedno vplivata druga na drugo. Za računanje novih stanj ptic v neki celici so tako potrebne le ptice v isti in sosednjih celicah. Uporabi se 8-kratna sosednost. Tako znatno pohitrimo izvajanje algoritma saj ni več potrebno iskati lokalnih ptic.

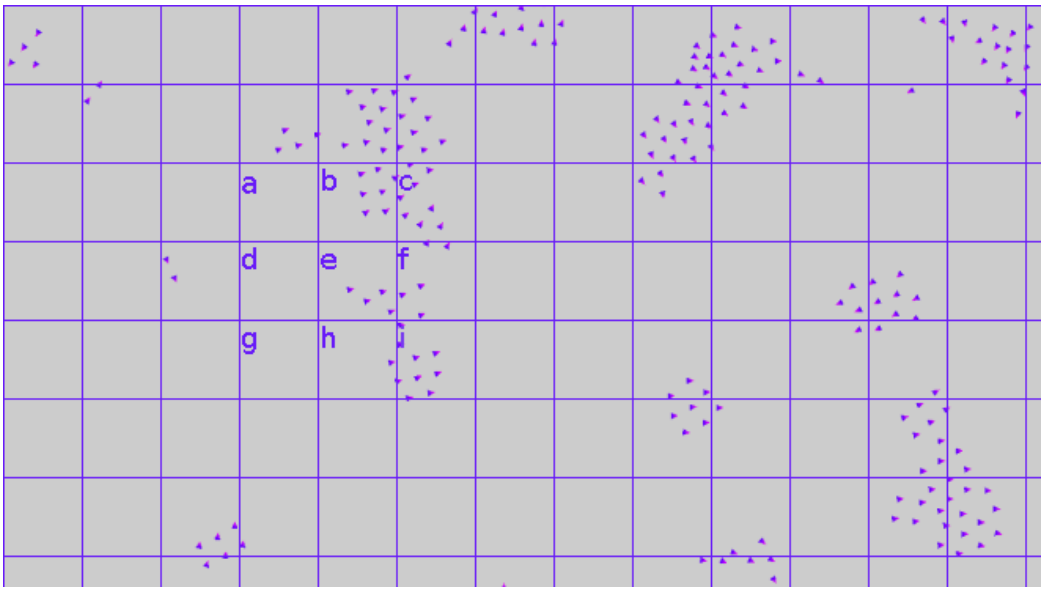
Rezultati meritev so tabelirani v Tabela 2.1. Algoritem z mrežo je opazno hitrejši. Pri 1000 pticah je algoritem z mrežo skoraj 6 krat hitrejši. Meritve so vizualizirane na Slika 2.3

Velikost jate	**Serijski algoritem [FPS]**	**Serijski algoritem z mrežo[FPS]**
100	653,222	1533,090
200	526,485	1347,730
300	390,755	1096,380
400	276,068	922,016
500	194,607	774,890
600	149,196	658,868
700	120,471	537,585
800	100,174	465,814
900	84.070	441,535
1000	71,061	419,116

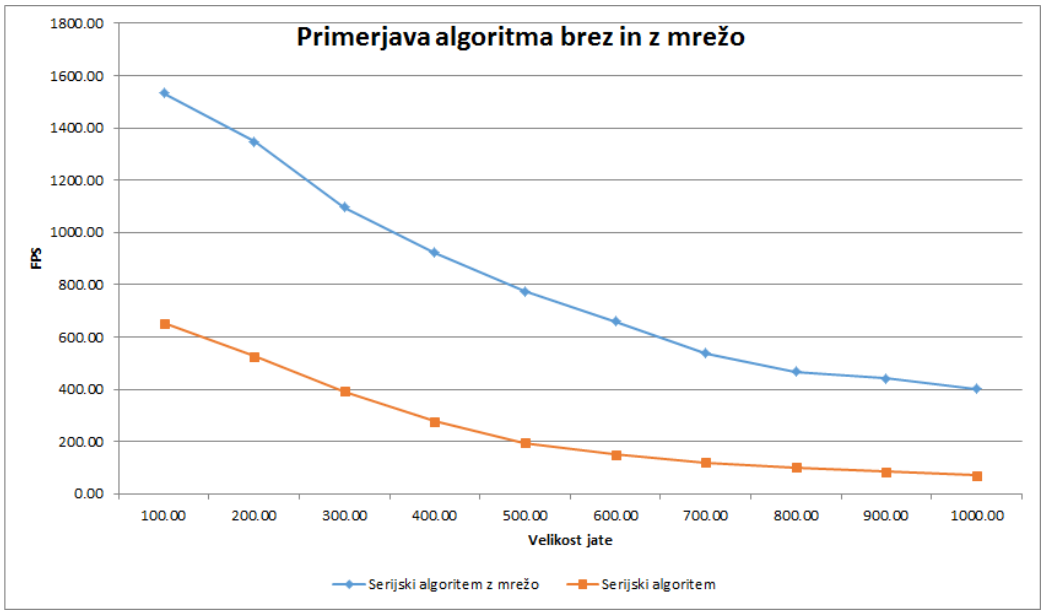
Tabela 2.1: Primerjava števila okvirjev na sekundo (ang. Frames per second) v odvisnosti od velikosti jate za serijski algoritem in serijski algoritem z mrežo

Algorithm 2 Groba psevdo koda serijskega algoritma z uporabo mreže

```
1: N ← Stevilo ptic
2: trenutnoStanje ← ptice[N]
3: naslednjeStanje ← ptice[N]
4: loop
5:     mrezaPtic ← razporediPticeVMrezo(trenutnoStanje);
6:     for i ← 0; i < N; i++ do
7:         naslednjeStanje[i] ← novoStanje(trenutnoStanje[i], mrezaPtic);
8:     izrisiStanje(naslednjeStanje);
9:     trenutnoStanje ← naslednjeStanje;
```



Slika 2.2: Ilustracija mrežne. Za računanje nove pozicije se uporabi 8-kratna sosednjost. To pomeni, da se za izračun novi stanj ptic v celici e uporabijo ptice iz celic $a, b, c, d, e, f, g, h, i$



Slika 2.3: Graf primerja število okvirjev na sekundo za serijski algoritem z in brez mreže v odvisnosti od velikosti jate.

Poglavje 3

Paralelna implementacija z uporabo pThreads

Prva paralelna implementacija algoritma uporablja pThreads za nitenje. Kot izhodišče za implementacijo smo uporabili serijski algoritem z mrežo. Implementacija je grobo opisana v psevdokodi 3. V vsaki iteraciji glavna nit ptice razporedi v mrežo. Nato pa se ustvarijo posamezne niti, ki izračunajo novo stanje jate. Komunikacije med nitmi ni potrebna saj vsaka nit dobi podatke o vseh pticah v jati. Ko niti končajo računanje se pridružijo glavni niti, da ta izriše novo stanje na ekran.

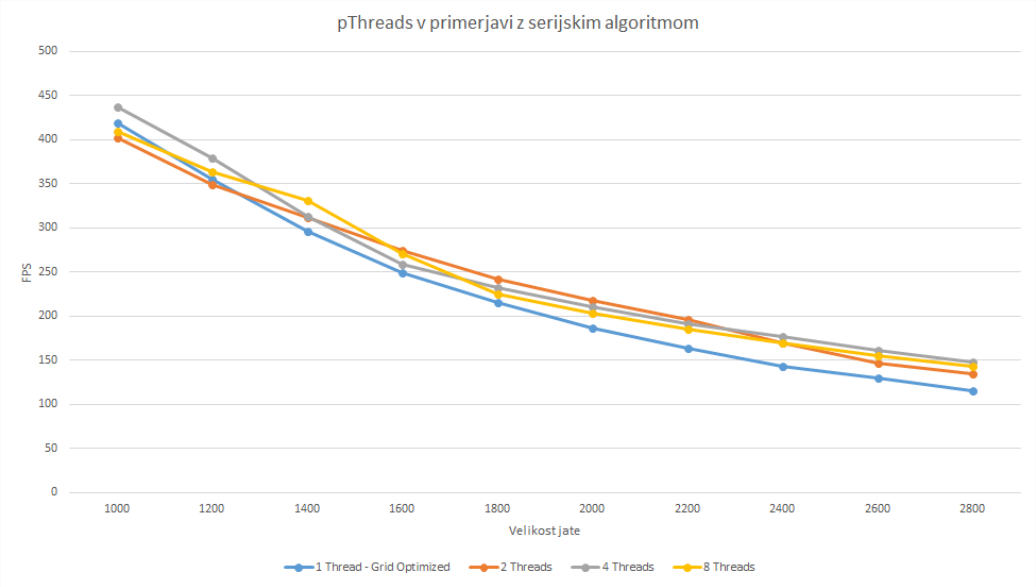
Meritve algoritma so tabelirane v Tabela 3.1. Razberemo lahko, da smo z paralelizacijo dosegli pohitritev. Ta je bila največja pri štirih nitih in se je večala z številom ptic. Učinkovitost se je prav tako večala z število ptic a je bila ta največja pri dveh nitih. Razlog za to je najverjetneje strojna oprema saj ima procesor le 2 jedri. Meritve smo izvajali na prenosnem računalniki z *Intel i7 – 5500U 2.40GHz* procesorjem in *16GB* pomnilnika. Meritve so vizualizirane v grafu na Slika 3.1.

Algorithm 3 Groba psevdo koda paralelnega algoritma z uporabo pThreads

```
1: N ← Število ptic
2: P ← Število niti
3: trenutnoStanje ← ptice[N]
4: naslendjeStanje ← ptice[N]
5: loop
6:     mrezaPtic ← razporediPticeVMrezo(trenutnoStanje);
7:     for i ← 0; i < P; i++ do
8:         ustvariNit(i, N, trenutnoStanje, naslendjeStanje, mrezaPtic);
9:     for i ← 0; i < P; i++ do
10:        pridruziNit(i);
11:    izrisiStanje(naslendjeStanje);
12:    trenutnoStanje ← naslendjeStanje;
```

	Serijski	**pThreads - 2 niti**			**pThreads - 4 niti**			**pThreads - 8 niti**		
Velikost jate	FPS	FPS	S	E	FPS	S	E	FPS	S	E
1000	419,12	401,96	0,96	0,48	437,63	1,04	0,26	409,84	0,98	0,12
1200	354,73	349,46	0,99	0,49	379,77	1,07	0,27	363,85	1,03	0,13
1400	296,02	311,61	1,05	0,53	312,54	1,06	0,26	331,14	1,12	0,14
1600	249,40	274,55	1,10	0,55	258,45	1,04	0,26	270,84	1,09	0,14
1800	216,03	242,16	1,12	0,56	232,21	1,07	0,27	225,42	1,04	0,13
2000	186,09	217,36	1,17	0,58	210,56	1,13	0,28	204,04	1,10	0,14
2200	163,80	195,88	1,20	0,60	191,45	1,17	0,29	184,85	1,13	0,14
2400	143,49	170,23	1,19	0,59	177,42	1,24	0,31	169,70	1,18	0,15
2600	129,57	146,25	1,13	0,56	160,77	1,24	0,31	155,48	1,20	0,15
2800	116,04	134,75	1,16	0,58	147,77	1,27	0,32	143,54	1,24	0,15

Tabela 3.1: Meritve za pThreads paralelni algoritem. FPS predstvlja število okvirjev na sekundo, S predstavlja pohitritev, E pa predstavlja učinkovitos.



Slika 3.1: Graf primerja paralelni pThreads algoritem z serijskim algoritmom v odvisnosti od velikosti jate.

Poglavje 4

Paralelna implementacija z uporabo OpenMP

Naslednja paralelna implementacija uporablja OpenMP za nitenje. Zaradi narave OpenMP-ja je bil ta algoritem najlažje implementirati. Kot izhodišče smo zopet uporabili serijski algoritem z mrežo. Kot je moč razbrati iz psevdokode 4 je bila za implementacijo potrebna le ena pragma. Tu smo poizkusili tudi z paralelizacijo notranje zanke, torej tiste znotraj funkcije novoStanje, ki iterira čez lokalne ptice. Taka implementacija se je žal izkazala za veliko počasnejšo, tako da smo uporabili to z le eno pragmo.

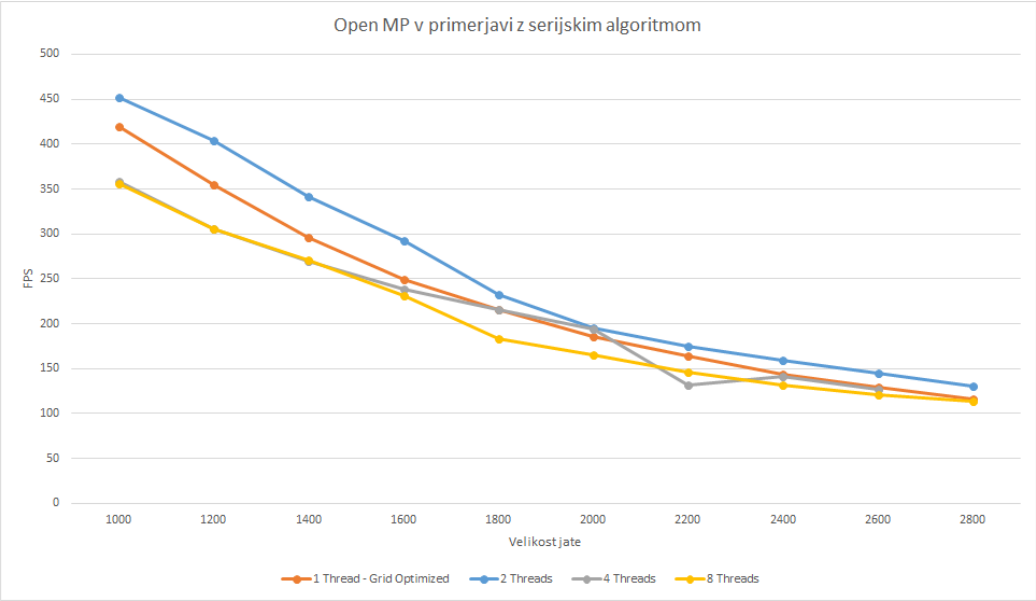
Meritve algoritma so tabelirane v Tabela 4. Tako kot prejšne meritve so bile tudi te izmerjene na prenosnem računalniku z *Intel i7 – 5500U 2.40GHz* procesorjem in *16GB* pomnilnika. Meritve so vizualizirane v grafu na Slika 3.1. Rezultati so nekoliko slabši od pThreads algoritma. Pri osmih nitih nam celo ni uspelo doseči pohitritve - vsaj ne pri izmerjenih velikostih jat. Se pa je pohitritev, tako kot pri pThreads implementaciji, večala z velikostjo jate tako, da bi pohitritev dosegli pri večji jati ptic. Pri dveh in štirih nitih pa je OpenMP popolnoma primerljiv z pThreads implementacijo.

Algorithm 4 Groba psevdo koda paralelnega OpenMP algoritma

```
1: N ← Stevilo ptic
2: trenutnoStanje ← ptice[N]
3: naslendjeStanje ← ptice[N]
4: loop
5:     mrezaPtic ← razporediPticeVMrezo(trenutnoStanje);
6:     #pragma omp parallel for
7:     for i ← 0; i < N; i++ do
8:         naslendjeStanje[i] ← novoStanje(trenutnoStanje[i], mrezaPtic);
9:     izrisiStanje(naslendjeStanje);
10:    trenutnoStanje ← naslendjeStanje;
```

	Serijski	**OpenMP - 2 niti**			**OpenMP - 4 niti**			**OpenMP - 8 niti**		
Velikost jate	FPS	FPS	S	E	FPS	S	E	FPS	S	E
1000	419,116	451,419	1,077	0,539	358,913	0,856	0,214	355,986	0,849	0,106
1200	354,729	404,158	1,139	0,570	306,139	0,863	0,216	306,216	0,863	0,108
1400	296,022	342,132	1,156	0,578	269,892	0,912	0,228	271,292	0,916	0,115
1600	249,400	292,425	1,173	0,586	239,009	0,958	0,240	231,754	0,929	0,116
1800	216,027	232,414	1,076	0,538	215,714	0,999	0,250	183,616	0,850	0,106
2000	186,088	195,283	1,049	0,525	194,283	1,044	0,261	164,767	0,885	0,111
2200	163,803	174,330	1,064	0,532	132,147	0,807	0,202	145,796	0,890	0,111
2400	143,485	158,778	1,107	0,553	141,602	0,987	0,247	131,347	0,915	0,114
2600	129,574	145,168	1,120	0,560	126,673	0,978	0,244	120,304	0,928	0,116
2800	116,038	130,722	1,127	0,563	118,434	1,021	0,255	113,196	0,976	0,122

Tabela 4.1: Meritve za OpenMP paralelni algoritem. FPS predstavlja število okvirjev na sekundo, S predstavlja pohitritev, E pa predstavlja učinkovitost.



Slika 4.1: Graf primerja paralelni OpenMP algoritem z serijskim algoritmom v odvisnosti od velikosti jate.

Poglavje 5

Paralelna implementacija z uporabo OpenCL

Pri serijskem algoritmu smo imeli na voljo 1 nit, ki je bila zadolžena za celotno računanje. Nato smo uporabili knjižnici pThread in OpenMP. Ti dve knjižnici sta nam omogočili ustvarjanje in upravljanje z nitmi. Tako smo imeli na testnih računalnikih lahko 2, 4 in 8 niti, ki so znatno pohitrile čas računanja. Nato pa je sledila še knjižnica OpenCL. OpenCL je knjižnica, ki lahko prevede in požene program na grafični kartici. To je v našem primeru pomenilo, da smo lahko 8 niti zamenjali z 640 delovnimi elementi (nitmi). Seveda se arhitektura grafične kartice zelo razlikuje od arhitekture procesorja in zato procesorske niti ne moremo enačiti z nitmi na grafični kartici. Procesorska arhitektura namreč omogoča izvrševanje ukazov v zaporednem vrstnem redu. Ta arhitektura je v primerjavi z arhitekturo grafične kartice čisto drugačna. Grafična kartica namreč deluje tako, da vsako njeno jedro vzporedno z drugimi jedri izvršuje ukaze.

OpenCL algoritem je bil najtežji za implementacijo saj je bilo potrebno prepisati večino algoritma v ščepec (ang. Kernel), ki teče na grafifični kartici. Na začetku vsakega računanja na grafični kartici, se podatki morajo vedno prenesejo iz glavnega pomninika na pomnilnik grafične kartice. To lahko pri večjem številu ptic zelo upočasni čas računanja. Da bi pohitrili prenašanje

podatkov, smo se odločili, da podatke prenesemo le enkrat. To se zgodi ob prvi iteraciji, ko zaženemo program na grafični kartici. Ko se prva iteracija konča, grafična kartica prenese rezultate v glavni pomnilnik, svoje podatke pa obdrži v svojem pomnilniku. Tako lahko za naslednjo iteracijo uporabi kar svoje stare podatke in jih ne potrebuje ponovno prenašati iz glavnega pomnilnika.

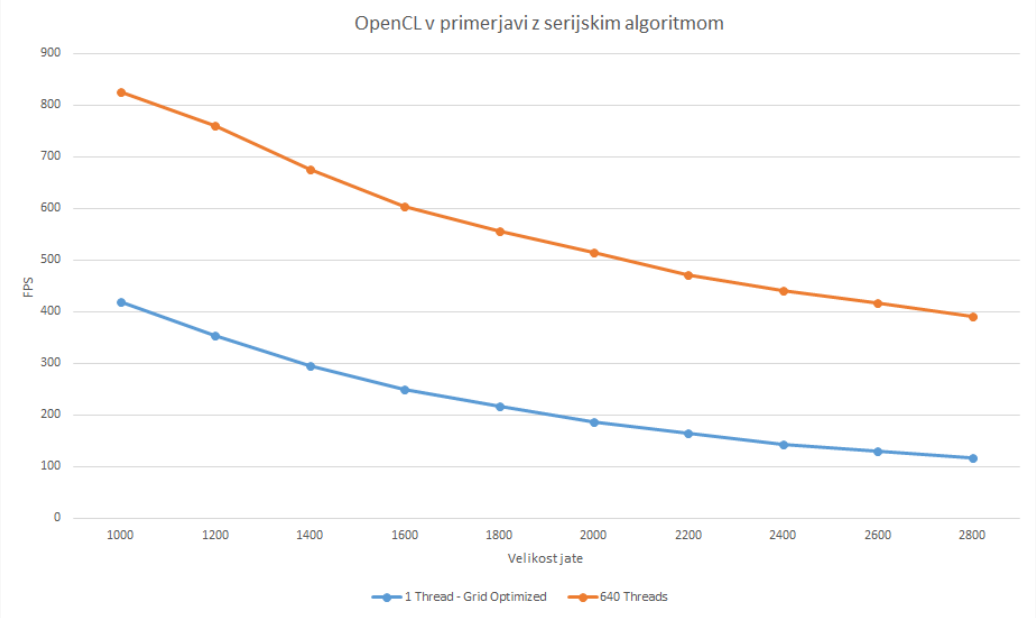
Ker v našem primeru skorajda ne potrebujemo komunikacije med jedri grafične kartice, je to področje kjer se grafična kartica zelo dobro izkaže. Hitrost računanja grafične kartice je bila mnogo hitrejša od računanja na procesorju, kar se še posebno lepo pokaže pri večjem številu ptic. To lahko opazimo v Tabeli 5.1 ali pa na Sliki 5.1. Pri 2800 pticah smo dosegli kar 3,36 kratno pohitritev. Kar je največja pohitritev, ki smo jo uspeli doseči s katerokoli arhitekturo.

Algorithm 5 Groba psevdo koda paralelnega OpenCL algoritma

```
1: N ← Stevilo ptic
2: trenutnoStanje ← ptice[N]
3: naslednjeStanje ← ptice[N]
4: buildKernel();
5: loop
6:     runKernel(trenutnoStanje);
7:     getResults(naslednjeStanje);
8:     izrisiStanje(naslednjeStanje);
```

	Serijski	OpenCL - 640 niti		
Velikost jate	FPS	FPS	S	E
1000	419,1160	826,0000	1,9708	0,0031
1200	354,7290	761,0000	2,1453	0,0034
1400	296,0220	676,0000	2,2836	0,0036
1600	249,4000	605,0000	2,4258	0,0038
1800	216,0270	556,0000	2,5738	0,0040
2000	186,0880	516,0000	2,7729	0,0043
2200	163,8030	471,0000	2,8754	0,0045
2400	143,4850	440,0000	3,0665	0,0048
2600	129,5740	418,0000	3,2260	0,0050
2800	116,0380	390,0000	3,3610	0,0053

Tabela 5.1: Primerjava števila okvirjev na sekundo (ang. Frames per second) v odvisnosti od velikosti jate za serijski algoritem in OpenCL algoritmom



Slika 5.1: Graf primerja število okvirjev na sekundo za serijski algoritem z mrežo in OpenCL algoritmom v odvisnosti od velikosti jate.

Poglavje 6

Paralelna implementacija z uporabo MPI

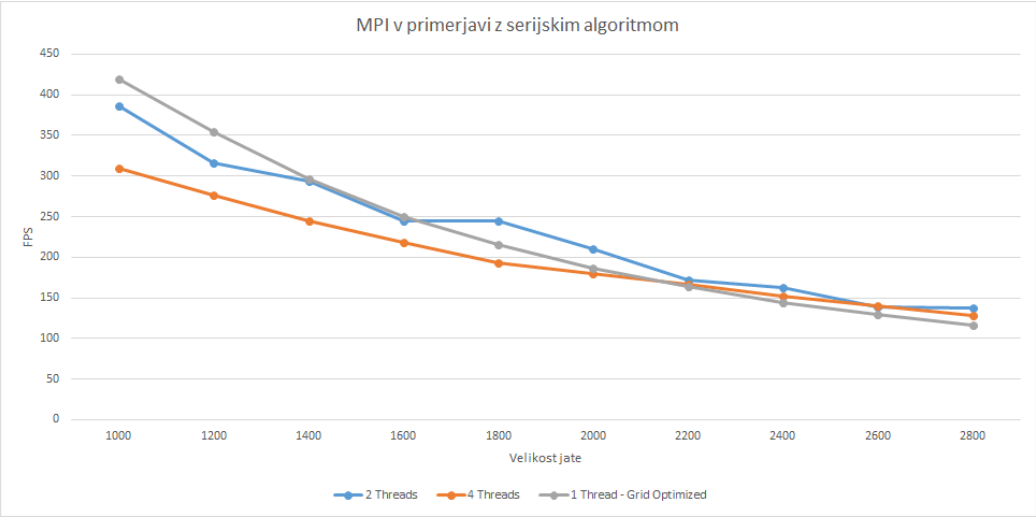
Kot zadnjo tehnologijo za nitenje smo uporabil MPI. Na začetku vsake iteracije ničti procesor pošlje tabelo vseh ptic vsem procesorjem v gruči. Vsak procesor si nato izračuna mrežo ter izračuna nove pozicije za svoje ptice. Svoj del seveda poračuna tudi ničti procesor. Nato vsi procesorji pošljejo tabelo ptic z novimi pozicijami ničtemu procesorju. Pošljejo se le ptice, ki jim je dani procesor izračunal novo pozicijo. Tako nekoliko prihranimo na komunikaciji. Ničti procesor združi posamezne tabele ptic v skupno tabelo. To skupno tabelo ob naslednji iteraciji spet pošlje vsem procesorjem za izračun novih pozicij. Če program zaganjamo lokalno je moč vključiti grafični izris. V tem primeru ničti procesor odpre grafično okno in skrbi za izris ptic vsako iteracijo.

Meritve izvedene na istem osebnem računalniku kot meritve za prejšne tehnologije so tabelirane v Tabela 6. Uspelo nam je doseči pohitritev, a so bile te najslabše od vseh uporabljenih tehnologij. To pripisujemo predvsem večji komunikaciji saj je to prva tehnologija kjer niti ne komunicirajo prek skupnega pomnilnika. Dodatno se nekaj časa izgubi pri pretvarjanju podatkov. Namreč nemoremo pošiljati tabele struktur *ptica*. Tako moramo vsako ptico pretvoriti v tabelo float števil, ki natančno opišejo stanje ptic (v našem

	Serijski	**MPI - 2 niti**			**MPI - 4 niti**		
Velikost jate	FPS	FPS	S	E	FPS	S	E
1000	419,12	386,30	0,92	0,46	309,44	0,74	0,18
1200	354,73	316,19	0,89	0,45	275,97	0,78	0,19
1400	296,02	293,20	0,99	0,50	245,19	0,83	0,21
1600	249,40	245,02	0,98	0,49	218,18	0,87	0,22
1800	216,03	244,74	1,13	0,57	192,51	0,89	0,22
2000	186,09	210,07	1,13	0,56	180,12	0,97	0,24
2200	163,80	171,62	1,05	0,52	166,15	1,01	0,25
2400	143,49	162,31	1,13	0,57	151,85	1,06	0,26
2600	129,57	138,84	1,07	0,54	139,74	1,08	0,27
2800	116,04	137,38	1,18	0,59	128,19	1,10	0,28

Tabela 6.1: Meritve za MPI paralelni algoritem. FPS predstavlja število okvirjev na sekundo, S predstavlja pohitritev, E pa predstavlja učinkovitost.

primeru so potrebne štiri float števila). Na strani prejemnika pa je to tabelo števil potrebno pretvoriti nazaj v tabelo struktur *ptica*.



Slika 6.1: Graf primerja paralelni MPI algoritem z serijskim algoritmom v odvisnosti od velikosti jate.