Univerza na Primorskem

Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Marko Grgurovič

Povzporejanje metahevristik na GPE

Zaključna naloga

Koper, september 2010

Mentor: prof. dr. Andrej Brodnik

Kazalo

Zahvala

Rad bi se zahvalil mentorju za ves trud, ki ga je vložil v pregledovanje pojmov in idej prisotnih v tej nalogi. Zahvalil bi se rad tudi svoji mami, ki mi je pomagala pri pregledovanju in odpravljanju slovničnih napak.

Slovarček

Slovarček vsebuje nekatere pojme, ki se pojavljajo v nalogi. Določene razlage so povzete po [41] in [32].

algoritem (angl. *algorithm*)	Postopek, ki zagotavlja rešitev problema v končnem številu korakov.
CPE (angl. *CPU*)	Centralna procesna enota (angl. *central processing unit*). Del računalnika, ki bere in izvaja ukaze, kateri tvorijo računalniški program. Na voljo ima glavni pomnilnik.
glavni pomnilnik (angl. *main memory*)	Osrednja pomnilniška naprava računalnika z bralno-pisalnim pomnilnikom.
GPE (*angl. GPU*)	Grafična procesna enota (angl. *graphics processing unit*). Posebna strojna oprema, namenjena obdelavi grafike.
jedro (angl. *core*)	Del procesorja, ki bere in izvaja ukaze. Pri enojedrnih procesorskih arhitekturah je to enako CPE.
nit (angl. *thread*)	Najmanjša enota opravil. Več niti lahko izvajamo istočasno, če strojna oprema to podpira.
pomnilnik (angl. *memory*)	Nosilec podatkov za zapisovanje in/ali branje.
prevajalnik (angl. *compiler*)	Računalniški program, ki pretvori izvorno kodo, napisano v določenem programskem jeziku v strojni jezik.
programski jezik (angl. *programming language*)	Umetni jezik, ki se uporablja za pisanje računalniških programov.
programsko jedro (angl. *kernel*)	Del računalniškega programa, ki se vzporedno izvaja na GPE.

računalniški program (angl. *computer program*)	Zaporedje ukazov, ki jih lahko izvede računalnik.
razhroščevalnik (angl. *debugger*)	Računalniški program, ki nam pomaga pri odkrivanju napak v kodi.
razporejevalnik (angl. *scheduler*)	Del operacijskega sistema, ki določa vrstni red izvajanja vzporednih procesov.
uteženi graf (angl. *weighted graph*)	Graf, v katerem ima vsaka povezava določeno težo.
navidezni stroj (angl. *virtual machine*)	Programska oprema, ki se obnaša kot fizični računalnik. Npr. izvaja ukaze tako, kot bi to sicer počel fizični računalnik.

Poglavje 1

Uvod

V zadnjih letih se je višanje frekvenc centralno procesnih enot (CPE) več ali manj ustavilo. Novejši procesorji so večjedrni, kar omogoča pospešek, v kolikor lahko izvajamo programe vzporedno. Vendar CPE niso edina večjedrna strojna oprema, ki je prisotna v današnjih računalnikih. Dandanes računalniki pogosto vsebujejo tudi zelo zmogljive grafične kartice, ki niso nič drugega kot večjedrni procesorji. V zadnjih letih se je področje splošnega računanja na grafičnih karticah precej razvilo, kar prikazujejo okolja kot so Nvidia CUDA in odprti standard OpenCL.

V nalogi si bomo najprej ogledali nekaj klasičnih konceptov povzporejanja ter zelo na kratko povedali še nekaj glede okolij za porazdeljeno računanje, da lahko bolje prikažemo trenutno sliko področja. Nato si bomo bolj podrobno ogledali Nvidino okolje za splošno računanje na grafičnih procesnih enotah (GPE), CUDA. Povedali bomo še nekaj podrobnosti glede strojne opreme in sicer Nvidinih grafičnih kartic. Na kratko bomo še predstavili okolje za računanje na GPE, OpenCL.

Ogledali si bomo nekaj problemov iz kombinatorične optimizacije in sicer: problem trgovskega potnika, najmanjšega pokritja grafa, največje neodvisne množice in pa problem največjega polnega podgrafa. Za potrebe reševanja problemov kombinatorične optimizacije bomo upeljali pojme optimumov ter metahevristične algoritme za iskanje le-teh.

Opisali bomo vrsto metahevristik in sicer: simulirano ohlajanje, metodo navzkrižne entropije, optimizacijo s kolonijo mravelj, iskanje s prepovedmi in evolucijske algoritme. Vsak algoritem bomo najprej predstavili v zaporednem načinu. Nato si bomo ogledali probleme, ki bi nastopili ob povzporejanju z uporabo grafične kartice in podali možne rešitve. Opis vsakega algoritma bomo nato zaključili s pregledom trenutnih načinov povzporejanja, pri čemer bomo izbrali takšne načine povzporejanja, ki čimbolj ustrezajo arhitekturi grafične kartice.

Zaključili bomo z implementacijami treh algoritmov v okolju CUDA in sicer: simulirano ohlajanje (implementacija povzeta po [32]), metoda navzkrižne entropije (implementacija povzeta po [32]) in sistem kolonije mravelj (lastna implementacija). Prikazali bomo tako negativne kot pozitivne rezultate in podali razloge, zakaj se implementacije tako obnašajo.

Poglavje 2

Povzporejanje

Povzporejanje je postopek preoblikovanja programa, da omogoča istočasno izvajanje enega ali več delov. Vzemimo kot primer program, ki je razdeljen na N enako časovno-intenzivnih delov ter ga vzporedno poženemo na N procesorjih: ker se posamezni deli programa izvajajo istočasno, lahko dobimo do N-kratno pohitritev! Žal je v realnih programih takšna pospešitev redko mogoča. Celo več, zaradi cene dodatnih postopkov, ki zagotavljajo izključujočnost dostopa na kritično območje, se lahko stvarni čas izvajanja celo podaljša. Razlog za to je pogosto zaporedna narava algoritmov, ki bi jih želeli povzporejati. Take programe je pogosto težko razdeliti po delih na takšen način, da se problemi kritičnih območij ne bi pojavili. Pri nekaterih programih se pojavi tudi problem potrebe po usklajevanju med vzporedno izvajajočimi se deli, kar dodatno oteži postopek povzporejanja in pogosto tudi podaljša izvajalni čas.

Dandanes je področje povzporejanja pomembnejše kot kadarkoli prej, saj se frekvence procesorjev ne povečujejo več tako, kot so se včasih, da bi omogočale pohitritev programov [36]. Po drugi strani, imajo procesorji več jeder in prav ta jedra nam omogočajo istočasno izvajanje več delov programske kode. Kot bomo videli v kasnejših poglavjih te naloge, imamo na voljo za splošno računanje izredno moč GPE, ki je večjedrna strojna oprema narejena prav za masovno povzporejanje. Vendar dovolj dobro povzporejanje tipičnih zaporednih algoritmov ni vedno možno. Prav to bo morda v naslednjih letih povzročilo intenzivno delo na področju že po naravi vzporednih algoritmov za današnje probleme, saj bo potrebna strojna oprema takorekoč prisotna v vsakem računalniku [40].

Problemov pri povzporejanju je veliko. Ogledali si bomo nekaj konceptov, ki nam omogočajo boljše razumevanje (in reševanje) tegob povzporejanja. Nato bomo opisali dve tehniki povzporejanja, ki pravzaprav predstavljata dve nasprotujoči-si skrajnosti. V realnosti se večina povzporejanja izvede kot kompromis med tehnikama, odvisno od problema samega.

2.1 Kritična območja

Recimo, da smo program, ki je sestavljen iz enega stavka $X := X + 1$ razdelili na N delov in ga želimo pognati na N procesorjih. Pojavi se problem, kjer mora več procesov obdelati iste podatke. Problem se pojavi zaradi načina branja in pisanja v pomnilnik. Če želi eden izmed procesov povečati spremenljivko za 1, ki se nahaja na pomnilniški lokaciji X mora

najprej naložiti podatke iz lokacije X, nato povečati za 1, in podatke ponovno naložiti nazaj na lokacijo X. Taki operaciji pravimo, da ni atomarna. V kolikor je operacija atomarna, se izvede samo en korak (omenjeni primer sestoji iz treh korakov). Če imamo samo en proces, ni konfliktov in se postopek izvede brez napak. Zgodba je drugačna, če imamo dva procesa A in B, ki želita hkrati povečati isto spremenljivko. Ker nimamo nadzora nad tem, kdaj se bo izvedla katera operacija, se lahko zgodi sledeče zaporedje atomičnih operacij:

1. Proces A prebere podatek iz lokacije X (trenutna vrednost $X = 0$).
2. Proces A poveča svojo kopijo podatka za 1 (trenutna vrednost $X = 0$).
3. Proces B prebere podatek iz lokacije X (trenutna vrednost $X = 0$).
4. Proces A naloži povečan podatek na lokacijo X (trenutna vrednost $X = 1$).
5. Proces B poveča svojo kopijo podatka za 1 (trenutna vrednost $X = 1$).
6. Proces B naloži povečan podatek na lokacijo X (trenutna vrednost X je še vedno 1)

Izkaže se, da lahko opisani problem posplošimo celo na N procesov in edino, kar lahko zagotovimo je, da bo vrednost na lokaciji vsaj 1. Programu $X := X + 1$ pravimo kritično območje (ang. *critical section*). Probleme take narave rešujemo z uporabo metod usklajevanja, ki si jih bomo ogledali v naslednjem podpoglavju. Specifičen problem, ki je opisan zgoraj, lahko rešujemo tudi z atomarnimi ukazi za istočasno branje in pisanje na pomnilnik, če jih arhitektura podpira.

2.1.1 Metode usklajevanja

Metod usklajevanja za vzporedne algoritme je veliko: semaforji, pregrade, monitorji, mutexi, razni algoritmi kot so Dekkerjev, Petersonov, strojna implementacija ukazov kot so: preveri in nastavi, zamenjaj, ipd. Mi se bomo omejili na tiste metode, ki jih lahko uporabljamo tudi na GPE.

Na GPE lahko uporabljamo samo metodo usklajevanja tipa pregrada. Ostale metode usklajevanja v tej nalogi ne bodo opisane.

Pregrada

Metoda usklajevanja tipa pregrada (ang. *barrier*) povzroči zaustavitev izvajanja procesa ob prihodu do pregrade dokler do pregrade ne prispejo še vsi ostali procesi. Vzporedno izvajanje procesov se nadaljuje šele, ko do pregrade prispejo vsi procesi.

2.2 Podatkovno povzporejanje

Pri podatkovnem povzporejanju (ang. *data-based parallelism*) razdelimo vhodne podatke na (tipično) disjunktne dele, na katerih poženemo več vzporednih instanc algoritma. Pri po-

datkovnem povzporejanju ponavadi izločimo (ali vsaj zmanjšamo) potrebo po usklajevanju s porazdelitvijo podatkov na čimbolj neodvisne dele.

Preprost primer podatkovnega povzporejanja je seštevanje dveh vektorjev števil. Imamo dva poljubna vektorja celih števil $v_1, v_2, v_r \in \mathbb{N}^n$, kjer sta v_1 in v_2 vhodna vektorja in v_r izhodni rezultat vektorske operacije $v_r = v_1 + v_2$, ter N procesov. Vsak proces ima enolično identifikacijsko stevilko $i \in (1, ...N)$ in dobi kot vhodne podatke $v_1[(i-1) \cdot k...min(i \cdot k-1, n)]$ in $v_2[(i-1) \cdot k...min(i \cdot k-1, n)]$, kjer $k = \lceil n/N \rceil$, nato ju sešteje in rezultat shrani v $v_r[(i-1) \cdot k...min(i \cdot k-1, n)]$.

Zgornji problem smo z uporabo podatkovnega povzporejanja rešili brez kakršnekoli uporabe metod usklajevanja. Morda je še bolj pomembno, da smo problem tako razdelili, da je bila kakršnakoli komunikacija med procesi nepotrebna in so se vzporedni procesi lahko izvajali povsem neodvisno. Komunikacija je tipično draga — še posebej, če se procesi izvajajo na fizično ločenih enotah.

Kot bomo videli v kasnejših poglavjih, je prav tak način povzporejanja idealen za računanje na GPE, kjer je lahko komunikacija zelo draga. Na žalost veliko problemov ni tako preprosto razdeliti na popolnoma disjunktne dele. Včasih se izkaže, da bi bila zadovoljiva tudi delitev, ki je morda slabša za zaporedno računanje. V takšem primeru bi upali, da bi jo lahko z zmogljivostjo masovnega vzporednega izvajanja – kot to nudi GPE – lahko rešili v skrajno krajšem času, kot to lahko počne optimalen zaporedni algoritem na CPE.

2.3 Povzporejanje opravil

Povzporejanje opravil (ang. *task-based parallelism*) je pogosto idealen način povzporejanja za izvajanje na večjedrnih CPE, ker je tipična delitev na opravila veliko bolj groba kot pri podatkovni delitvi, kar ustreza hitrejšemu dostopu do pomnilnika (v primerjavi z GPE), ki ga premorejo večjedrni CPE. Najpreprostejši primer povzporejanja opravil je neodvisno poganjanje več instanc algoritmov, kjer lahko dodelimo vsakemu jedru CPE eno ali več instanc.

Strojna oprema GPE je optimizirana za čim boljše razporejanje niti ob dolgotrajnih strojnih operacijah (npr. branje iz pomnilnika), kar pomeni, da grobo povzporejanje, kot je tipično za povzporejanje opravil, ne nudi dovolj velikega števila niti, da bi to zmogljivost dobro izkoristile. Posledično je izvajalni čas takih algoritmov neprimerljivo slabši od CPE.

2.4 MPI

MPI (ang. *Message Passing Interface*) je standard za komunikacijo med procesi vzporednih programov, ki se izvajajo na ločenih procesorjih. Glavne lastnosti MPI [34] so med drugim:

- Definicija vmesnika, ki ga lahko implementiramo na različnih okoljih brez velikih sprememb komunikacijske in sistemske programske opreme.
- Zmožnost implementacij, ki se lahko uporabljajo v heterogenih okoljih.
- Definicija zanesljivega komunikacijskega vmesnika. Uporabnik naj se ne ukvarja s komunikacijskimi napakami.

- Semantika naj bo neodvisna od jezika implementacije.
- Zmožnost uporabe jezikov *C* in *Fortran 77* za dostopanje do vmesnika.
- Učinkovita komunikacija med procesi.

Komunikacija med procesi je v MPI zelo draga, zato ponavadi omejimo količino podatkov, ki si jih procesi izmenjujejo. Če je količina podatkov prevelika in je ne moremo smiselno zmanjšati, se lahko zgodi, da je čas komunikacije večji kot čas, ki ga porabimo za računanje.

2.4.1 Način komunikacije

Ker MPI deluje po principu izmenjave sporočil med procesi, je delitev na komunikacijske kanale, ki jih imenujemo komunikatorji (ang. *communicator*) naravna. Vsak proces lahko posluša na enem ali več komunikatorjev. Pošiljanje sporočil se tako izvaja znotraj komunikatorjev (npr. en proces pošlje sporočilo vsem, ali vsak vsakemu posebej), katerim pripada določeno število procesov. To izvajamo s funkcijo ***MPI_Send***. Primer uporabe te funkcije v programskem jeziku C je podan v algoritmu 1.

Algoritem 1 Pošiljanje sporočil procesom v MPI

// pošlji polje ,,podatki" dolžine 5, tipa int, po komunikatorju ,,comm"
$MPI\_Send(podatki, 5, MPI\_INT, 0, 0, comm)$;

2.4.2 Usklajevanje

Tako kot pri klasičnem povzporejanju, se tudi pri MPI pojavijo problemi usklajevanja, vendar so tukaj še toliko bolj časovno potratni, saj so vzporedne instance pogosto fizično ločene. Za reševanje teh problemov MPI vključuje veliko funkcij, ki omogočajo razpošiljanje podatkov po mreži, pri čemer takšne funkcije zagotavljajo, da bo npr. nek proces poslal podatek vsem ostalim procesom ter, da bo sam prejel podatek od vseh ostalih procesov.

Te funkcije nam omogočajo realizacijo poljubnih shem usklajevanja, vendar so le-te pogosto v MPI že implemenitrane in nam tako olajšajo delo. MPI podpira tudi že prej omenjeno metodo usklajevanja tipa pregrada in sicer v obliki funkcije ***MPI_Barrier***, pri čemer podamo še komunikator. Ob klicu funkcije se uskladijo vsi procesi v podanem komunikatorju. V kolikor hočemo izvesti globalno uskladitev, podamo kot argument *MPI_COM_WORLD*.

2.5 PVM

PVM (ang. *Parallel Virtual Machine*) je programsko okolje, ki omogoča uporabo heterogene skupine računalnikov za vzporedno računanje. Ideja PVM je, da več fizično oddaljenih računalnikov predstavimo kot en sam vzporedni stroj (virtualizacija). Glavne naloge PVM [35] so med drugim:

- Nastavljiva količina vzporednih strojev. Tako eno kot večjedrske arhitekture naj bodo lahko del vzporednih strojev (lahko tudi mešano).
- Zmožnost dostopa do strojne opreme posameznih vzporednih strojev, vendar naj bo možno tudi ignoriranje tega vidika.
- Računanje naj poteka po procesih: enota povzporejanja je opravilo (ang. *task*). Več opravil se lahko izvaja na enem procesorju.
- Podpora za heterogenost sistemov, ki so predmet virtualizacije.
- Podpora za večprocesorske arhitekture, kjer naj PVM uporablja obstoječo strojno implementacijo za pošiljanje sporočil.
- Model pošiljanja sporočil — procesi naj med seboj komunicirajo s pošiljanjem sporočil eden drugemu.

PVM pogosto primerjamo z implementacijami MPI, saj programski opremi rešujeta podobne probleme vzporednega izvajanja. Vredno je omeniti, da je PVM starejši od MPI. Za razliko od MPI, ki je samo standard, PVM ni standard in obstaja le v obliki konkretne implementacije (za razliko od mnogih implementacij MPI standarda). Pomanjkanje standarda za povzporejanje z izmenjavo sporočil je bil pravzaprav eden izmed razlogov za razvoj MPI.

2.5.1 Način komunikacije

Podobno kot to počne MPI, tudi PVM omogoča komunikacijo med procesi v obliki sporočil. Kadar želi dolo"en proces poslati sporočilo drugemu procesu, to počne s klicem funkcije ***pvm_send*** oz. če želi sporočilo poslati več procesom hkrati pa s klicem funkcije ***pvm_mcast***. Kot parameter poda identifikatorje procesov, katerim je sporočilo namenjeno. Primer za ***pvm_send*** v programskem jeziku C je podan v algoritmu 2.

Algoritem 2 Pošiljanje sporočil procesom v PVM

```
pvm_initsend(PvmDataDefault); // inicializiraj
pvm_pkint(podatki, 5, 1); // pripravi polje ,,podatki" tipa int, dolžine 5
pvm_send(tid, 0); // pošlji polje ,,podatki" procesu ,,tid"
```

2.5.2 Usklajevanje

PVM vključuje zmožnost usklajevanja z uporabo metode tipa pregrada, kot to omogoča MPI in tako, kot je to tudi omogočeno v okoljih za računanje na GPE (kot si bomo ogledali kasneje). Klic funkcije ***pvm_barrier*** povzroči postavitev pregrade na točki v programu, kjer je poklicana.

Poglavje 3

Arhitektura GPE in CUDA

Compute Unified Device Architecture (CUDA) je okolje, ki ga razvija Nvidia za splošno uporabo grafičnih procesnih enot (GPE)[1]. CUDA deluje izključno na Nvidia grafičnih karticah, kjer omogoča vzporedno izvajanje aplikacij in sicer v obliki programskih jeder (ang. *kernel*). Programsko jedro je pravzaprav funkcija, ki jo kliče glavni program, ki se izvaja vzporedno na enem ali več procesorjih GPE. Omeniti moramo, da je Aprila 2010 Nvidia izdala serijo grafičnih kartic *GeForce 400*, poznano tudi pod imenom Fermi [39]. Ta serija je še nova in zaradi tega v tej nalogi ne bo opisana.

Vsaka instanca programskega jedra ima definiran enoličen indeks, katerega se uporablja za krmiljenje programa in dostopa do pomnilnika. Programskih jeder je pravzaprav lahko več, kot je število samih procesorjev na kartici, pri čemer se bodo čakajoča programska jedra izvajala, kadar procesor čaka, kot npr. pri dostopu do pomnilnika. Prenos podatkov iz glavnega pomnilnika v pomnilnik grafične kartice (ter obratno, ko beremo rezultate), je ena izmed časovno najbolj zahtevnih operacij, zato ga običajno opravljamo pred zagonom programskega jedra, ko naložimo vhodne podatke ter na koncu, ko beremo končne rezultate, ki jih vrnejo programska jedra.

Če poganjamo CUDA aplikacije na operacijskem sistemu Windows, ali na Unix operacijskih sistemih (Linux, FreeBSD), kjer imamo vključen X strežnik, je dostop do GPE omejen na približno 5 sekund — prekoračitev te meje povzroči sesutje vseh programskih jeder, ki se trenutno izvajajo na GPE. V kolikor je problem X strežnik, ga lahko začasno izključimo s preklopom na terminalni način dela. Problem lahko tudi rešimo z uvedbo dodatne grafične kartice, ki je operacijski sistem ne uporablja za izrisovanje grafičnega vmesnika. Drugi način je, da program, ki ga želimo izvajati na GPE, razdelimo na rezine tako, da se ne bo nobena od rezin izvajala dlje kot 5 sekund. Na GPE tako izvajamo npr. samo eno iteracijo, ter ob končani iteraciji ponovno poženemo programska jedra. Pri tem ne sprostimo dodeljenega pomnilnika na GPE in ga lahko programska jedra ponovno uporabijo, ko jih poženemo naslednjič. Vendar to ni dobra rešitev, saj je izvajalni čas odvisen od hitrosti GPE ter vhodnih podatkov in zato pogosto ne moremo zagotoviti pravočasen zaključek programskih jeder.

3.1 Arhitektura GPE

Glavni elementi GPE so multiprocesorji. Vsak multiprocesor sestoji iz osmih procesorjev. Multiprocesor ima vlogo razporejevalnika, ki dodeljuje niti svojim procesorjem. SIMT (ang. *Single-instruction multiple-thread*) narava procesorjev dovoli, da izvajajo več niti hkrati. Izvajanje več niti hkrati na enem procesorju je možno, kadar v programskem jedru ni vejitev, t.j. vse niti izvajajo enako kodo. V kolikor pride do vejitev, mora procesor izvajati niti zaporedno, dokler se ponovno ne uskladi tok izvajanja — tu ne mislimo na metode usklajevanja, temveč na dejanski potek programskega jedra. Procesor lahko izvaja 32 niti istočasno. Zaradi tega se bodo niti izvedle hitreje, če pišemo kodo z minimalnim številom vejitev, vendar je to pogosto zelo težko.

Tabela 3.1, povzeta po [1] prikazuje število multiprocesorjev nekaterih CUDA-zmogljivih Nvidinih GPE. Tabela ne vključuje Fermi arhitekture.

GPE	**Št. multiprocesorjev**
GeForce GTX 295	2x30
GeForce GTX 285, GTX 280	30
GeForce GTX 260	24
GeForce 9800 GX2	2x16
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GeForce 8800 Ultra, 8800 GTX	16
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	14
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT, GeForce 8800 GTS	12
GeForce 9600 GT, 8800M GTS, 9800M GTS	8
GeForce 9700M GT	6
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	4
GeForce G100, 8500 GT, 8400 GS, 8400M GT	2

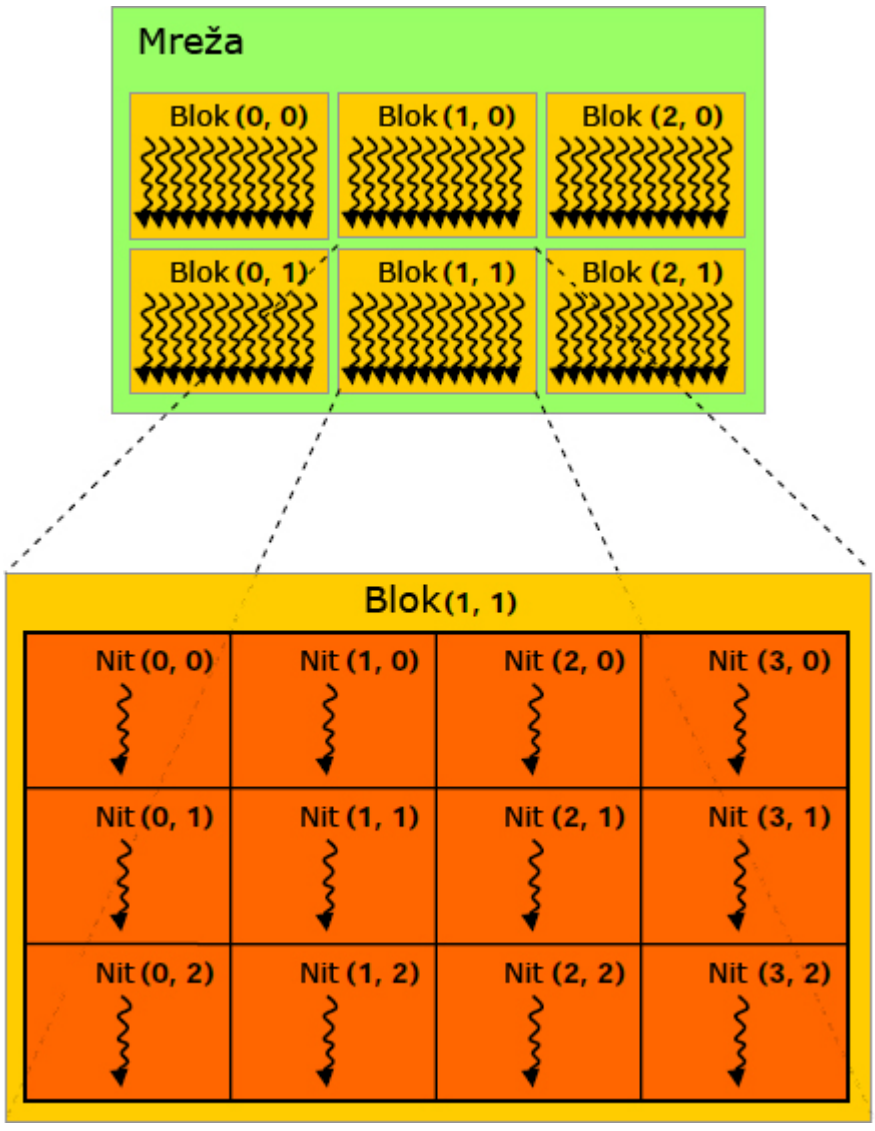
Tabela 3.1: Pri pisanju te naloge je bila uporabljena grafična kartica **Nvidia GeForce 9600 GT**.

3.2 Organizacija procesov

Vsaka nit pripada nekemu bloku. Blok je skupek niti, ki je dodeljen posamezemu multiprocesorju. Skupno število niti v posameznem bloku je lahko največ 512. Multiprocesor ima lahko dodeljenih tudi več blokov, vendar dva multiprocesorja ne moreta imeti dodeljen isti blok. Blokov je lahko več. Vsi bloki skupaj tvorijo mrežo.

Običajno dodelimo vzporednim procesom skalarni indeks, ki jih razlikuje med seboj in na podlagi tega indeksa vemo, katere podatke moramo obdelati in kateri potek izvajanja naj izberemo. Tak način organizacije je enak enoličnemu identifikatorju procesa pri MPI.

CUDA vpelje nekoliko drugačno organizacijo procesov: namesto skalarnega indeksa vpelje indeks, ki je sestavljen iz več komponent. Prva komponenta je lokacija bloka v mreži; ta komponenta ima dve dimenziji. Druga komponenta je lokacija niti v bloku; ta komponenta ima tri dimenzije. V kolikor ga potrebujemo, lahko enoličen skalarni indeks izračunamo na podlagi teh komponent. Taka organizacija se pravzaprav izkaže za naravno, kar se zadeva GPE, kot bomo videli kasneje pri pomnilniku in usklajevanju. Slika 3.1 prikazuje primer takega načina organizacije.



Slika 3.1: Organizacija procesov na mreži dimenzij (3, 2) in blokov dimenzij (4, 3, 1). Slika povzeta po [1].

Vzemimo kot osnovni primer organizacijo, določeno s samo eno dimenzijo. Uporabljeni podatki so:

- $(x, y) = (20, 1)$ dimenziji mreže (število blokov: $x * y = 20$)
- $(x, y, z) = (5, 1, 1)$ dimenzije bloka (število niti na blok: $x * y * z = 5$)
- skupno število niti: $5 * 20 = 100$

Organizacija, za katero se odločimo, je načeloma odvisna od problema, ki ga rešujemo. Pomniti moramo, da je število niti na posamezen blok omejeno na 512. To pomeni, da so velikosti blokov $(512, 1, 1), (256, 2, 1), (16, 16, 2)$ sprejemljive, vendar, npr. velikost bloka $(64, 16, 1)$ ni, saj bi bilo število niti v tem bloku 1024. Dobra stran take organizacije je, da lahko pišemo programe, ki so neodvisni od strojnih zmogljivosti sistema, na katerem jih poganjamo. Tako lahko program požene večje število blokov na GPE, kjer je to mogoče ter manjše število tam, kjer je strojna oprema manj zmogljiva[3]. To lahko namreč počnemo v času teka programa – ko poženemo programsko jedro – tako, da definiramo velikost mreže na podlagi zmogljivost GPE, kjer poganjamo program. GPE ima že vgrajene funkcije, ki nam vrnejo zmogljivosti (te lahko kličemo v programu) in je tako celoten postopek lahko skrit uporabniku.

3.3 Struktura pomnilnika

Pomnilnik je razdeljen na šest nivojev [1, 2]:

- registri
- deljeni pomnilnik
- globalni pomnilnik
- nespremenljivi pomnilnik
- strukturni pomnilnik
- lokalni pomnilnik

3.3.1 Registri

Registri predstavljajo najhitrejši pomnilnik, ki je na voljo. Vsak multiprocesor ima na voljo 8192 registrov. Lokalne spremenljivke se privzeto nahajajo v registrih. Če je prostor, ki ga zaseda spremenljivka v registrih prevelik ali neznan v času prevajanja, bo prestavljena v lokalni pomnilnik. Registri se nahajajo na čipu. Spremenljivke, ki se nahajajo na registrih, imajo življensko dobo niti. Do registrov lahko dostopa samo nit, ki si jih lasti.

3.3.2 Deljeni pomnilnik

Deljeni pomnilnik predstavlja potencialno tako hiter pomnilnik, kot so registri. V primeru konfliktov je počasnejši, npr. če bere ali piše več niti na isto lokacijo. Velikost je 16 KB na multiprocesor. Nahaja se na čipu in ima življensko dobo bloka. Do deljenega pomnilnika lahko dostopajo vse niti v posameznem bloku.

3.3.3 Globalni pomnilnik

Globalni pomnilnik je najpočasnejši izmed pomnilnikov GPE. Do globalnega pomnilnika lahko dostopajo vse niti iz vseh blokov. Globalni pomnilnik je najpogostejši razlog za prekomerno upočasnitev aplikacije (kot bomo videli, spada sem tudi lokalni pomnilnik), saj je čas dostopa lahko do 100x počasnejši kot do deljenega pomnilnika. Ima življensko dobo aplikacije in se ne nahaja na čipu.

3.3.4 Nespremenljivi pomnilnik

Nespremenljivi pomnilnik se ne nahaja na čipu, vendar ima GPE vgrajen predpomnilnik, ki se nahaja na čipu in je zato običajno veliko hitrejši od globalnega pomnilnika. Ta pomnilnik je strojno optimiziran za masovno branje. Podatke v nespremenljivi pomnilnik lahko naloži samo CPE in jih je nemogoče kasneje spreminjati na GPE. Velikost nespremenljivega pomnilnika je 64 KB, velikost predpomnilnika na multiprocesorju pa 8 KB. Ima življensko dobo celotne aplikacije, in do njega lahko dostopajo vse niti v vseh blokih.

3.3.5 Strukturni pomnilnik

Strukturni pomnilnik ima na voljo predpomnilnik, ki je omejen na velikost 6 ali 8 KB na multiprocesor. Ta tip pomnilnika je predviden za polja podatkov. Pred zagonom programskega jedra morajo biti znani nekateri parametri, ki definirajo podatke, katere hrani določena spremenljivka v strukturnem pomnilniku. Ti paramentri so:

- Tip elementov: lahko ali *int* ali $float$ oz. vektorski tipi teh dveh tipov (*int*, $int2$, $int4$ oz. $float$, $float2$, $float4$)
- Dimenzija polja: lahko 1, 2 ali 3-dimenzijsko polje. Če parameter izpustimo, je privzeto 1.
- Tip branja: **cudaReadModeElementType** ali **cudaReadModeNormalizedFloat**. Če parameter izpustimo, je privzeto **cudaReadModeElementType** in je to navaden način branja. V nasprotnem primeru (t.j. **cudaReadModeNormalizedFloat**), če je tip elementa *int*, je ob branju vrnjena vrednost elementa v intervalu [0.0, 1.0] za nepredznačene tipe in [-1.0, 1.0] za predznačene tipe.

Parametri definirani zgoraj so konstantni in se ne smejo spreminjati v času izvajanja programskega jedra. Ta tip pomnilnika lahko nudi nekolikošno pospešitev za aplikacije, ki dostopajo do podatkov naključno. Strukturni pomnilnik se fizično nahaja v globalnem pomnilniku in zanj veljajo enake lastnosti, kot za globalni pomnilnik.

3.3.6 Lokalni pomnilnik

Lokalni pomnilnik ne obstaja na strojnem nivoju. Dejansko gre za abstrakcijo globalnega pomnilnika, ki ni dostopen vsem procesom in se obnaša enako kot registri, vendar z enako velikim časovnim zamikom ob branju ter pisanju kot globalni pomnilnik. Velikost je največ 16 KB na nit.

3.4 Jezik C for CUDA

Za pisanje programskih jeder se uporablja jezik *C for CUDA*, ki je okrnjena različica programskega jezika C. Slednji je bil uporabljen tudi pri implementaciji vzporednih različic algoritmov omenjenih v tej nalogi. *C for CUDA* vključuje samo nekatere standardne knjižnice. Dinamično zaseganje pomnilnika ni možno in je pri pisanju programov potrebno biti pozoren na omejeno velikost pomnilnika, ki ga imamo na voljo. Če potrebujemo veliko prostora, je potrebna uporaba globalnega pomnilnika, ki ga ponavadi rezervira glavni program in ga ob klicu programskega jedra poda v obliki reference, vendar je ta pomnilnik neprimerljivo počasnejši.

C for CUDA vključuje nove predpone, ki jih podamo ob deklaraciji funkcije in določajo ali se lahko funkcijo kliče iz GPE ali iz CPE. Te so:

- __**global**__ : rezervirano za funkcije, ki imajo vlogo programskega jedra. Kličemo jo lahko samo iz CPE, vendar se izvede na GPE.
- __**device**__ : funkcijo lahko kličemo samo iz GPE. Izvaja se na GPE.
- __**host**__ : funkcijo lahko kličemo samo iz CPE. Izvaja se na CPE.

Če izpustimo predpono pri deklaraciji funkcije, je predpona privzeto __**host**__. Možno je kombinirati predponi __**host**__ in __**device**__, pri čemer bo funkcija prevedena tako za CPE, kot tudi za GPE (vendar sta to dve različni funkciji). Programsko jedro deklariramo kot funkcijo, ki ima predpono __**global**__. Programskih jeder je lahko več, vendar lahko na enem GPE teče istočasno samo eno.

Funkcije, ki imajo predpono __**global**__ ali __**device**__ ne podpirajo rekurzije, ne smejo deklarirati statičnih spremenljivk v svojih telesih in ne smejo imeti variabilnega števila argumentov. Reference na funkcije, ki imajo predpono __**device**__ niso možne. Funkcije s predpono __**global**__ morajo vrniti tip *void*.

Jezik vključuje tudi nove predpone, ki jih podamo ob deklaracij spremenljivk in določajo kateri pomnilnik bo uporabljen za hrambo podatkov. Te so:

- __**device**__ : podatki se nahajajo v globalnem pomnilniku.
- __**shared**__ : podatki se nahajajo na deljenem pomnilniku.
- __**constant**__ : podatki se nahajajo na nespremenljivem pomnilniku.

Predpono __**device**__ lahko združimo z eno od ostalih dveh predpon, kar določi na katerem pomnilniku naj se podatek nahaja — pravzaprav je to enako kot, da namesto predpone __**device**__ uporabimo eno od drugih dveh predpon. V primeru, da tega ne storimo, se podatki privzeto nahajajo v globalnem pomnilniku.

3.4.1 Načini usklajevanja

Globalno usklajevanje v CUDA sicer ni podprto eksplicitno, a ga lahko realiziramo preprosto tako, da končamo izvajanje programskih jeder na GPE, nato izvedemo potrebne operacije (lahko na CPE, ali pa kot drugo programsko jedro na GPE) ter ponovno poženemo usklajena programska jedra. Pri tem ni potrebno sprostiti dodeljenega pomnilnika na grafični kartici, prav tako tudi ni potrebno podatkov ponovno prenesti na GPE. Uporaba je prikazana v algoritmu 3.

Algoritem 3 Primer globalnega usklajevanja izvedenega na CPE.

pokliči programsko jedro na GPE
obdelaj podatke // korak se izvede šele, kadar se izvajanje programskega jedra zaključi
pokliči programsko jedro na GPE

Edini ekspliciten način usklajevanja v CUDA je ukaz **__syncthreads()**, ki povzroči uskladitev (tipa pregrada) vseh niti v trenutnem bloku. Ukaz blokira izvajanje niti, dokler vse niti v istem bloku ne prispejo do točke uskladitve, nato se izvajanje vseh niti v bloku nadaljuje. Uporaba je prikazana v algoritmu 4.

Algoritem 4 Primer bločnega usklajevanja izvedenega na GPE.

obdelaj podatke
__syncthreads()
obdelaj podatke // ob začetku izvajanja tega koraka, so vse niti v bloku usklajene

3.4.2 Prevajalnik

Pri prevajanju jezika se uporablja Nvidia prevajalnik *nvcc*, ki je zasnovan na odprtokodnem prevajalniku *Open64* [38]. Zelo pomembno je, da se zavedamo nekaj osnovnih implementacijskih lastnosti prevajalnika *nvcc*, saj nam to omogoči razumevanje omejitev jezika samega. Ena najpomembnejših je, da so pravzaprav vse funkcije, ki jih kličemo, tako imenovane *vstavljene* funkcije. Ob klicu takih funkcij, prevajalnik preprosto vrine klicano funkcijo v del kode, kjer je bila klicana. Lahko bi rekli, da zamenja klic funkcije z njenim telesom. Posledica take implementacije je, da uporaba rekurzivnih funkcij ni mogoča. Poleg tega uporaba referenc na funkcije ni mogoča, saj prevajalnik ne ve, katero funkcijo naj razširi v času prevajanja.

Prevajalnik omogoča razhroščevanje z uporabo prirejenega GNU Debugger (GDB) razhroščevalnika, ki se imenuje CUDA-GDB. Razhroščevanje je omogočeno za kodo pisano za GPE in tudi za kodo pisano za CPE. Običajno je to del CPE kode, ki požene programsko jedro. CUDA-GDB deluje izključno na Linux sistemih, pri čemer mora biti X strežnik izključen, ali moramo imeti dodaten GPE, ki ga X strežnik ne uporablja[4]. Za razliko od podobnega problema pri časovni omejitvi, kjer je bilo možno začasno izključiti X strežnik in preklopiti na terminalski način dela, to pri razhroščevalniku ni dovolj — X strežnik moramo izključiti.

Ena od pomembnejših stikal prevajalnika je ,,$--ptxas-options=-v$", katerega namen je izpis podrobnosti postopka prevajanja. Najpogosteje nas zanima koliko spremenljivk

je prevajalnik dodelil v posamezen tip pomnilnika. Če je prevajalnik uporabil lokalni pomnilnik, nas to še posebej zanima, saj se uporabi le-tega najpogosteje želimo izogniti. Glede na to, da prevajalnik samodejno izvaja optimizacijo, moramo biti še posebej pozorni na izpis, ki ga omogoči vključitev tega stikala.

Poglavje 4

OpenCL

OpenCL (ang. *Open Computing Language*)[5] je odprt standard za porazdeljeno računanje na heterogenih okoljih, ki ga vzdržuje Khronos Group. Pravzaprav je OpenCL podoben odprtim industrijskim standardom, ki jih tudi vzdržuje Khronos Group: OpenGL za 3D grafiko in OpenAL za zvok. Standard OpenCL, za razliko od Nvidia CUDA, ni predviden izključno za Nvidia GPE in lahko programe napisane za OpenCL poganjamo tudi na AMD grafičnih karticah.

Vredno je omeniti, da je imel AMD prav tako kot Nvidia lastno okolje za računanje na GPE, ki se je imenovalo *Close to Metal*, vendar se je AMD odločil opustiti razvoj in se osredotočiti izključno na izdelavo delujoče OpenCL implementacije. Zaradi tega v tej nalogi *Close to Metal* ne bo opisan.

Kljub temu, da je OpenCL dokaj nov standard, ima že delujoči implementaciji tako s strani Nvidie kot tudi AMD-ja. Ker je Nvidia že imela delujočo implementacijo za računanje na GPE CUDA, je OpenCL za Nvidia GPE implementiran kot sloj nad CUDA. OpenCL je tako le eden od možnih načinov dostopanja do virov GPE. Glede na to, da je AMD opustil razvoj svojega načina za dostopanje do virov GPE, je OpenCL edina možnost za AMD GPE.

OpenCL nudi zelo ugodno prozornost glede strojnega sloja in omogoča razvijalcem pisanje programov pospešenih z GPE za splošno uporabo, podobno kot to omogočata industrijska standarda za grafiko in zvok. Vendar OpenCL nudi nekaj več kot Nvidia CUDA. Omogoča namreč vzporedno izvajanje programov po vsej strojni opremi — ne samo GPE, temveč tudi CPE. OpenCL programi povzporejajo po platformah in le-te predstavljajo posamezen kos strojne opreme (bodisi GPE, bodisi CPE).

4.1 Organizacija procesov

Posamezni procesi se imenujejo delovne enote (ang. *work-items*). Vsaka delovna enota spada k eni delovni skupini (ang. *work-group*). Komunikacija med procesi je načeloma zagotovljena samo za delovne enote znotraj iste delovne skupine. Če primerjamo s CUDA, je delovna skupina enaka bloku, delovna enota pa niti.

4.2 Struktura pomnilnika

Pomnilnik, do katerega dostopamo preko OpenCL je razdeljen na štiri nivoje:

- zasebni pomnilnik
- lokalni pomnilnik
- globalni pomnilnik
- nespremenljivi pomnilnik

4.2.1 Zasebni pomnilnik

Zasebni pomnilnik predstavlja pomnilnik, ki ga naslavlja posamezna delovna enota in je lasten tej delovni enoti. Do spremenljivk, ki jih definiramo v zasebnem pomnilniku ene delovne enote, ne moremo dostopati v neki drugi delovni enoti.

4.2.2 Lokalni pomnilnik

Lokalni pomnilnik predstavlja delež pomnilnika, ki je dodeljen posamezni delovni skupini. Do podatkov, ki pripadajo eni delovni skupini, lahko dostopajo vse delovne enote iz iste skupine. Dostop do lokalnega pomnilnika, katerega si lasti neka druga delovna skupina, ni mogoč. Lokalni pomnilnik je lahko implementiran bodisi kot območje posebnega pomnilnika na strojni opremi (če ga ta omogoča — npr. deljeni pomnilnik na Nvidinih GPE) ali kot področje v globalnem pomnilniku.

4.2.3 Globalni pomnilnik

Globalni pomnilnik dovoljuje tako pisanje kot tudi branje vsem delovnim enotam iz vseh delovnih skupin. Posamezne delovne enote lahko berejo poljubne podatke iz globalnega pomnilnika, saj si področja le-tega ne lastijo posamezne enote ali skupine — to je podobno dostopanju do globalnih spremenljivk v jeziku C99. Glede na zmogljivost strojne opreme, je tukaj lahko omogočen predpomnilnik.

4.2.4 Nespremenljivi pomnilnik

Nespremenljivi pomnilnik predstavlja delež globalnega pomnilnika, ki se ne spreminja v času izvajanja programskega jedra. Pred zagonom programskega jedra je potrebno naložiti željene podatke v nespremenljivi pomnilnik. Veljajo enake zahteve pri predpomnilniku, kot za globalni pomnilnik, saj je to dejansko del globalnega pomnilnika.

4.3 Jezik OpenCL

Sam jezik za pisanje programskih jeder je zasnovan na C99, vendar je na voljo tudi veliko vmesnikov za C++, Python, Java itd. Posameznih vmesnikov v nalogi ne bomo omenjali. Za določanje, katera funkcija naj ima vlogo programskega jedra, se deklaraciji funkcije

doda predpona ***__kernel***. Vključenih je še nekaj predpon, ki določajo tip pomnilnika, ki ga uporabljajo spremenljivke: ***__global***, ***__local***, ***__constant*** in ***__private***. Te predpone predstavljajo tipe pomnilnika opisane zgoraj.

4.3.1 Načini usklajevanja

Tako kot pri CUDA, je tudi pri OpenCL možna globalna uskladitev s končanjem izvajanja programskih jeder in ponovni zagon ob opravljeni uskladitvi s strani CPE. To počnemo na enak način, kot smo pokazali pri CUDA.

OpenCL pozna metodo usklajevanja tipa pregrada. Kličemo jo preko funkcije **barrier**. Gre za tipično pregrado, kjer se pregrada izvede za vse enote posamezne delovne skupine. Uskladimo lahko samo enote znotraj iste delovne skupine. Funkcija **barrier** zahteva parameter, ki je lahko **CLK_LOCAL_MEM_FENCE**, **CLK_GLOBAL_MEM_FENCE** ali kombinacija obeh. Ta parameter določa uskladitev podatkov v pomnilniku. Vrednost **CLK_LOCAL_MEM_FENCE** zagotovi, da so podatki v lokalnem pomnilniku usklajeni, med tem, ko vrednost **CLK_GLOBAL_MEM_FENCE** zagotovi, da so usklajeni podatki v globalnem pomnilniku. Uporaba je prikazana v algoritmu 5.

Algoritem 5 Primer usklajevanja enot delovne skupine na GPE.

obdelaj podatke v lokalnem in globalnem pomnilniku
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)
obdelaj v usklajene podatke in shrani v lokalni pomnilnik
barrier(CLK_LOCAL_MEM_FENCE)
obdelaj usklajene podatke in shrani v globalni pomnilnik
barrier(CLK_GLOBAL_MEM_FENCE)
obdelaj usklajene podatke

4.3.2 Prevajalnik

Ker je OpenCL odprt standard in sam po sebi nima implementacije, je vprašanje prevajalnika odvisno od proizvajalca opreme. AMD je izdal v okviru *ATI Stream SDK 2.0* (za Windows in Linux) implementacije za nekaj svojih GPE ter za vse CPE, ki podpirajo SSE3 (ang. *Streaming SIMD Extensions 3*), kar vključuje tudi Intelove CPE. Nvidia je prav tako izdala implementacije za svoje GPE kot sloj nad CUDA. Posameznih implementacij standarda ne bomo opisali.

Vseeno bomo omenili atraktivno zmogljivost, ki jo nudijo implementacije OpenCL in sicer prevajanje programskega jedra v času izvajanja programa. Za vzporedno računanje potrebujemo samo programsko jedro, zato lahko celoten program, ki se ne bo izvajal vzporedno, pišemo v poljubnem jeziku. V CUDA to naredimo tako, kot bi to sicer naredili z navadnim programom. OpenCL nam nudi še boljšo možnost: programsko jedro lahko prevedemo v času izvajanja programa in sicer tako, da celotno programsko jedro shranimo v niz (npr. preberemo iz datoteke). Klic funkcije **clBuildProgram** nato prevede program. Seveda imamo pa tudi zmožnost prevajanja programskega jedra že vnaprej. Prevajanje v času izvajanja programa nam lahko omogoči npr. optimizacijo konstant programskega jedra

za potrebe določenega problema. Vendar je potrebno poudariti, da tak način prevajanja programskega jedra podaljša čas izvajanja celotnega programa.

Poglavje 5

Kombinatorična optimizacija

Kombinatorična optimizacija (ang. *combinatorial optimization*) je področje uporabne matematike in teoretičnega računalništva, ki se ukvarja z iskanjem najcenejše rešitve določenih diskretnih problemov. Za razliko od problemov zvezne optimizacije – kjer so parametri, ki definirajo rešitev, zvezne narave – se pri kombinatorični optimizaciji ukvarjamo s problemi, kjer so parametri, ki definirajo rešitev, diskretni. Rešitev lahko tako zavzame vrednost neke podmnožice, grafa, celega števila in podobnih diskretnih struktur. Definicija 5.1 formalno definira probleme s katerimi se ukvarja področje kombinatorične optimizacije.

Definicija 5.1 *Problem kombinatorične optimizacije* $\mathbf{P} = (\mathcal{S}, \mathbf{f})$*, kjer je* $\mathcal{S}$ *prostor rešitev, lahko definiramo[6] kot:*

- *množico vrednosti* $\mathbf{X} = \mathbf{x_1}, ..., \mathbf{x_n}$
- *domeno spremenljivk* $\mathbf{D_1}, ..., \mathbf{D_n}$
- *omejitve med spremenljivkami*
- *cenilno funkcijo* $\mathbf{f}$*, ki jo želimo minimizirati ali maksimizirati, kjer* $\mathbf{f} : \mathbf{D_1} \times ... \times \mathbf{D_n} \to \mathbb{R}^+$

Uporabnost kombinatorične optimizacije je razvidna iz široke palete problemov, ki jih je možno prevesti v to obliko: optimizacija voznega reda, iskanje najbolj ekonomičnih poti, iskanje minimalnih vpetih dreves, problem nahrbtnika, itd. Ogledali si bomo nekaj problemov na grafih, ki jih rešuje kombinatorična optimizacija.

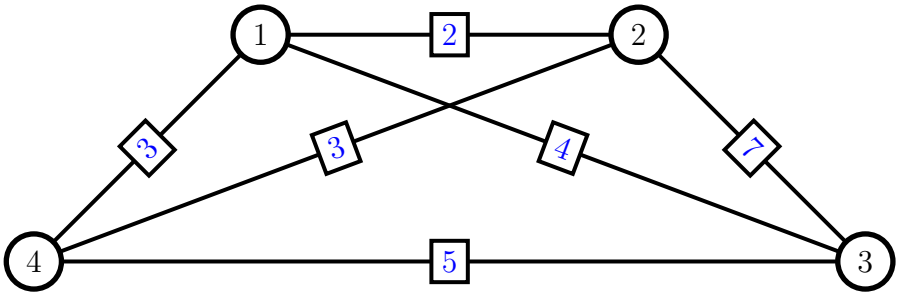
5.1 Problem trgovskega potnika

Pri optimizacijskemu problemu trgovskega potnika (ang. *TSP* ali *Traveling Salesman Problem*) imamo podan graf $G(V, E)$, kjer vozlišča V predstavljajo mesta in povezave E predstavljajo povezave med mesti. Običajno je vsako mesto povezano z vsakim. Povezave so obtežene tako, da utež povezave predstavlja razdaljo med mestoma, ki jih povezuje.

Hamiltonov cikel definiramo kot pot na grafu G, ki vsako vozlišče grafa, razen začetnega, obišče natančno enkrat. Začetno vozlišče obišče natančno dvakrat in sicer kot prvi in zadnji element poti. Pri problemu trgovskega potnika iščemo najkrajši Hamiltonov cikel. Definicija 5.2 poda definicijo cenilne funkcije f za problem trgovskega potnika.

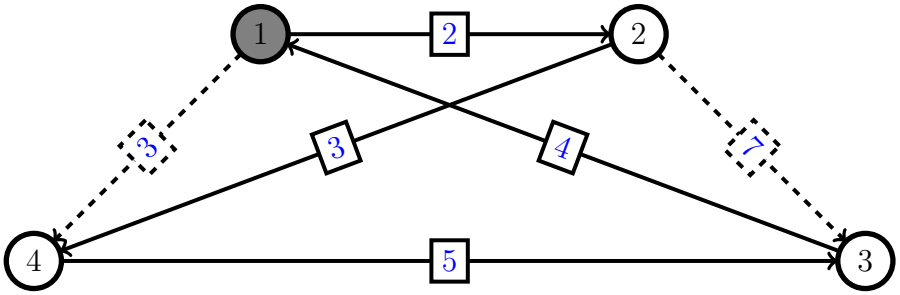
Definicija 5.2 *Naj bo* $\mathbf{Z}$ *množica vseh Hamiltonovih ciklov na grafu* $\mathbf{G}(\mathbf{V}, \mathbf{E})$. *Vzemimo nek poljuben Hamiltonov cikel* $\mathbf{H} \in \mathbf{Z}$. *Naj* $\mathbf{d}(\mathbf{v}, \mathbf{v'}), \mathbf{v}, \mathbf{v'} \in \mathbf{V}$ *predstavlja razdaljo med vozliščema in naj bo* $\mathbf{D}(\mathbf{H}) = \sum_{\mathbf{i=1}}^{|\mathbf{V}|} \mathbf{d}(\mathbf{H_{i-1}}, \mathbf{H_i})$. $\mathbf{H}$ *je najkrajši Hamiltonov cikel, če velja, da* $\mathbf{D}(\mathbf{H}) \leq \mathbf{D}(\mathbf{H'}), \forall \mathbf{H'} \in \mathbf{Z}$.

Graf 5.1 prikazuje problem trgovskega potnika, kjer tevilke na povezavah predstavljajo razdaljo.



Graf 5.1: Problem trgovskega potnika na grafu s štirimi mesti, kjer utež povezave predstavlja razdaljo med mestoma, ki jih povezuje.

Rešitev TSP problema z grafa 5.1 je podana na grafu 5.2. Dolžina najkrajše poti je 14. Povezave, ki v rešitev niso vključene so označene črtkano.



Graf 5.2: Problem trgovskega potnika na grafu s štirimi mesti. Rešitev problema TSP z grafa 5.1.

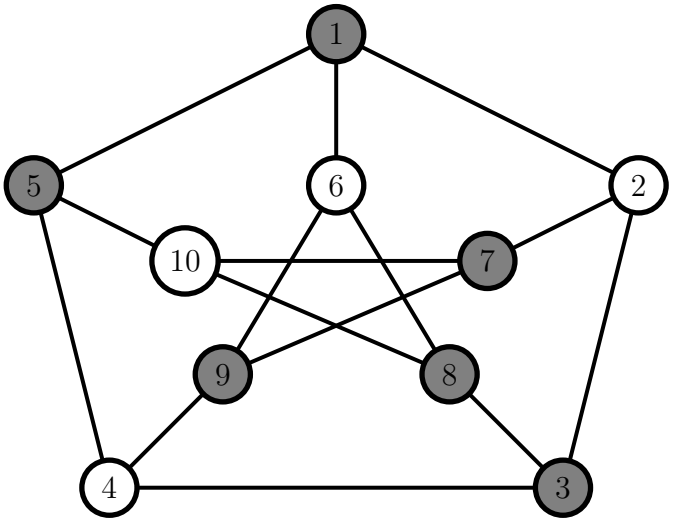
5.2 Najmanjše pokritje grafa

Optimizacijski problem najmanjšega točkovnega pokritja grafa (ang. *minimum vertex cover*) spada v razred NP-ekvivalentnih problemov. Poleg točkovnega poznamo tudi povezavno pokritje grafa, vendar ga tukaj ne bomo predstavili. Od sedaj naprej, se bomo sklicevali na *pokritje grafa*, pri čemer mislimo na točkovno pokritje grafa.

Definicija 5.3 *Naj bo* $\mathbf{V}$ *množica vozlišč in* $\mathbf{E}$ *množica povezav grafa* $\mathbf{G}(\mathbf{V}, \mathbf{E})$. *Pokritje grafa* $\mathbf{G}$ *je taka množica* $\mathbf{C} \subset \mathbf{V}$, *za katero velja, da, če vzamemo poljubno povezavo* $\mathbf{p} \in \mathbf{E}$ *se mora najmanj eno od vozlišč, ki jih povezuje povezava* $\mathbf{p}$ *nahajati v* $\mathbf{C}$.

Definicija 5.3 poda definicijo pokritja grafa. Pokritje grafa je preprosto poiskati, saj množica vseh vozlišč grafa G že predstavlja pokritje. Definirajmo še koncept najmanjšega pokritja grafa — dejansko definiramo funkcijo f iz definicije 5.1:

Definicija 5.4 *Naj* $\mathbf{Z}$ *predstavlja množico vseh pokritij grafa* $\mathbf{G}(\mathbf{V}, \mathbf{E})$. *Najmanjše pokritje grafa* $\mathbf{G}$ *je taka množica* $\mathbf{C} \in \mathbf{Z}$, *za katero velja* $|\mathbf{C}| \leq |\mathbf{C}'|, \forall \mathbf{C}' \in \mathbf{Z}$.



Graf 5.3: Najmanjše pokritje Petersenovega grafa. Vozlišča, ki so elementi najmanjšega pokritja, so obarvana sivo.

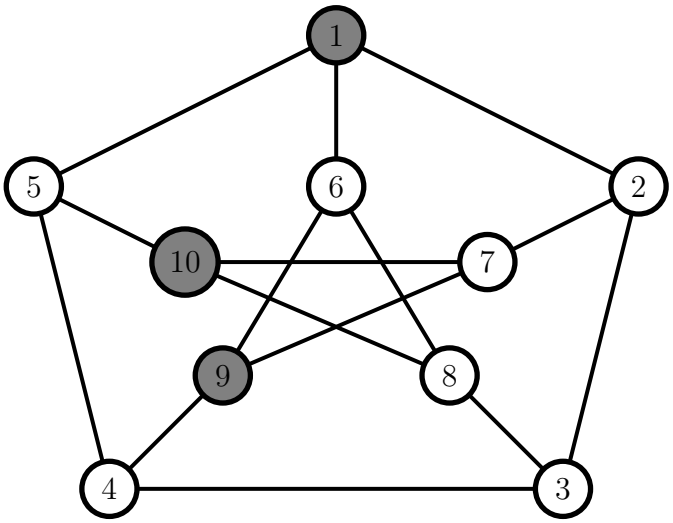
Z grafom 5.3 je prikazan primer enega od najmanjših točkovnih pokritij v skladu z definicijo 5.4. Poleg omenjenega pokritja obstajajo še druga najmanjša točkovna pokritja.

5.3 Največja neodvisna množica

Optimizacijski problem največje neodvisne množice (ang. *maximum stable set*, *maximum independent set*) tudi spada v razred NP-ekvivalentnih problemov[8].

Definicija 5.5 *Naj bo* $\mathbf{V}$ *množica vozlišč in* $\mathbf{E}$ *množica povezav grafa* $\mathbf{G}(\mathbf{V}, \mathbf{E})$. *Neodvisna množica vozlišč grafa* $\mathbf{G}$ *je taka množica* $\mathbf{I} \subset \mathbf{V}$, *da za vsak par vozlišč* $\mathbf{v}, \mathbf{w} \in \mathbf{I}$ *velja, da* $(\mathbf{v}, \mathbf{w}) \notin \mathbf{E}$.

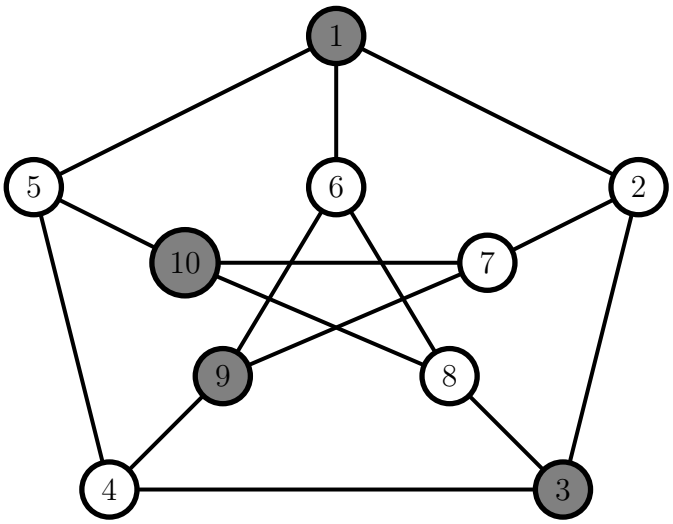
Definicija 5.5 poda definicijo neodvisne množice. Primer slednje za Petersenov graf je prikazan z grafom 5.4. Prikazana je samo ena izmed možnih množic.



Graf 5.4: Neodvisna množica v Petersenovem grafu. Vozlišča, ki so elementi neodvisne množice, so obarvana sivo.

Problem iskanja neodvisne množice je povezan s problemom iskanja točkovnega pokritja grafa [9]. Vozlišča grafa G, ki niso vsebovana v pokritju C na grafu G, definirajo neodvisno množico I na grafu G . Poglejmo si še definicijo največje neodvisne množice.

Definicija 5.6 *Naj* $\mathbf{Z}$ *predstavlja množico vseh neodvisnih množic grafa* $\mathbf{G}(\mathbf{V}, \mathbf{E})$. *Največja neodvisna množica grafa* $\mathbf{G}$ *je taka množica* $\mathbf{I} \in \mathbf{Z}$ *za katero velja* $|\mathbf{I}| \geq |\mathbf{I}'|, \forall \mathbf{I}' \in \mathbf{Z}$.



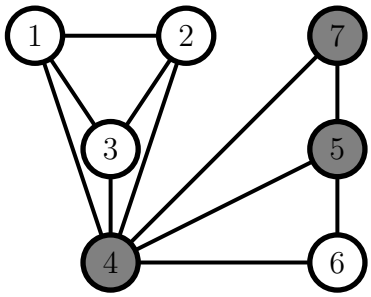
Graf 5.5: Največja neodvisna množica v Petersenovem grafu. Vozlišča, ki so elementi največje neodvisne množice, so obarvana sivo.

Definicija 5.6 poda definicijo *največje* neodvisne množice. Primer slednje za Petersenov graf je prikazan na grafu 5.5. Prikazana je samo ena izmed možnih množic. Problem iskanja največje neodvisne množice je povezan s problemom iskanja najmanjšega pokritja grafa na enak način, kot je bila povezava definirana med pokritjem grafa ter neodvisno množico.

5.4 Največji polni podgraf

Optimizacijski problem največjega polnega podgrafa (ang. *maximum clique problem*), tako kot prejšnja dva problema, tudi spada v razred NP-ekvivalentnih problemov [10].

Definicija 5.7 *Naj bo* $\mathbf{V}$ *množica vozlišč grafa* $\mathbf{G}(\mathbf{V}, \mathbf{E})$*. Polni podgraf grafa* $\mathbf{G}$ *je taka množica* $\mathbf{C} \subset \mathbf{V}$*, da za vsak par vozlišč* $(\mathbf{v}, \mathbf{w}) \in \mathbf{C}$ *velja, da so na grafu* $\mathbf{G}$ *sosednja.*

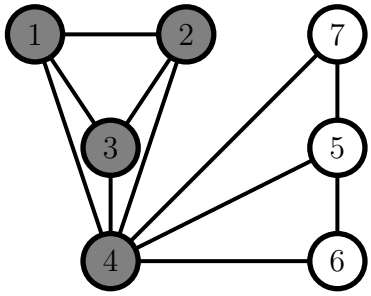


Graf 5.6: Polni podgraf: vozlišča, ki spadajo v polni podgraf, so obarvana sivo.

Definicija 5.7 poda definicijo polnega podgrafa. Primer polnega podgrafa je prikazan z grafom 5.6. Prikazan je le eden izmed polnih podgrafov. Poglejmo si še največji polni podgraf.

Definicija 5.8 *Naj* $\mathbf{Z}$ *predstavlja množico vseh polnih podgrafov grafa* $\mathbf{G}$*. Največji polni podgraf grafa* $\mathbf{G}$ *je taka množica* $\mathbf{C} \in \mathbf{Z}$ *za katero velja* $|\mathbf{C}| \geq |\mathbf{C}'|, \forall \mathbf{C}' \in \mathbf{Z}$*.*

Definicija 5.8 definira koncept *največjega* polnega podgrafa. Primer le-tega je prikazan z grafom 5.7. Prikazana je edina možna izbira.



Graf 5.7: Največji polni podgraf: vozlišča, ki so elementi največjega polnega podgrafa, so obarvana sivo.

Poglavje 6

Metahevristična optimizacija

Eno izmed področij optimizacijskih algoritmov so tako imenovani metahevristični optimizacijski algoritmi. Že po imenu lahko razberemo, da uporabljajo za doseganje cilja (t.j. optimizacija) hevristiko. Algoritmi iz tega področja nam sicer ne zagotavljajo optimalnosti rešitve, ki jo najdejo, vendar lahko kljub temu najdejo rešitev, ki zadovoljuje naše potrebe, v kolikor se zadovoljimo s približkom. Za preproste probleme, kjer lahko najdemo optimalno rešitev razmeroma hitro, je metahevristika slaba izbira. Veliko je takih problemov, kjer časa, potrebnega za iskanje optimalne rešitve, preprosto ni dovolj. To so npr. NP-ekvivalentni problemi.

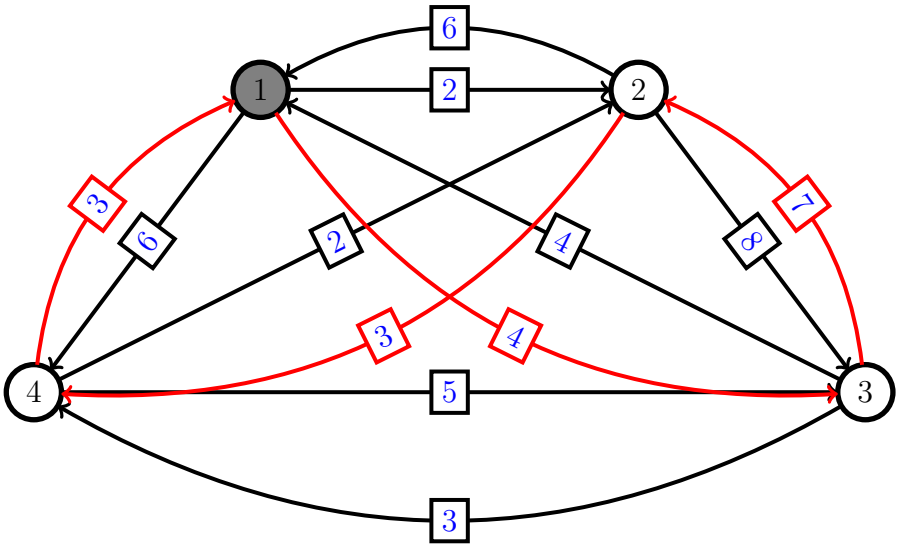
V tej nalogi so obravnavane implementacije algoritmov: simulirano ohlajanje, navzkrižna entropija ter optimizacija s kolonijo mravelj. Omenili bomo še dve metahevristiki in sicer evolucijski algoritem in iskanje s prepovedmi. Metahevristike so načeloma splošni algoritmi oz. strategije za reševanje problemov optimizacije. Zaradi jasnejše razlage, se bomo omejili na aplikacijo metahevristik za reševanje določenega problema.

6.1 Iskanje optimuma

Optimum je željen ekstrem funkcije, ki je predmet optimizacije. V primeru, ko iščemo najcenejšo rešitev, želimo optimizirati funkcijo, ki poda oceno rešitve in je željen ekstrem take funkcije minimum — če bi bil željen ekstrem maksimum, bi iskali najdražjo rešitev.

Ker se bomo omejili na iskanje optimuma v problemih kombinatorične optimizacije, je vhod take funkcije, ki nam poda oceno, določena permutacija. Optimum lahko ločimo na lokalni in globalni, pri čemer moramo povedati, da je globalni optimum vedno tudi lokalni optimum, obratno pa ne velja za lokalni optimum. Zaradi jasnosti razlage, se bomo omejili na problem trgovskega potnika, na katerem bomo prikazali razlike med različnimi optimumi.

Vzemimo torej kot začetno konfiguracijo problem trgovskega potnika, ki je podan v grafu 6.1. Začetno konfiguracijo predstavlja Hamiltonov cikel $(1, 3, 2, 4, 1)$, skupna razdalja je 17.



Graf 6.1: Problem trgovskega potnika s štirimi mesti. Začetna točka je obarvana sivo. Trenutna rešitev je obarvana rdeče.

Definirajmo sedaj še operacijo, ki jo lahko izvajamo za izboljšavo rešitve: zamenjamo lahko poljubni dve mesti v urejenem seznamu, ki ga predstavlja Hamiltonov cikel, s tem, da ne dovolimo zamenjav, ki vključujejo prvo (začetno) ali zadnje mesto[1]. Naj bo soseščina neke rešitve S definirana kot $N(S)$ in naj predstavlja množico vseh možnih rešitev, ki jih lahko dobimo, če izvedemo operacijo zamenjave na rešitvi S. Soseščina rešitve $(1, 3, 2, 4, 1)$ tako vključuje rešitve $(1, 2, 3, 4, 1), (1, 3, 4, 2, 1)$ in $(1, 4, 2, 3, 1)$, kot tudi rešitev samo.

Potrebujemo funkcijo, ki nam poda kakovost rešitve. Za problem trgovskega potnika bomo kot kakovost rešitve uporabili vsoto vseh razdalj povezav, ki jih mora prehoditi trgovec. To je ravno funkcija D, ki smo jo definirali v poglavju Kombinatorična optimizacija. Zgoraj omenjena rešitev ima skupno razdaljo 17.

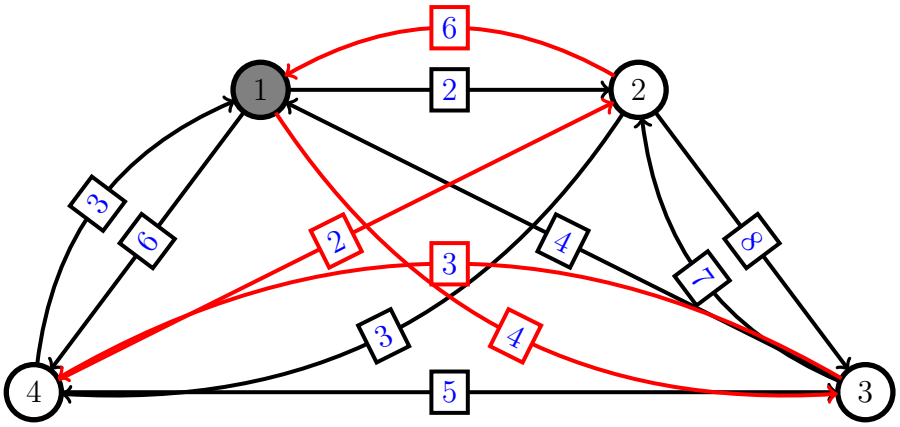
6.1.1 Lokalni optimum

Ker bomo iskali najkrajši Hamiltonov cikel, se bomo omejili na lokalni minimum. Poglejmo si najprej definicijo lokalnega minimuma, podano v definiciji 6.1.

Definicija 6.1 *Naj* $\mathbf{N(S)}$ *predstavlja soseščino rešitve* $\mathbf{S}$. *Lokalni minimum je taka rešitev* $\bar{\mathbf{S}} \in \mathbf{N(S)}$ *za katero velja* $\mathbf{Q}(\bar{\mathbf{S}}) \leq \mathbf{Q}(\mathbf{S}'), \forall \mathbf{S}' \in \mathbf{N(S)}$.

Sedaj uporabimo to definicijo, da poiščemo lokalni minimum soseščine rešitve $(1, 3, 2, 4, 1)$. Kot smo prej omenili, soseščina vključuje rešitve $(1, 2, 3, 4, 1), (1, 3, 4, 2, 1), (1, 4, 2, 3, 1)$ in pa rešitev samo. Kakovosti teh rešitev so po vrsti: $16, 15, 20$ ter 17 za trenutno rešitev. Lokalni minimum je v tem primeru $(1, 3, 4, 2, 1)$, ki ima kakovost 15, prikazuje ga graf 6.2.

[1] V kolikor bi to dovolili, bi moralo biti zadnje mesto vedno enako začetnemu.



Graf 6.2: Lokalni minimum $(1, 3, 4, 2, 1)$ rešitve $(1, 3, 2, 4, 1)$. Začetna točka je obarvana sivo. Povezave, ki predstavljajo lokalni minimum, so obarvane rdeče.

Sledenje lokalnim optimumom ni dober postopek za iskanje najboljše rešitve. Slednje razumemo pod pojmom izostritve (ang. *intensification*). Postopku, ki sledi izključno takim lokalnim optimumom, pravimo požrešna metoda (ang. *greedy method*). Pogosto se boljša rešitev nahaja nekje izven soseščine $N(S)$. Da pridemo do take soseščine $N(S')$, ki premore boljše lokalne optimume kot soseščina $N(S)$, je potrebno izbrati zamenjave, ki lokalno niso optimalne. Slednje razumemo pod pojmom razpršitve (ang. *diversification*).

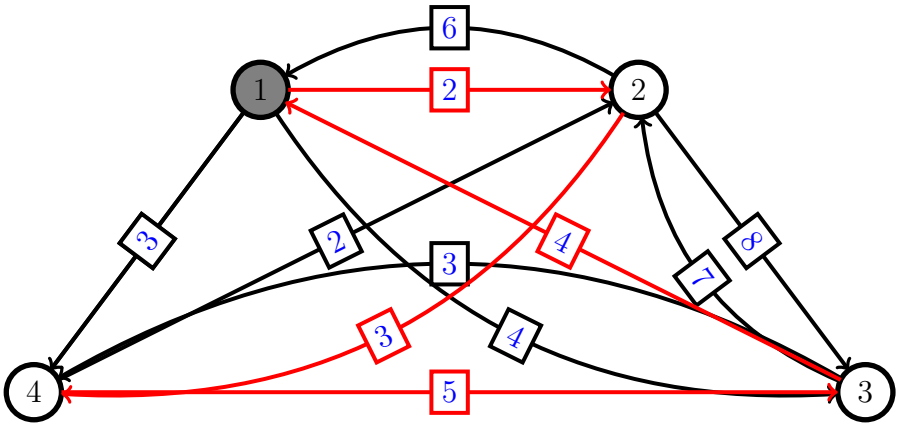
Kljub temu pa moramo pomniti, da nas prekomerno sledenje slabim rešitvam iz soseščine $N(S)$ lahko pripelje do take soseščine $N(\bar{S})$, kjer je lokalni optimum slabši kot v $N(S)$. Potrebno je izvajati smiselno obliko kompromisa med postopkoma izostritve in razpršitve.

6.1.2 Globalni optimum

Medtem, ko se lokalni optimum nanaša zgolj na soseščino rešitve S, se globalni optimum nanaša na *vse* možne rešitve, ki jih lahko dobimo z uporabo operacije zamenjave. Definicija 6.2 poda definicijo globalnega minimuma, ki je za naše potrebe optimum.

Definicija 6.2 *Naj* $\mathbf{Z}$ *predstavlja množico vseh možnih rešitev. Globalni minimum je rešitev* $\mathbf{S} \in \mathbf{Z}$ *za katero velja* $\mathbf{Q}(\mathbf{S}) \leq \mathbf{Q}(\mathbf{S}'), \forall \mathbf{S}' \in \mathbf{Z}$.

Trenutno ne poznamo hitrejšega postopka iskanja globalnega optimuma, kot je preiskovanje vseh preostalih možnih rešitev, ki pa v tem primeru niso omejene zgolj na dolčeno soseščino, temveč na vse možne rešitve nasploh. Takšno preiskovanje je zelo potratno in se izkaže, da je, z izjemo nekaterih manjših problemov, za to potrebno preveč časa, da bi bilo praktično izvedljivo. Za naš omejen problem je preiskovanje celotnega prostora možno, s čimer smo ugotovili, da je globalni optimum cikel (1, 2, 4, 3, 1) skupne dolžine 14 . Prikazan je v grafu 6.3.



Graf 6.3: Globalni minimum $(1, 2, 4, 3, 1)$. Začetna točka je obarvana sivo. Povezave, ki predstavljajo globalni minimum, so obarvane rdeče.

6.2 Povzporejanje metahevristik

Naiven način povzporejanja metahevristik je zelo preprost: vzporedno poženemo več instanc programov in, ko vrnejo rešitve, preprosto izberemo najboljšo. Tak način povzporejanja je možen zato, ker večina metahevristik uporablja naključna števila. Zato je tudi kakovost rešitve, ki jo vrne tak algoritem, v povprečju boljša, če izvedemo čim več neodvisnih poizkusov. Tako pokrijemo večji spekter rešitev. Kljub temu pa takšen način povzporejanja ni dober, saj lahko nekaj izmed instanc programov že na začetku zabrede v lokalni optimum in tam ostane do konca.

Postopek lahko izboljšamo tako, da vzporedno poženemo več instanc programov, ter ob določeni iteraciji uskladimo instance in postopek nadaljujemo. Ob uskladitvi razširimo določene vrednosti med vse instance. Te vrednosti so odvisne od algoritma, vendar so to vedno vrednosti, ki vplivajo na postopek tvorjenja novih rešitev. Tak način povzporejanja se razlikuje od naivnega pristopa, kjer preprosto poženemo več instanc programov in pustimo, da tečejo dokler ne vrne vsaka lastno rešitev. Pri zgoraj navedenem načinu namreč izvedemo vmesni uskladitveni korak, kjer že del dobljene rešitve razširimo na ostale instance programov. Problem ostane pri odločitvi, kako pogosto naj izvajamo usklajevanje instanc, saj prepogosto usklajevanje upočasni izvajalni čas.

V naslednjih podpoglavjih si bomo ogledali razne metahevristične optimizacijske algoritme. Ogledali si bomo že nekaj obstoječih vzporednih implementacij teh algoritmov. Ker je vzporednih implementacij veliko, si bomo ogledali takšne, ki že delujejo na GPE, oziroma takšne, ki bi bile čim bolj smiselne za računanje na GPE.

6.3 Simulirano ohlajanje

Metoda metahevristične optimizacije simulirano ohlajanje (ang. *simulated annealing*) izhaja iz idej statistične mehanike. Je iterativna metoda, kjer začnemo optimizacijo s poljubno rešitvijo (ki je lahko tudi poljubno zgrešena) ter začetno toploto T_0. Ob vsaki iteraciji naključno izberemo eno izmed sosednjih rešitev trenutne rešitve, v odvisnosti od trenutne

toplote sistema T_k. Sam mehanizem izbire rešitev je pravzaprav najpomembnejši del metode. Običajno ne želimo izbrati vedno najboljše sosednje rešitve, saj bi se omejili na lokalni minimum funkcije, na kateri izvajamo optimizacijo in bi bila metoda požrešna. Vendar je tudi res, da nočemo izbrati vedno slabše sosednje rešitve, saj tako ne bomo našli minimuma.

Simulirano ohlajanje rešuje problem na način, ki je opisan v nadaljevanju. Kadar je toplota sistema visoka, obstaja večja verjetnost, da izberemo slabšo rešitev, kot pa če je toplota sistema nizka. Obratno velja, da, ko je toplota sistema zelo nizka, izberemo najboljšo rešitev skoraj vedno. Pri simuliranem ohlajanju govorimo o energiji sistema E, ki je pravzaprav ocena (ugodnost) rešitve. Za ocenjevanje rešitev bomo uporabili bomo funkcijo f, ki smo jo definirali v poglavju Kombinatorična optimizacija. Definiramo tudi ΔE, ki predstavlja spremembo energije sistema ob določeni spremembi rešitve. Metropolisov algoritem simuliranega ohlajanja pravi[7], da ob vsaki iteraciji tvorimo določeno naključno spremembo rešitve ter jo vključimo v rešitev, če $\Delta E \leq 0$, kar predstavlja izboljšavo rešitve. V nasprotnem primeru, torej $\Delta E > 0$, vključimo spremembo v rešitev z verjetnostjo $P(\Delta E) = \exp(-\Delta E / k_B T_k)$, kjer je T_k trenutna toplota sistema in k_B je Boltzmannova konstanta. Algoritem 6 prikazuje tipično simulirano ohlajanje.

Algoritem 6 Algoritem simuliranega ohlajanja.

α, T_{min}, T_0, $iter_{max} \leftarrow$ inicializiraj
$r \leftarrow$ začetna rešitev
$E \leftarrow f(r)$
$T \leftarrow T_0$
while $T > T_{min}$ **do**
  $iter \leftarrow 0$
  while $iter < iter_{max}$ **do**
    $s \leftarrow$ sosednja rešitev
    $E_s \leftarrow f(s)$
    $E_\Delta \leftarrow E - E_s$
    if $P(E_\Delta) \geq rand(0,1)$ **then**
      $r \leftarrow s$
      $E \leftarrow E_s$
    end if
    $iter \leftarrow iter + 1$
  end while
  $T \leftarrow T \cdot \alpha$
end while

6.3.1 Problemi pri povzporejanju na GPE

Za hranjenje nespremenljivih podatkov, kot je na primer opis problema (npr. predstavitev grafa), je smiselno uporabiti nespremenljivi ali strukturirani pomnilnik, saj se definicija problema ne spreminja (je konstantna) in sta ta dva tipa pomnilnika strojno optimizirana za množično branje. Poleg tega imamo pri uporabi nespremenljivega in strukturiranega pomnilnika na razpolago tudi hiter predpomnilnik. Pri nespremenljivem pomnilniku se

pojavi problem velikosti, saj je 64 KB pomnilnika za že malo večji problem daleč premalo.

Večji problem se pojavi pri hrambi rešitve (in pogosto tudi sosednje rešitve), saj je deljenega pomnilnika zelo malo. Uporaba globalnega pomnilnika za hrambo rešitve lahko povzroči veliko upočasnitev programa. To še posebej velja za prepisovanje podatkov, npr. pri tvorjenju sosednje rešitve. Globalni pomnilnik je smiselno uporabiti za hrambo najboljše rešitve, saj se izboljšava trenutno najboljše konfiguracije ne najde zelo pogosto. Pravzaprav je pri tipičnih aplikacijah simuliranega ohlajanja[11] do 99% možnih premikov (sosedov rešitve) zavrnjenih.

Moč GPE je zmožnost poganjanja ogromnega števila vzporednih procesov, vendar, če mora vsaka nit hraniti celotno rešitev, to pomeni, da se deljeni pomnilnik multiprocesorja zelo hitro napolni in ni možno poganjati velikega števila niti. Prostora zelo hitro zmanjka že za manjše probleme — pogosto se zgodi, da lahko poganjamo samo eno ali dve niti na multiprocesor.

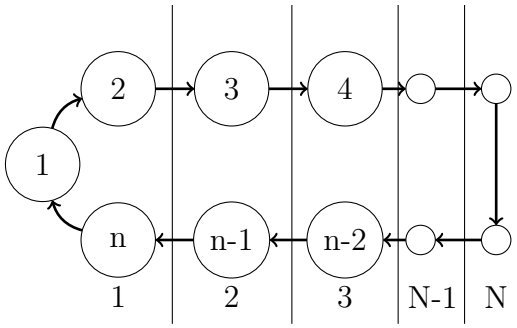
Ena od možnih rešitev je, da izvedemo nekakšno obliko podatkovnega povzporejanja. Idealno bi radi problem razdelili na podprobleme dovolj majhnih velikosti, na katerih bi pognali algoritem simuliranega ohlajanja. Tak algoritem bi se obnašal popolnoma v skladu s simuliranim ohlajanjem, vendar bi bilo potrebno ob koncu algoritma rešitve nekako združiti. Tak postopek pogosto ni splošen, saj je zmožnost razbitja problema na podprobleme odvisna od narave problema [11]. V naslednjem podpoglavju si bomo ogledali primer takega postopka, ki bi ga bilo smiselno implementirati na GPE, vendar ga zaradi omejitve na specifičen problem, ne bomo obravnavali z vidika GPE.

6.3.2 Povzporejanje

V [17] podajo pet različnih načinov povzporejanja. Vendar take metode povzporejanja iz vidika GPE niso smiselne, saj zahtevajo, da posamični vzporedni procesi gradijo celotne rešitve.

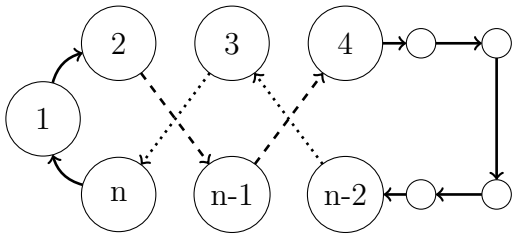
Za GPE je bolj smiselni tak pristop povzporejanja , ki omogoča, da vsak proces tvori samo del rešitve. Tak model povzporejanja je opisan v [33], kjer so povzporejali problem trgovskega potnika na verigi procesorjev brez skupnega pomnilnika.

Načeloma je istočasno izvajanje več naključnih zamenjav v ciklu nemogoče, saj lahko pridemo do več zaprtih podpoti. V izogib takim podpotem, graf najprej razdelimo na več disjunktnih poti. Cikel, ki predstavlja neko začetno rešitev si lahko predstavljamo kot dolgo ,,elastiko", ki jo razdelimo na več delov. Vsak del pripada določenemu procesorju. Primer je podan v grafu 6.4, pri čemer je delitev na N procesorjev izvedena z navpičnimi mejami. Številke pod grafom predstavljajo posamezne procesorje od 1 do N.



Graf 6.4: Graf prikazuje ,,elastiko", ki jo dobimo če raztegnemo cikel, ki predstavlja rešitev.

Premike na rešitvi nato izvajamo tako, da izvedemo zamenjavo med dvema sosednjima deloma — pri čemer sta sosednja dela tista, ki ležita na nasprotnih straneh ,,elastike". Naj bodo mesta označena z $1, 2, ..., n$. Premik rešitve iz mesta 2 do mesta 3 povzroči premik (na drugi strani ,,elastike") iz mesta n do mesta $(n-1)$. To je prikazano v grafu 6.5. Premiku iz mesta 3 do mesta 4 sledi premik iz $(n-1)$ do $(n-2)$. Zgodba se nadaljuje do premika $((n-3)/2)$ do mesta $((n-1)/2)$ s premikom $((n+3)/2)$ do $((n+1)/2)$.



Graf 6.5: Črtkani povezavi predstavljata premik iz mesta 2 do mesta 3, s pikami pa je prikazan dodaten premik iz n do $n-1$, ki nam zagotavlja smiselnost rešitve ob podanem premiku.

Potrebno je opomniti, da predstavljena metoda povzporejanja zahteva liho število mest. V primeru, da je število mest sodo, lahko eno izmed mest ponovimo. Prvo (začetno) mesto moramo izbrati naključno ob vsaki iteraciji, s čimer zadovoljimo stohastičnim zahtevam algoritma simuliranega ohlajanja.

Algoritem poteka tako, da najprej glavni procesor pošlje začetno pot po verigi in jo prejme nazaj na koncu verige. Glavni procesor javi posodobitev tako, da pošlje trenutno toploto po verigi. Ko se posodobitev konča, glavni procesor prejme število izmenjav in novo dolžino poti, pri čemer je vsak procesor odgovoren za dodajanje sprememb na svojem delu rešitve. Glavni procesor nato pošlje razdaljo med $-n/2$ in $n/2+1$, ki določa za koliko naj se pot zavrti, da bo določeno naključno izbrano mesto na položaju 1. Na koncu se glavni procesor odloči ali naj zmanjša toploto sistema oz. nadaljuje ob isti toploti.

Avtorji poročajo pohitritev s faktorjem približno $1/P$, pri čemer je P število procesorjev (največja opravljena meritev je bila s 30 procesorji). Opisani način povzporejanja je specifičen za problem trgovskega potnika.

6.4 Metoda navzkrižne entropije

Metahevristična optimizacijska metoda navzkrižna entropija (ang. *cross-entropy method*) uporablja ideje iz teorije informacij in sicer minimizira relativno entropijo[2] $H(p, q)$ med verjetnostnima porazdelitvima p in q. Za diskretni porazdelitvi p in q velja $H(p, q) = -\sum p(x) \log q(x)$. Relativna entropija doseže minimum, ko sta verjetnostni porazdelitvi p in q enaki.

Pogledali si bomo praktično uporabo te metode za kombinatorično optimizacijo in sicer na problemu iskanja največjega polnega podgrafa, kot je opisano v [26] in [27]. Omejili se bomo na neusmerjen graf $G(V, E)$. Rešitev problema definiramo kot množico vozlišč, ki predstavljajo največji polni podgraf. Definirati moramo še sledeče:

- postopek tvorjenja polnih podgrafov
- postopek posodobitve verjetnostne porazdelitve v skladu z metodo navzkrižne entropije

Tvorjenje rešitev

Postopka opisana v [26] in [27] rešujeta problem na način, ki je opisan v nadaljevanju. Definirajmo matriko M, velikosti $|V| \times |V|$. Naj v_i in v_j predstavljata vozlišči grafa G in $deg(v_i)$ naj predstavlja stopnjo vozlišča v_i, t.j. število povezav, ki jih premore to vozlišče. Sprva inicializiramo i-to vrstico matrike z vrednostjo $M_{ij} = \frac{1}{deg(v_i)}$, če sta si vozlišči v_i in v_j sosednji, ter $M_{ij} = 0$, če nista.

Začnemo s prazno množico $S = \emptyset$. Prvo vozlišče dodamo na podlagi hevristike, ki v [26] ni izrecno določena. Označimo vozlišče, ki smo ga nazadnje dodali kot v_{t-1}. Novo vozlišče v_t izberemo tako, da upoštevamo verjetnostno porazdelitev, ki jo poda vektor M_{t-1}^T. To ponavljamo dokler ne pridemo do takega vozlišča v_t, ki ni del polnega podgrafa. Nato vrnemo množico vozlišč S, ki predstavlja polni podgraf grafa G. S ponavljanjem postopka lahko tvorimo več rešitev.

Posodobitev verjetnostne porazdelitve

Ker želimo, da bodo naslednje rešitve, ki jih bo tvoril zgoraj opisani korak boljše od preteklih, moramo spremeniti parameter, ki določa njihovo tvorjenje. To je ravno matrika M, ki predstavlja verjetnosti vključitve posameznih vozlišč. Posodobitev verjetnostne porazdelitve opravimo na podlagi informacij, ki nam jih nudijo rešitve, katere smo tvorili v prejšnjem koraku.

Kot funkcijo, ki nam poda oceno rešitve, bomo uporabili funkcijo f, definirano v poglavju Kombinatorična optimizacija. Najprej razporedimo vse rešitve po oceni od najmanjše do največje in dobljeno urejeno množico definiramo kot $S_1, ..., S_N$. Nato definiramo še skupino množic Υ_{ij}, ki vsebujejo vse rešitve, kjer je bil uporabljen prehod iz vozlišča i na vozlišče j. Definiramo parameter $\rho \in [0, 1]$, ki določa število najboljših rešitev, ki jih

[2]Pogosto sta v literaturi zapisa za relativno entropijo in za vezano entropijo enaka, čeprav sta si ta dva pojma različna.

bomo uporabili za posodobitev matrike M. Sedaj izračunamo vzorec z uporabo 6.1 in novo vrednost elementa (i, j) matrike M z uporabo 6.2.

$$\tau = \cup_{i=\lceil(1-\rho)N\rceil}^{N} S_i. \tag{6.1}$$

$$M_{ij} = \frac{\sum_{k=1}^{N} I_{f(S_k)\geq\tau} I_{S_k\in\Upsilon_{ij}}}{\sum_{k=1}^{N} I_{f(S_k)\geq\tau}}. \tag{6.2}$$

Če povzamemo, je metoda navzkrižne entropije prikazana v algoritmu 7. Koraka tvorjenja rešitev in posodobitve verjetnostne porazdelitve sta enaka, kot opisana zgoraj.

Algoritem 7 Metoda navzkrižne entropije.

$N \leftarrow$ število rešitev
$M \leftarrow$ inicializiraj matriko
$najboljšaRešitev \leftarrow \emptyset$
while *iteracije brez izboljšave* $<$ *max iteracij brez izboljšave* **do**
  $i \leftarrow 1$
  while $i \leq N$ **do**
    $S_i \leftarrow tvoriRešitev(M)$
    $i \leftarrow i + 1$
  end while
  $urejeniSeznam \leftarrow S_1, ..., S_N$ urejeni po naraščajočem vrstnem redu po $f(S)$
  $trenutnoNajboljšaRešitev \leftarrow urejeniSeznam_N$
  $skrajšaj(urejeniSeznam, \lceil\rho N\rceil)$
  if $f(trenutnoNajboljšaRešitev) > f(najboljšaRešitev)$ **then**
    $najboljšaRešitev \leftarrow trenutnoNajboljšaRešitev$
    iteracije brez izboljšave $\leftarrow 0$
  else
    iteracije brez izboljšave $\leftarrow$*iteracije brez izboljšave*$+1$
  end if
  $posodobiVerjetnosti(urejeniSeznam, M)$
end while

6.4.1 Problemi pri povzporejanju na GPE

Algoritem 7 lahko razdelimo na dva dela. Prvi del je tvorjenje in ocenjevanje rešitev z uporabo matrike M. Drugi del je dodajanje najboljših rešitev v seznam in posodobitev matrike M. Za potrebe prvega dela je matrika M konstantna in zanjo lahko uporabimo strukturirani pomnilnik. Načeloma bi lahko uporabili nespremenljivi pomnilnik, vendar so take matrike pogosto prevelike. Prvi del lahko izvajamo popolnoma na GPE, vendar ostane vprašanje hrambe rešitev odprto.

Preden se lotimo problema hrambe rešitev, si poglejmo povzporejanje drugega dela metode — drugi del bo namreč potreboval te rešitve. Najpreprosteje je, če drugi del opravimo na CPE. Če to počnemo na CPE, moramo prenesti vse rešitve iz pomnilnika GPE v glavni pomnilnik. To se sicer ne zdi najbolj smiselen način povzporejanja za GPE in bi

morda privedel celo do slabšega izvajalnega časa kot na CPE. Oglejmo si še možne alternative za ta korak.

V kolikor bi želeli izvajati drugi del metode vzporedno, bi morali posodobitev verjetnostne porazdelitve (t.j. matrike M) izvajati vzporedno. Če bi se za to odločili, bi morali najboljše rešitve še vedno hraniti v globalnem pomnilniku — povzporejanje tega koraka namreč zahteva globalno uskladitev, kar pomeni končanje izvajanja programskega jedra. Glede na to, da ima deljeni pomnilnik življensko dobo bloka (in se le-ta konča, kadar se konča programsko jedro), bi morali vse rešitve premestiti v globalni pomnilnik. To pa nas ponovno pripelje do zgornjega problema.

Hrambi rešitev v globalnem pomnilniku se torej ne moremo izogniti. Kljub temu smo se uspešno izognili prenosu podatkov iz GPE v glavni pomnilnik ob vsaki iteraciji, vendar ne povsem. Ker je potrebno ocenjene rešitve razporediti po vrstnem redu glede na kakovost, moramo le-te prenesti v glavni pomnilnik, kjer jih CPE nato razporedi in prenese nazaj na pomnilnik GPE. Zaradi omenjenih problemov se opisana metoda navzkrižne entropije ne sklada dobro z zmogljivostmi GPE.

6.4.2 Povzporejanje

Povzporejanje metode navzkrižne entropije je podano v [28], kjer so vzporedni program pognali v okolju MPI. Slednje so počeli na več-procesorskih arhitekturah, ki premorejo deljeni pomnilnik. Problem, ki so ga reševali, je problem največjega reza (ang. *Maximum cut problem*), ki ga tukaj ne bomo opisali.

Povzporejali so prvi del metode, t.j. postopek tvorjenja rešitev z uporabo matrike M, saj se lahko več instanc tega koraka izvaja popolnoma neodvisno. Naj bo s število CPE in N število tvorjenih vzorcev, potem lahko vsak CPE tvori $\frac{N}{s}$ vzorcev. Postopka posodobitve matrike M niso povzporejali. Pri posodobitvi matrike M so uporablili samo en CPE, ki posodobi matriko in jo razširi na ostale CPE. Pri tem moramo opomniti, da CPE, ki izvaja posodobitev prejme vzorce od vseh ostalih CPE. Ta način povzporejanja se imenuje SRIP (ang. *Single replication in parallel*).

Ker so zahteve komunikacije v SRIP postopku zelo velike, so opišali še alternativen način povzporejanja. Na posameznih CPE so pognali statistično neodvisne instance metode navzkrižne entropije, kar zmanjša potrebo po komunikaciji. Ta način povzporejanja se imenuje MRIP (ang. *Multiple replication in parallel*).

V primerjavi SRIP z MRIP za problem največjega reza s parametri, ki dajo enak izvajalni čas za oba algoritma, se izkaže, da SRIP vrne boljše rezultate.

6.5 Optimizacija s kolonijo mravelj

Opisali bomo osnovni sistem mravelj[12][18] vendar je potrebno poudariti, da obstajajo razširitve, kot npr. sistem kolonije mravelj[13] in $\mathcal{MAX} - \mathcal{MIN}$ sistem mravelj[14], ki v tej nalogi ne bodo opisane.

Sistem mravelj deluje tako, da požene na grafu $G(V, E)$ več umetnih mravelj z nalogo iskanja najkrajše poti. Problem, ki ga rešuje sistem mravelj je iskanje najkrajše poti v grafu. Zato moramo vse druge probleme najprej pretvoriti v to obliko, če hočemo uporabiti

to metahevristiko. Sistem mravelj posnema metodo, ki jo uporabljajo mravlje pri iskanju najbolj optimalnih poti (npr. pot od mravljišča do nahajališča hrane) in sicer s komunikacijo preko feromonskih sledi.

Mravlje postopoma gradijo rešitev s premikanjem po grafu. To počnejo stohastično, na podlagi feromonskega modela. Feromonski model ni nič drugega, kot množica parametrov vezanih na povezave grafa, ki jih lahko mravlje spreminjajo tekom algoritma. Za lažje razumevanje si poglejmo primer uporabe sistema mravelj na problemu trgovskega potnika, povzetem po [16].

Vzemimo problem trgovskega potnika s 4 mesti. Nato vpeljimo še feromonske in hevristične vrednosti za vsako povezavo v grafu. Feromonske vrednosti povezav predstavljajo mehanizem, ki ga spreminjajo mravlje in dovoljuje postopno izboljševanje poti. Hevristične vrednosti povezav so odvisne od posameznega problema. Pri problemu trgovskega potnika so hevristične vrednosti enake obratu uteži (razdalj) posameznih povezav. Ob začetku algoritma običajno nastavimo feromonske vrednosti na 1.

Algoritem se nadaljuje tako, da poženemo več mravelj, ki bodo gradile rešitve. Posamezno mravljo poženemo iz naključno izbranega vozlišča (mesta). Na vsakem koraku algoritma se mravlja premakne po eni izmed povezav vozlišča kjer se nahaja. Mravlja izbere tiste povezave, ki je ne peljejo do že prej obiskanih vozlišč. Med temi povezavami mravlja izbere z največjo verjetnostjo tisto povezavo, ki ima visoko tako hevristično, kot tudi feromonsko vrednost. Ko posamezne mravlje obiščejo vsa vozlišča v grafu, izvedemo posodobitev feromonskih vrednosti.

Feromonske vrednosti posodobimo tako, da vse feromonske vrednosti najprej zmanjšamo za določen faktor (temu pravimo feromonsko izparevanje). Kakovost rešitve izračunamo z uporabo funkcije f, ki smo jo definirali v poglavju Kombinatorična optimizacija. Feromonsko vrednost vsake povezave nato povečamo na podlagi kakovosti rešitev, katerim pripada: vsaka mravlja namreč vrne eno rešitev (ko obišče vsa vozlišča grafa). Zato je število rešitev, katerim pripada povezava , enako številu mravelj, ki so povezavo izbrale kot komponento svojega obhoda. Celoten postopek ponavljamo, dokler ne pridemo do zadovoljive rešitve. Sistem mravelj torej lahko opišemo s poenostavljenim algoritmom prikazanem v algoritmu 8.

Algoritem 8 Poenostavljen sistem mravelj.

parametri, kakovost, željena kakovost $\leftarrow$ *inicializiraj*
while *kakovost* $<$ *željena kakovost* **do**
  poženi mravlje
  posodobi feromone
  kakovost $\leftarrow f(najboljšaRešitev)$
end while

Za sistem mravelj je torej potrebno definirati sledeče vrednosti:

- faktor, ki določi količino izparevanja
- način posodobitve feromonov
- pravilo za izbor povezav na podlagi feromonskih sledi

Konstrukcija rešitev

Brez izgube splošnosti se omejimo na eno mravljo. Vzemimo graf $G(V, E)$, ki definira problem trgovskega potnika. Definiramo trenutno nahajališče mravlje in sicer $p \in V$, ki je na začetku inicializirano na neko vozlišče v grafu G, nato pa se spreminja vsakič, ko mravlja doda komponento rešitvi (saj se mravlja premakne). Mravlja začne z rešitvijo, ki jo predstavlja prazna množica $S = \emptyset$. Definiramo mehanizem, ki nam vrne sosednja vozlišča trenutne rešitve $N(S, p) \subset V$. Sosednja vozlišča trenutne rešitve niso nič drugega, kot vozlišča, ki so sosednja vozlišču, kjer se trenutno nahaja mravlja, vendar ne vključujejo že prej obiskanih vozlišč. Ob vsakem premiku do nekega vozlišča $v \in N(S, p)$ mravlja doda trenutni rešitvi par (p, v) in nastavi $p = v$.

V primeru, ko imamo več možnih sosedov, se med njimi odločimo na podlagi nekega pravila za izbor. Definiramo matriko $ph_{v,w}$, ki hrani feromonsko sled med vozliščema $v, w \in V$. Definiramo tudi matriko $h_{v,w}$, ki vrne hevristično vrednost med dvema vozliščema. Eno izmed najbolj znanih [16] pravil izbora je pravilo sistema mravelj [18], ki določa izbor na podlagi verjetnosti in sicer

$$Pr(p, v, S) = \frac{ph_{p,v}^{\alpha} \cdot h_{p,v}^{\beta}}{\sum_{\bar{v} \in N(S,p)} ph_{p,\bar{v}}^{\alpha} \cdot h_{p,\bar{v}}^{\beta}},$$

kjer so $\alpha, \beta \in \mathbb{R}$ realna števila, ki določajo relativno pomembnost feromonov in hevristike, p predstavlja trenutno pozicijo mravlje v grafu, S predstavlja trenutno rešitev ter $v \in N(S, p)$ predstavlja kandidata za vključitev v rešitev. Mravlja dodaja vozlišča z uporabo zgornjega pravila dokler ne obišče celotnega grafa (to se zgodi natanko tedaj, ko $N(S, p) = \emptyset$).

Posodobitev feromonov

Naloga postopka posodobitve feromonov je pravzaprav podpiranje boljših rešitev in oviranje slabših. Kot smo že omenili, sestoji iz dveh delov: izparevanje feromonov in povečevanje feromonskih vrednosti, povezanih z ugodnimi rešitvami. Definiramo $\rho \in (0, 1]$, t.j. parameter, ki določa hitrost izparevanja. Definiramo še množico vseh trenutnih rešitev $Z \subset V$ (vsaka posamezna mravlja prispeva eno rešitev).

Posodobitev feromonov je dana z

$$ph_{v,w} \leftarrow (1 - \rho) \cdot ph_{v,w} + \rho \cdot \sum_{S \in Z | (v,w) \in S} f(S).$$

Feromonsko izparevanje nudi način raziskovanja novih področij znotraj iskalnega območja. Različni algoritmi za optimizacijo s kolonijo mravelj se razlikujejo prav v metodi posodobitve feromonov.

6.5.1 Problemi pri povzporejanju na GPE

Nespremenljivi pomnilnik je idealen za hrambo samega grafa, vendar ga hitro zmanjka, če imamo kakšen večji graf. Zato je smiselneje uporabiti strukturirani pomnilnik, ki je tudi zelo učinkovit. Še bolj problematičen je postopek posodobitve feromonske matrike, saj je to, v primeru, da poganjamo več mravelj, kritično območje. Lahko poganjamo več neodvisnih instanc algoritma, vendar je tako povzporejanje naivno. Bolj smiselno bi bilo, če bi lahko poganjali samo eno instanco algoritma in več mravelj. Prav to možnost si bomo pogledali.

Predstavljajmo si, da poganjamo samo eno skupino mravelj, ki gradi skupno rešitev. Problem nastopi pri kritičnem območju, in sicer pri posodobitvi feromonov. Tu bi bilo potrebno uvesti globalno uskladitev — vendar implementacija le-te na strojni opremi ne obstaja, zato bi moral algoritem na GPE zaključiti delo ob prvem koraku (grajenju rešitev). Zaradi tega razloga bi bila potrebna razdelitev algoritma na dva dela, kjer oba potekata na GPE, vendar v obliki dveh različnih programskih jeder: prvo programsko jedro požene mravlje, dobi množico rešitev Z in zaključi izvajanje, nato se požene drugo programsko jedro, ki feromone vzporedno posodobi. Nato se postopek ponovi.

To nam olajša zahteve do hrambe feromonov, saj so le-ti konstantni kar zadeva izvajanja prvega dela algoritma. Ker poganjamo večje število mravelj, pomeni, da vsaka mravlja vrne eno rešitev — velikost slednje omejuje število niti, ki jih lahko poženemo v enem bloku, če hočemo hraniti podatke v deljenem pomnilniku. V tem primeru bi lahko ostale spremenljivke hranili v registrih, feromonske sledi in strukturo grafa pa v nespremenljivem ali strukturiranem pomnilniku. Opomniti je tudi treba, da bi lahko hranili rešitve v globalnem pomnilniku, saj do njih ne dostopamo pogosto in tako omogočili poganjanje veliko večjega števila niti in blokov.

Ostane nam samo še implementacija metode posodobitve feromonov. Če želimo metodo implementirati na GPE, nam predstavitev rešitev v obliki seznama vozlišč predstavlja oviro, saj ponovno nastopi problem kritičnega območja. Rešitve lahko hranimo tudi v matrični obliki, t.j. matrikah Ph_i dimenzij $N \times N$, ki vsebujejo vrednost 1 za povezave, ki jih je mravlja i prehodila in 0 drugod. V tem primeru lahko posodobitev izvedemo vzporedno na GPE tako, da vsaki niti dodelimo lastno območje v matrikah, za katerega izvede operacijo seštevanja.

V poglavju Rezultati si bomo podrobnejše ogledali ta način povzporejanja in podali psevdokodo.

6.5.2 Povzporejanje

Povzporejanje sistema mravelj za problem proizvodnega razporejanja je opisano v [25]. Vzporedni program so pognali v okolju *OpenMP* z arhitekturo, ki omogoča deljen pomnilnik. Implementacija v [25] temelji na povzporejanju metode tvorjenja rešitev in ocenjevanja le-teh, saj zavzamejo večino izvajalnega časa algoritma. Problem industrijskega razporejanja

je bil najprej preveden v obliko trgovskega potnika, za katero je enostavno uporabiti sisteme mravelj.

Pri vzporedni implementaciji nastopi problem uskladitve. In sicer, ko vse mravlje končajo delo, je potrebno vpeljati uskladitev, da se lahko začne postopek posodobitve feromonov. Korak posodobitve se mora namreč izvesti šele potem, ko je korak tvorjenja rešitve že izveden. Ta problem rešijo z uporabo pregrade.

Povzporejanje izvedejo tako, da posameznemu procesorju dodelijo več mravelj. Feromone hranijo v obliki matrike, ki je v času koraka tvorjenja rešitev, dostopna samo v načinu branja. Ob zaključku koraka, glavna nit izvede posodobitev feromonske matrike. Matriko razdalj (reprezentacijo grafa) hranijo v deljenem pomnilniku, saj se le-ta ne spreminja v času izvajanja algoritma.

V [25] podajo še rezultate, ki prikazujejo skoraj 2-kratno pospešitev z uporabo dveh procesorjev, nekoliko več kot 3-kratno pospešitev z uporabo štirih procesorjev, vendar pospešitve z uporabo 8 in 16 procesorjev niso več tako dramatične.

6.6 Iskanje s prepovedmi

Iskanje s prepovedmi (ang. *tabu search*, *TS*)[19] je optimizacijska metoda, ki pripada razredu metod lokalnega iskanja. Iskanje s prepovedmi izboljša običajno delovanje metod iz tega razreda tako, da z dodatkom pomnilnika dovoli, da metoda ne obišče iste rešitve večkrat. Iskanje s prepovedmi je ena izmed najbolj citiranih in najpogosteje uporabljenih metod za reševanje problemov kombinatorične optimizacije[6].

Pomnilnik omogoča izogibanje lokalnim minimumom in ciklom. Pomnilnik je ponavadi implementiran v obliki seznama prepovedi, ki hrani nazadnje obiskane rešitve in prepreči premike, ki bi vodile v njih. Drugače povedano, seznam prepovedi omejuje možne sosednje rešitve na tiste, ki niso v seznamu. Za izračun kakovosti rešitve uporabimo funkcijo f, ki smo jo definirali v poglavju Kombinatorična optimizacija. Ob vsaki iteraciji algoritma je izbrana najboljša izmed sosednjih rešitev (na podlagi f) in ta postane *trenutna rešitev*. Ta rešitev je nato dodana v seznam prepovedi. Ker je seznam prepovedi omejene velikosti, je ponavadi najstarejša rešitev v seznamu odstranjena ob tem koraku. Algoritem konča, če se izpolni pogoj za končanje ali, če je množica sosednjih rešitev prazna (v primeru, da smo obiskali že vse sosednje rešitve). Algoritem 9 predstavlja poenostavljeno različico metode.

Algoritem 9 Poenostavljena metoda iskanja s prepovedmi.

```
rešitev, kakovost ←inicializiraj
while kakovost < željena kakovost do
  sosedi ←sosednjeRešitve(rešitev)
  if sosedi = ∅ then
    končaj
  end if
  rešitev ← najboljša(sosedi)
  kakovost ← f(rešitev)
  dodajVSeznamPrepovedi(rešitev)
end while
```

Velikost seznama ima velik vpliv na delovanje metode. Večje velikosti seznama povzročijo preiskovanje večjega prostora rešitev, saj so vse lokalne izboljšave že prisotne v seznamu prepovedi. Podobno, manjši seznam ima učinek preiskovanja manjšega, lokalnega prostora rešitev. Nekatere izboljšave [20, 21, 22, 23] metode iskanja s prepovedmi dinamično spreminjajo velikost seznama prepovedi in tako vodijo do robustnejših algoritmov[6]. V nadaljevanju se bomo omejili na metodo iskanja s prepovedmi za problem trgovskega potnika.

6.6.1 Problemi pri povzporejanju na GPE

Pri iskanju s prepovedmi moramo najprej sprejeti odločtev, kaj bomo hranili v seznamu prepovedi. Hramba celotnih sosednjih rešitev bi bila nesmiselno zahtevna do pomnilnika, zato se bomo odločili za hrambo posameznih premikov oblike $(i, j), i, j \in \mathbb{N}$, ki predstavljajo zamenjavo mesta i z mestom j v naši rešitvi. Z dolžino seznama se ne bomo posebej ukvarjali.

Nato se moramo vprašati, kako bomo povzporejanje organizirali — kje naj se kaj izvaja. Poganjanje več instanc algoritma na GPE bi bilo nesmiselno, saj tu ne gre za stohastično metodo in bi posledično dobili enake rešitve povsod. Pognati moramo torej eno instanco algoritma s smiselnim načinom povzporejanja. Pri iskanju s prepovedmi nastopi nekaj območij, ki zahtevajo uskladitev:

- izbor najboljšega soseda: samo eden izmed procesov naj izbere najboljšega soseda. Drugi naj čakajo.
- dodajanje nove rešitve v seznam: samo eden izmed procesov naj doda novo rešitev v seznam. Drugi naj čakajo.

Kot vemo, je globalno uskladitev nemogoče izvesti na kakršenkoli drugi način kot je, da program prekinemo in ukaze izvedemo na CPE. Dodajanje nove rešitve v seznam je preprosto in to lahko počne CPE (v kolikor je seznam smiselnih velikosti, da ne povzroči zakasnitve zaradi prenosa podatkov iz GPE na glavni pomnilnik in obratno).

Izbor najboljšega soseda pa lahko povzroči zakasnitev, saj je polje sosedov lahko zelo veliko in bi prišlo do velike zakasnitve ob prenosu podatkov. Vsaj nekolikšen del tega koraka moramo izvesti na GPE, da prenos podatkov omejimo. Polje sosedov lahko razdelimo na N enako velikih disjunktnih delov in poženemo N niti z namenom iskanja najboljšega kandidata v posameznem delu. Tako zmanjšamo velikost polja za faktor N, kar pomeni, da ne skrajšamo le izvajalni čas, ki ga bo porabil CPE za izbor najboljšega soseda, temveč tudi čas, ki ga bomo porabili za prenos podatkov iz GPE na glavni pomnilnik.

Tu nastopi še en problem. In sicer, kako naj izvedemo zgornji korak, kateri se mora izvesti na koncu koraka ocenjevanja sosedov. Globalne uskladitve tukaj ne moremo uporabiti, saj se ji hočemo izogniti. Uporabiti moramo bločno uskladitev. Pri bločni uskladitvi lahko uskladimo le niti v istem bloku. To pomeni, da moramo najprej razdeliti polje sosedov na $N =$*število blokov*. To nam dovoli, da lahko bloki, ki hitreje končajo delo, takoj začnejo postopek izbora najboljšega soseda. Vsak izmed N delov nato razdelimo na M

enako velikih, disjunktnih delov, na katerih bomo pognali posamezne niti v bloku. V kolikor to želimo, je predpogoj, da del i, ki pripada določenemu bloku že vsebuje potrebne podatke (t.j. kakovosti posameznih sosedov). To bomo morali pomniti, ko bomo povzporejali preostali del metode iskanja s prepovedmi.

Preostane nam torej še postopek tvorjenja sosedov rešitve in ocenjevanja le-teh. Enega izmed območij, ki zahteva uskladitev še nismo omenili, in sicer je to problem tvorjenja sosedov ter posledičnega ocenjevanja. Vprašanje, ki se izpostavi je namreč: kako naj nek proces zazna, da je postopek tvorjenja sosedov zaklju"en in naj sedaj izvede ocenjevanje. Kot smo omenili že zgoraj je predpogoj, da izvajamo ta proces lokalno za posamezen blok, kar nam omogoči uporabo bločne uskladitve med tema dvema korakoma.

Oba koraka moramo smiselno ločiti na disjuntkne dele. V primeru problema trgovskega potnika dobimo polje sosedov z izvedbo zamenjav vseh parov v ciklu dolžine $N + 1$, kjer je N število mest. Pri tem nam zamenjava mesta i z mestom j vrne enakega soseda kot zamenjava mesta j z mestom i. Vsakemu bloku dovolimo tvorjenje določenega deleža zamenjav pri čemer nočemo, da bi katerikoli drugi blok tvoril zamenjave, ki vključujejo iste elemente. Vsak blok naj nato shrani svoje zamenjave na lastni prostor v pomnilniku. Enako povzporejanje izvedemo za niti v posameznem bloku. Ob vsaki možni zamenjavi mora proces preveriti, ali seznam prepovedi izključuje ta premik. Ko posamezna nit v bloku izvede korak tvorjenja sosedov, lahko takoj oceni svoje sosede (to sicer lahko počne tudi hkrati). V primeru, da ohranimo delitev podatkov tudi v koraku izbire najboljšega soseda, ne potrebujemo nobenega uskladitvenega koraka — vendar to pomeni, da bo polje sosedov veliko in bo zato slabša pospešitev.

Osredotočimo se sedaj na hrambo podatkov. Pri iskanju s prepovedmi moramo hraniti trenutno rešitev, trenutni seznam prepovedi, reprezentacijo grafa, soseščino rešitve in oceno posameznih sosedov. Reprezentacija grafa je konstantna in je zato idealna za hrambo bodisi v nespremenljivem ali strukturiranem pomnilniku — če hočemo obdelati kakršnekoli večje probleme, bomo tu morali posegati po strukturiranem pomnilniku.

Za našo metodo povzporejanja, ki je opisana zgoraj, je smiselna hramba trenutne rešitve v strukturiranem pomnilniku. Le CPE bo izbral novo rešitev in jo naložil, nato pa bodo le-to vse niti morale brati za postopek tvorjenja sosedov. Enako velja za seznam prepovedi, saj bo vanj dodajal samo CPE, brale pa ga bodo vse niti.

Za hrambo polja sosedov (premikov) in povezanih ocen bi želeli uporabiti deljeni pomnilnik, saj se bo tukaj izvajala večina operacij. Posamezen sosed je oblike (i, j, k), kjer i, j predstavlja zamenjavo mest in k, ki predstavlja oceno rešitve po zamenjavi. Če uporabimo deljeni pomnilnik z uporabo delitve, kot je opisana zgoraj in uporabimo za predstavitev posameznega soseda vektorski tip ***ushort4*** potem je velikost posameznega elementa 8 bajtov. Na posameznem multiprocesorju je možno hraniti do 2000 sosedov, kar je zelo veliko število še posebej, če vključujemo ostale multiprocesorje.

6.6.2 Povzporejanje

Izkaže se, da je metoda iskanja s prepovedmi na GPE že bila implementirana in sicer v [24]. Vendar je to bila implementacija z uporabo orodji za obdelavo grafike in ne z orodji za splošno računanje, kot je CUDA. Implementacija opisana v [24] uporablja enostavne prijeme povzporejanja, saj je uporaba grafičnih metod manj primerna splošnemu programiranju.

To implementacijo bomo bolj podrobno opisali, prej pa bomo opisali še nekaj podrobnosti. Program, ki se je izvajal na CPE je bil implementiran v jeziku C#, le-ta je bil uporabljen za primerjavo izvajalnega časa. Program, ki se je izvajal tako na GPE, kot CPE (opisano spodaj), je bil napisan v jeziku C#.

Oglejmo si implementacijo opisano v [24]. Metoda je inicializirana na naključni rešitvi (t.j. naključno dobljenem seznamu obiskanih mest). Soseščina je definirana kot zamenjava dveh mest (enega, ki je vsebovan z enim, ki ni vsebovan). Definirajmo posamezno rešitev, kot seznam obiskanih mest π, urejen po vrstnem redu obiska. $\pi(i), i \in \mathbb{N}$ naj predstavlja i-ti element rešitve (t.j. mesto). Seznam prepovedi hrani premike v obliki urejenega para $\big(\pi(i), \pi(j)\big), i \neq k$, kjer i predstavlja vsebovano mesto in j predstavlja mesto, s katerim bi ga zamenjali. Seznam prepovedi ne dovoli rešitev, ki vključujejo zamenjave, ki jih vsebuje seznam prepovedi. Dolžina seznama prepovedi je konstantna in sicer 7. Metoda se ustavi po 1000 iteracijah.

Največ časa implementacije porabi ocenjevanje kakovosti sosednjih rešitev, t.j. izračun dolžine poti vsake rešitve v soseščini, ki predstavlja 90% celotnega izvajalnega časa [24]. Ker je to preprosto izvesti vzporedno, je to objekt povzporejanja. V implementaciji naložijo na GPE tri slike.

- Prva slika je velikosti $N \times N$ pikslov kjer je na koordinatah (i, j) razdalja d_{ij} (t.j. razdalja med dvema mestoma i in j). Ta slika vsebuje delež potrebnih podatkov za izračun kakovosti sosednjih rešitev.

- Druga slika, tudi velikosti $N \times N$ pikslov, predstavlja soseščino — t.j. vsaka koordinata predstavlja en premik v obliki urejenega para (i, j). Ta slika predstavlja drugi delež potrebnih podatkov.

- Tretja slika je velikosti $N \times 1$ in predstavlja trenutno rešitev (permutacijo mest). Predstavlja tretji delež potrebnih podatkov in sicer, v kombinaciji z drugo sliko lahko dobimo celotne permutacije mest, ki jim ustrezajo sosednje rešitve. Če nato uporabimo še prvo sliko lahko izračunamo razdalje (kakovosti) vseh sosednjih rešitev.

Ob prejemu slik, GPE izračuna dolžino vseh poti v soseščini z uporabo podanih podatkov v obliki slik, kot je to opisano zgoraj. Ostali deli algoritma se izvajajo na CPE.

Pospešitev algoritma na GPE je bila 12% za 100 mest. Kot razlog za to, navajajo zakasnitev, ki nastopi ob nalaganju podatkov na GPE (to namreč počnejo ob vsaki iteraciji) in relativno preprostost izračunov, ki jih delegirajo GPE. Boljše rezultate (16-kratna pospešitev) so dobili za problem razporejanja. Ponovno moramo povdariti, da je tukaj prisotnih veliko omejitev, ki nastopijo zaradi uporabe grafičnih metod, na katere opozarjajo tudi avtorji članka.

6.7 Evolucijski algoritem

Evolucijski algoritem (ang. *evolutionary algorithm*) je metahevristika, ki črpa ideje iz teorije evolucije. Podobno kot sistemi mravelj, katere smo si že ogledali in temeljijo na populaciji

mravelj, ki gradijo rešitev, tudi evolucijski algoritem temelji na populaciji entitet, vendar z razliko, da so le-te že same po sebi rešitve. Evolucijski algoritem tako hrani množico rešitev oz. populacijo, na kateri izvaja razne oblike sprememb, kot so npr. mutacije in parjenje rešitev. Posameznemu mehanizmu spremembe pravimo operator.

Pogosto lahko operatorje, ki jih uporabljajo evolucijski algoritmi, razdelimo v dva razreda. V prvi razred spadajo operatorji, ki se izvajajo na posamični rešitvi, iz katere sledi nova rešitev. Primer takih operatorjev sta mutacija (ang. *mutation*) in modifikacija (ang. *modification*). V drugi razred spadajo operatorji, ki iz dveh ali več rešitev naredijo novo rešitev, katera vključuje neke lastnosti rešitev, ki so bile uporabljene za izgradnjo nove. Primer takih operatorjev sta križanje (ang. *crossover*) in rekombinacija (ang. *recombination*).

Evolucijski algoritmi periodično posegajo v populacijo z namenom odstranitve najslabših rešitev tako, da se parjenje omogoči samo med boljšimi rešitvami. To je nekakšna analogija preživetju najbolj prilagodljivih, ki se izvaja v naravi.

V nadaljevanju, se bomo omejili na probleme kombinatorične optimizacije in sledili opisu evolucijskih algoritmov, kot je ta podan v [6, 29]. Ni nujno, da populacija predstavlja celotne rešitve. Pogosto populacija predstavlja delne rešitve, ki jih lahko z uporabo določenega mehanizma kombiniramo. Označimo P_t populacijo v času t — to pravzaprav predstavlja iteracijo t. V vsaki iteraciji evolucijskega algoritma izvedemo nekaj operatorjev nad trenutno populacijo P_t in jih ocenimo z uporabo funkcije f, ki smo jo definirali v poglavju Kombinatorična optimizacija. Nato izberemo nekaj osebkov $P_{t+1} \subset P_t$, na podlagi kakovosti (boljše osebke z večjo verjetnostjo) na katerih nato izvedemo operatorje, povečamo t in postopek ponovimo). Skelet algoritma je predstavljen v algoritmu 10, povzetem po [6].

Algoritem 10 Skelet evolucijskih algoritmov.

$P, kakovost \leftarrow inicializiraj$
while $kakovost <$ *željena kakovost* **do**
  $P' \leftarrow rekombiniraj(P)$
  $P'' \leftarrow mutiraj(P')$
  if $najboljši(f(P'')) > kakovost)$ **then**
    $kakovost \leftarrow najboljši(f(P''))$
  end if
  $P \leftarrow izbor(P'' \cup P)$
end while

Običajno predstavimo osebke kot bitne nize ali kot permutacije n celih števil, vendar obstaja še veliko različnih pristopov in je pravilna predstavitev ključna za posamezen problem za dobro delovanje algoritma. Kot smo omenili, se ob vsaki iteraciji izvaja proces selekcije. Pogosto se populacijo omeji na določeno fiksno število, s tem, da vedno hranimo najboljšega izmed osebkov (torej je vedno izbran z verjetnostjo 1.0). Kot možen pogoj za končanje izvajanje algoritma je tudi izumrtje populacije, kjer v neki iteraciji t v populaciji ostane le še trenutno najboljši osebek.

Način združevanja dveh osebkov pri tvorbi novega osebka lahko omejimo na določeno podmnožico osebkov. V primeru, da ta omejitev obstaja, govorimo o strukturirani populaciji P, v nasprotnem primeru pa o nestrukturirani populaciji P. Omeniti moramo še

problem nekompatibilnih operacij, npr. če parjenje osebkov x in y vrne osebka z, ki ni več smiselna rešitev ali del rešitve problema. V takem primeru lahko takšna parjenja zavrnemo, vendar se lahko zgodi, da za preverjanje porabimo preveč časa. Ta problem lahko rešimo v postopku ocenjevanja, kjer takim osebkom damo slabšo oceno.

6.7.1 Problemi pri povzporejanju na GPE

Problemi, ki se pojavijo pri povzporejanju evolucijskih algoritmov na GPE, so predvsem kritična območja. V kolikor razdelimo populacijo na N disjunktnih delov, kjer vsaka nit obdela samo enega izmed teh delov, nastopi problem mešanja populacije. Tako implementacijo imenujemo pristop z uporabo otokov[31] in je eden izmed najpogostejših pristopov povzporejanja evolucijskih in genetskih algoritmov.

Tak pristop ima veliko skupnega s pristopi poganjanja več instanc algoritmov, pri čemer ob vsaki iteraciji razdelimo najboljšo trenutno rešitev med vse instance. Tu se pojavijo problemi mešanja populacij, saj ta problem ni preprosto rešljiv. Pristop z uporabo otokov bomo opisali v naslednjem podpoglavju.

Osredotočimo se sedaj na problem povzporejanja evolucijskega algoritma brez uporabe otokov. Potrebujemo določen seznam osebkov, t.j. populacijo P. Evolucijski algoritem moramo razdeliti na dva dela: izvedba operatorjev na populaciji in posodobitev (selekcija) te populacije.

Populacijo je smiselno obravnavati kot konstatno za potrebe prvega dela algoritma. Lahko jo hranimo v strukturiranem pomnilniku. Tvorjenje posameznih osebkov z uporabo operatorjev poteka na nivoju niti, ki nove osebke ocenijo in nato prenesejo v globalni pomnilnik. Ko se prvi del algoritma konča, potrebujemo globalno uskladitev, zato končamo izvajanje programskega jedra.

Sedaj moramo vrnjene osebke integrirati v populacijo z uporabo selekcije. Tu se stvari zapletejo, saj običajno potrebujemo razvrščanje osebkov na podlagi kakovosti in je težko vpeljati kakršnokoli smiselno povzporejanje. Operacijo bi morali izvajati v tem primeru na CPE, kar zahteva prenos podatkov na glavni pomnilnik ob vsaki iteraciji. V primeru evolucijskega algoritma je teh podatkov lahko zelo veliko .

Kot lahko vidimo, je povzporejanje osnovne oblike tega algoritma na GPE nekoliko nenaravno. Bolj smiseln, vendar nekoliko zahtevnejši pristop bomo opisali v naslednjem podpoglavju.

6.7.2 Povzporejanje

Povzporejanje enega izmed evolucijskih algoritmov – genetskega algoritma – je bilo že izvedeno na GPE in sicer v [30]. Za vzporedno izvajanje so uporabili CUDA. Pri povzporejanju so uporabili model otokov, t.j. populacij, ki so medseboj ločene in je mešanje osebkov med otoki manj pogosto. Posamezen otok so implementirali kot blok, saj otok zahteva pogosto izvajanje operatorjev na osebkih in uskladitev, kar nudi deljeni pomnilnik bloka.

V posamičnem bloku so torej pognali genetski algoritem, kjer so kritična območja (selekcija, ipd) reševali s pregrado. Ob vsaki iteraciji algoritma je bila izvedena migracija na drugi otok (mešanje osebkov) na podlagi verjetnosti in kakovosti osebkov. Nato so bili izvedeni procesi selekcije, križanja in mutacije. Potek algoritma je prikazan v algoritmu 11.

Algoritem 11 Genetski algoritem na GPE.

```
P ←inicializiraj
while !konec do
  kvaliteta ← f(P)
  if migracija(kvaliteta) then
    migriraj
  end if
  P ← selekcija(P)
  P ← križaj(P)
  P ← mutiraj(P)
end while
```

Migracija se izvaja na nivoju globalnega pomnilnika, saj je to edini možni način komunikacije med bloki. Uporabljen način komunikacije je popolnoma neusklajen, vendar kljub temu učinkovit pri mešanju populacij iz različnih otokov. Migracija se izvaja na sledeč način: vsak otok razvrsti svoje osebke in N najboljših naloži v globalni pomnilnik. Vsak otok si tudi zapomni N najslabših osebkov in jih nato zamenja z N najboljšimi osebki določenega drugega otoka. Uporabljena metoda vzporednega razvrščanja je Bitonic-Merge sort.

Avtorji poročajo o do 2600-kratni pohitritvi v primerjavi z optimizirano implementacijo na CPE, pri čemer je kakovost dobljenih rešitev primerljiva s CPE implementacijo. Uporabljena oprema je bila GeForce 8800 GTX GPE in Intel Core i7 920, 3.2 GHz CPE.

Poglavje 7

Rezultati

Za testiranje smo uporabili primerke DIMACS (ang. *Center for Discrete Mathematics and Theoretical Computer Science*)[15]. Uporabljeni primerki DIMACS so sicer za problem največjega polnega podgrafa, vendar jih z izvedbo komplementa lahko uporabimo za problem največje neodvisne množice. Za problem trgovskega potnika smo uporabili primerke TSPLIB [37].

Testiranje je bilo izvedeno na Nvidia GeForce 9600 GPE, ki vsebuje 64 procesorjev in AMD Athlon 64 3500+ CPE z 2 GB pomnilnika na operacijskem sistemu Ubuntu 10.04.

Vzporedne različice algoritmov smo implementirali v Nvidinem okolju, CUDA. Za CUDA smo se odločili, ker omogoča uporabo struktur in kazalcev. V kolikor bi se odločili za OpenCL, bi morali implementaciji podani v [32], ki uporabljajo tako strukture kot tudi kazalce, znatno spremeniti.

7.1 Simulirano ohlajanje

CUDA implementacija simuliranega ohlajanja je zasnovana na implementaciji za porazdeljeno računanje v okoljih MPI in PVM, ki je podana v [32]. Implementacija rešuje problem največje neodvisne množice. Obstoječi implementaciji smo sledili skoraj do potankosti. Povzporejanje je bilo izvedeno tako, da smo pognali več instanc algoritma in vsakih N iteracij razširili trenutno najboljšo rešitev na vse instance. Ker smo za to potrebovali globalno uskladitev, je bil ta del postopka izveden na CPE, nato je bilo programsko jedro ponovno zagnano na GPE.

Predstavitev problema hranimo v nespremenljivem pomnilniku. Rešitve hranimo v deljenem pomnilniku. V tej implementaciji globalni pomnilnika pri računanju ne uporabljamo.

Del algoritma, ki teče na CPE je prikazan v algoritmu 12. Delež, ki se izvaja na GPE, pa je prikazan v algoritmu 13.

Algoritem 12 Del algoritma simuliranega ohlajanja, ki se izvaja na CPE.

```
α, T_min, T_0 ← inicializiraj
Nsinh ← pogostost uskladitve
T ← T_0
while T > T_min do
  poženiProgramskoJedroGPE(α, T, T · α^Nsinh)
  rezultati ← prenesiNajboljšeIzGPE
  GPE ← izberiNajboljšega(rezultati)
  T ← T · α^Nsinh
end while
```

Algoritem 13 Del algoritma simuliranega ohlajanja, ki se izvaja na GPE.

```
funkcija poženiProgramskoJedroGPE(α, T, T_min)
r ← trenutno najboljša rešitev
E ← ocena rešitve r
while T > T_min do
  iter ← 0
  while iter < iter_max do
    s ← kopiraj r
    s ← sosednjaRešitev(r)
    E_s ← oceniRešitev(s)
    E_Δ ← E − E_s
    if P(E_Δ) ≥ rand(0, 1) then
      r ← s
      E ← E_s
    end if
    iter ← iter + 1
  end while
  T ← T · α
end while
```

Ta algoritem se z lahkoto požene na CPE ali v okoljih MPI oz. PVM, ki ga izvedejo relativno hitro v primerjavi z GPE. Razlog za upočasnitev na GPE je prekomerno dostopanje do pomnilnika, kot se to zgodi npr. ob kopiranju rešitve. To prikazuje eno izmed tegob programiranja GPE — algoritmi pisani za računanje na GPE morajo biti tako snovani že od samega začetka. Prekomernemu dostopanju do pomnilnika se je treba izogibati. Kodo je potrebno tudi strukturirati tako, da je v njej čim manj vejitev.

Parametri algoritma so podani v tabeli 7.1 in so enaki parametrom, ki so bili uporabljeni za meritve v [32].

Parameter	Vrednost)
α	0,9995
L	100
T_0	100
T_{min}	1
$Nsinh$	500

Tabela 7.1: Uporabljeni parametri algoritma simuliranega ohlajanja.

Tabela 7.2 prikazuje izvajalni čas algoritma za primerke DIMACS. Zaradi hranjenja podatkov v deljenem pomnilniku, je bila največja velikost primerkov zelo omejena, zato smo lahko izmerili izvajalni čas le za manjše primerke. Meritve za zaporedni, MPI ter PVM algoritem se nahajajo v [32].

Primerek	Čas (sekunde)
brock200_1	151
brock200_2	127
brock200_3	147
brock200_4	161
c-fat200-5	135

Tabela 7.2: Izvajalni čas izbranih DIMACS primerkov.

Število pognanih niti je 1 na blok. Pognanih je bilo 16 blokov. Za število blokov od 1 do 16 ni bilo razlike v času izvajanja. Število blokov od 17 do 32 pa podvoji izvajalni čas. Kljub temu, da uporabljamo deljeni pomnilnik, algoritem porabi ogromno časa. Samo za izvajanje koraka kopiranja rešitev (brez iskanja soseda) porabi do 50 sekund.

7.2 Metoda navzkrižne entropije

Tako kot pri simuliranem ohlajanju, je tudi CUDA implementacija metode navzkrižne entropije zasnovana na implementaciji podani v [32]. Implementacija rešuje problem največje neodvisne množice. Obstoječi implementaciji smo tudi tukaj zelo natančno sledili. Izvorna implementacija je bila, tako kot simulirano ohlajanje, napisana za MPI in PVM okolji in tudi za zaporedno izvajanje. Povzporejanje je bilo izvedeno tako, da smo na GPE pognali neodvisne instance algoritma, ki izvedejo standardno metodo navzkrižne entropije. Nato smo izvajanje na GPE zaključili, prenesli verjetnostne matrike na glavni pomnilnik in jih združili. Združene verjetnostne matrike smo naložili na GPE in ponovno zagnali programsko jedro.

Predstavitev problema hranimo v nespremenljivem pomnilniku. Verjetnostne matrike in rešitve hranimo v globalnem pomnilniku. V tej implementaciji deljenega pomnilnika ne uporabljamo.

Del algoritma, ki teče na CPE, je prikazan v algoritmu 14. Del algoritma, ki teče na GPE, pa je prikazan v algoritmu 15. Parameter α določa glajenje pri posodobitvi verjetnosti na tak način, da je α delež nove verjetnosti združen z $(1 - \alpha)$ deležem stare verjetnosti.

Algoritem 14 Del algoritma metode navzkrižne entropije, ki se izvaja na CPE

$n \leftarrow$ število vozlišč grafa
$c \leftarrow$ število tvorjenih rešitev v odvisnosti od n
$\alpha \leftarrow$ faktor glajenja pri posodobitvi verjetnosti
$N \leftarrow \lceil n \cdot c \rceil$
$\rho \leftarrow$ delež sosedov uporabljenih za posodobitev
$N_{najb} \leftarrow \lceil \rho \cdot N \rceil$
$iter_{maks} \leftarrow$ maks. število iteracij brez izboljšave
$iter \leftarrow 0$
while $iter < iter_{maks}$ **do**
  $poženiProgramskoJedroGPE(N, N_{najb}, \alpha)$
  $verjetnostneMatrike \leftarrow prenesiVerjetnostiIzGPE()$
  $združi(verjetnostneMatrike)$
  $GPE \leftarrow verjetnostneMatrike$
  if $trenutniNajboljšiRezultat > najboljšiRezultat$ **then**
    $najboljšiRezultat \leftarrow trenutniNajboljšiRezultat$
    $iter \leftarrow 0$
  else
    $iter \leftarrow iter + 1$
  end if
end while

Algoritem 15 Del algoritma metode navzkrižne entropije, ki se izvaja na GPE

funkcija $poženiProgramskoJedroGPE(N, N_{najb}, \alpha)$
$i \leftarrow 1$
while $i \leq N$ **do**
  $S_i \leftarrow tvoriSoseda(M)$
  $i \leftarrow i + 1$
end while
$urejeniSeznam \leftarrow S_1, ..., S_N$ urejeni po naraščajočem vrstnem redu po $f(S)$
$skrajšaj(urejeniSeznam, N_{najb})$
$posodobiVerjetnosti(urejeniSeznam, M, \alpha)$

Parametri algoritma so podani v tabeli 7.3 in so enaki parametrom, ki so bili uporabljeni za meritve v [32].

Parameter	Vrednost)
α	0,5
c	0,5
ρ	0,1
$iter_{maks}$	10

Tabela 7.3: Uporabljeni parametri algoritma navzkrižne entropije.

Tabela 7.4 prikazuje izvajalni čas algoritma za primerke DIMACS. Meritve za zaporedni, MPI ter PVM algoritem se nahajajo v [32]. Število pognanih niti je 1 na blok. Pognanih je bilo 16 blokov.

Primerek	Čas (sekunde)
brock200_1	0,923
brock200_2	0,756
brock200_3	0,762
brock200_4	0,769
c-fat200-5	0,759
C1000.9	120

Tabela 7.4: Izvajalni čas izbranih DIMACS primerkov.

Sprva se morda zdijo rezultati nesmiselni, saj navzkrižna entropija uporablja globalni pomnilnik, ki je še toliko počasnejši od deljenega, ki ga uporablja simulirano ohlajanje. Vendar je potrebno pomniti, da algoritem navzkrižne entropije izvaja veliko manj iteracij. Število korakov kopiranja in premika rešitve, ki jih izvede simulirano ohlajanje je podano z

$$L \cdot \frac{\log \frac{T_{min}}{T_0}}{\log \alpha}$$

Za uporabljene vrednosti parametrov to znaša približno 920800 korakov. Metoda navzkrižne entropije pa izvede za zgornje primerke in parametre v povprečju zgolj 11 iteracij, pri čemer ob vsaki iteraciji tvori $n \cdot c$ rešitev in se sprehodi po matriki velikosti $n \times n$.

Tako kot pri simuliranem ohlajanju, se tudi tukaj pojavi problem, da je implementacija algoritma napisana za izvajanje na CPE in se ne poda dobro zmogljivostim GPE, kljub temu pa veliko boljše kot simulirano ohlajanje. Povečano število blokov oz. niti poveča tudi izvajalni čas, vendar nudi določeno pospešitev. Za primer C1000.9 porabi 32 blokov (enako za 16 blokov z 2 nitmi na blok) 150 sekund. Večje število blokov oz. niti ni možno, saj matrike zasedejo celoten pomnilnik GPE.

7.3 Sistem kolonije mravelj

GPE implementacija sistema kolonije mravelj je bila napisana specifično za računanje na GPE. Pri tem smo uporabili ugotovitve, ki smo jih dobili ob negativnih poskusih povzporejanja metod navzkrižne entropije in simuliranega ohlajanja. Implementacija tako ne

uporablja struktur in kazalcev ter uporablja čim manjše število pogojnih stavkov. Posamezne niti zasegajo čim manj pomnilnika, kar omogoča poganjanje veliko večjega števila niti in blokov, kot v implementacijah simuliranega ohlajanja in navzkrižne entropije.

Metoda povzporejanja je orisana že v prejšnjem poglavju in sicer pri obravnavanju možnih problemov, ki bi se pojavili ob povzporejanju sistema kolonije mravelj na GPE. Tukaj podamo konkretno vzporedno implementacijo za GPE in sicer za reševanje problema trgovskega potnika. Poleg implementacije na GPE, podamo še implementacijo optimizirano za izvajanje na CPE ter nato primerjamo izvajalni čas obeh implementacij za enako število mravelj. Poglejmo si najprej implementacijo na CPE, ki je podana v algoritmu 16.

Algoritem 16 Sistem mravelj na CPE.

```
iter_max, sosednostnaMatrika ← argumenti algoritma
N ← število mest
phMatrika ← inicializiraj na 1.0
hMatrika ← inverz razdalj sosednostne matrike
najboljšaKakovost ← inicializiraj
while iter < iter_max do
  začetnoMesto ← naključnoMesto()
  tvoriRešitev(začetnoMesto)
  kakovost ← oceni rešitev
  posodobiFeromone()
  if kakovost < najboljšaKakovost then
    najboljšaKakovost ← kakovost
  end if
  iter ← iter + 1
end while
```

Algoritem, ki se izvaja na GPE, je zelo podoben. Razlikuje se predvsem v tem, da izvaja na GPE več mravelj hkrati in, da izvaja posodobitev feromonske matrike v obliki ločenega programskega jedra na GPE. Za hrambo feromonske matrike in pa hevristične matrike uporabljamo strukturni pomnilnik. Za druge podatke uporabljamo globalni pomnilnik. Ko posamezna mravlja sestavi rešitev, jo oceni. Tukaj se pojavi problem posodobitve feromonske matrike. To bo sicer počelo drugo programsko jedro (zaradi potrebe po globalni uskladitvi), vendar je smiselno tu predstaviti podatke v obliki, ki se bo bolje podala povzporejanju.

Predstavljajmo si posodobitev feromonske matrike kot operacijo seštevanja matrik — to je namreč operacija, ki jo lahko GPE izvede zelo učinkovito. Ocene rešitev shranimo v matrično obliko. Na vsaki uporabljeni povezavi je vrednost, ki predstavlja oceno celotne rešitve, ki jo je sestavila mravlja. Te matrike lahko enostavno prištejemo feromonski matriki. Problem se pojavi, ker je mravelj lahko zelo veliko in sicer toliko, kot je niti. Tak način hrambe ocen rešitev lahko za malo večje probleme kaj hitro zapolni globalni pomnilnik, ki ga ima na voljo GPE. V kolikor hranimo samo eno matriko na blok, ki vsebuje feromonsko matriko z vsemi rešitvami v bloku, se prostorskemu problemu izognemo. Bločna uskladitev je zelo hiter postopek, ki ga lahko uporabimo za gradbo matrične reprezentacije ocen rešitev posameznega bloka.

Kadar imamo ocene rešitev v matričnih oblikah, lahko izvedemo vzporedno seštevanje teh matrik kot novo programsko jedro (glej glavno zanko algoritma 18). Vsaka nit v novem programskem jedru ima dodeljen en element teh matrik (vključno s feromonsko), na katerem izvede operacijo seštevanja. Vsaka nit sešteje N števil, kjer je N število rešitev v matričnih oblikah in jih shrani kot element feromonske matrike. Smiselno je, da vsa N števila teh matrik (razen feromonske), nato nastavi na vrednost 0 in s tem pripravi strukture za naslednjo iteracijo mravelj. Najprej si oglejmo programsko jedro, ki sešteje matrike vseh mravelj, kot je prikazano v algoritmu 17. Prikazan algoritem seštevanja določi eno nit (enolično identificirana s parametroma i in j) vsakemu elementu feromonske matrike.

Algoritem 17 Programsko jedro seštevanja feromonskih matrik.

funkcija $seštejMatrikeGPE(\rho)$
$mravlja \leftarrow 0$
$vsota \leftarrow 0$
while $mravlja < mravlje_{max}$ **do**
  $vsota \leftarrow vsota + rešitve_{mravlja,i,j}$
  $rešitve_{mravlja,i,j} \leftarrow 0$
  $mravlja \leftarrow mravlja + 1$
end while
$phMatrika_{i,j} \leftarrow (1 - \rho) \cdot phMatrika_{i,j} + \rho \cdot vsota$

Sedaj si oglejmo še del algoritma, ki teče na CPE in poganja programski jedri na GPE ob vsaki iteraciji. Prikazan je v algoritmu 18.

Algoritem 18 Del algoritma sistema mravelj, ki se izvaja na CPE.

$iter_{max}, \rho, hMatrika \leftarrow$ inicializiraj
$phMatrika \leftarrow$ inicializiraj na 1
$GPE \leftarrow phMatrika$
$GPE \leftarrow hMatrika$
$iter \leftarrow 0$
while $iter < iter_{max}$ **do**
  $poženiMravljeGPE(\rho)$
  $seštejMatrikeGPE(\rho)$
  $iter \leftarrow iter + 1$
end while

Ker hranimo vse podatke na GPE, ne potrebujemo nobenega nalaganja podatkov iz glavnega pomnilnika na GPE oz. obratno med iteracijami. Nato si poglejmo še algoritem posameznih mravelj, ki gradijo rešitve. Prikazan je v algoritmu 19.

Algoritem 19 Del algoritma sistema mravelj, ki se izvaja na GPE.

funkcija $poženiMravljeGPE(\rho)$
$začetnoMesto \leftarrow naključnoMesto()$
$tvoriRešitev(začetnoMesto)$
$ocena \leftarrow oceniRešitev(rešitev)$
$shraniVMatričnoObliko(rešitev, ocena)$

Algoritem smo testirali na primerih TSPLIB simetričnega trgovskega potnika. Pognali smo toliko mravelj, kot je mest na posameznem primeru in vsaka mravlja je začela graditi rešitev na različnem mestu (v nasprotju z naključnim pristopom podanim v zgornjem algoritmu). V vseh primerih smo izvedli smo 10 iteracij algoritma. Rezultati so prikazani v tabeli 7.5. Prvi stolpec prikazuje uporabljen primerek. Drugi stolpec prikazuje število mest (vozlišč grafa) v uporabljenem primerku. Tretji stolpec prikazuje izvajalni čas algoritma na CPE v sekundah. Četrti stolpec prikazuje izvajalni čas algoritma na GPE v sekundah, peti stolpec pa prikazuje faktor pospešitve na GPE v primerjavi s CPE. Opis CPE in GPE je podan v začetku tega poglavja. Omeniti je potrebno, da se implementacija ne približa faktorju pospešitve 64, ki je število procesorjev uporabljenega GPE. Vendar 64-kratni pospešek verjetno ni možen sam po sebi, saj so procesorske enote GPE različne od CPE. Kot je razvidno iz tabele pospešitev najprej raste in se nato ustavi, kar lahko kaže na to, da so procesorji GPE že prezaposleni.

Primerek	**Št. mest**	**CPE**	**GPE**	**Pospešitev**
ts225	225	1,5	0,5	3
pr439	439	11	2	5,5
rat575	575	26	4	6,5
pr1002	1002	134	14	9,5
nrw1379	1379	366	48	7,6

Tabela 7.5: Izvajalni čas izbranih TSPLIB primerkov.

Poglavje 8

Ugotovitve in nadaljne delo

V nalogi smo si ogledali klasične koncepte povzporejanja ter omenili okolja za porazdeljeno računanje MPI in PVM. Nadaljevali smo s predstavitvijo Nvidinega okolja CUDA in podali nekaj splošnih informacij glede arhitekture Nvidinih kartic. Na kratko smo še predstavili okolje OpenCL. Nato smo si pogledali nekaj problemov iz kombinatorične optimizacije: problem trgovskega potnika, največjega polnega podgrafa, najmanjše neodvisne množice in najmanjšega pokritja grafa.

Omenjene probleme smo nato uporabili za obrazložitev tehnike metahevristične optimizacije in algoritmov iz tega področja: simulirano ohlajanje, navzkrižna entropija, evolucijski algoritmi, optimizacijo s kolonijo mravelj in iskanje s prepovedmi. Ogledali smo si še vzporedne implementacije teh algoritmov in jih analizirali iz vidika povzporejanja na GPE.

Pri iskanju s prepovedmi smo videli, da že obstaja implementacija na GPE, vendar je to implementacija preko standardnega vmesnika za grafiko in ne za splošno računanje, kot je CUDA. Ker tak pristop ne omogoča splošnega programiranja GPE, je povzporejanje omejeno na izračun dolžine poti, ki jih naloži CPE. Tak pristop prinaša minimalno pospešitev. Predstavili smo idejo, kako bi se lahko povzporejanja na GPE lotili z uporabo bločne uskladitve in delitve dela, pri čem smo minimizirali količino komunikacije s CPE.

Sledil je ogled evolucijskega algoritma, za katerega tudi obstaja vzporedna implementacija za GPE in sicer z uporabo okolja CUDA. Implementacija temelji na delitvi populacije osebkov na otoke, med katerimi se manj pogosto izvaja migracija, t.j. mešanje rešitev, za potrebe izogibanja lokalnim optimumom.

Pri simuliranem ohlajanju smo si ogledali povzporejanje, ki temelji na delitvi vhodnih podatkov med procese. Problem, ki nastopi pri povzporejanju simuliranega ohlajanja na GPE, je metoda kopiranja rešitev ob vsaki iteraciji, saj je število iteracij tega algoritma pogosto zelo veliko. V kolikor bi želeli algoritem smiselno povzporejati na GPE, bi bilo potrebno kopiranje minimizirati.

Ogledali smo si vzporedno implementacijo metode navzkrižne entropije z uporabo okolja MPI. Implementacija ne izvaja vzporedne posodobitve verjetnostne matrike. Izkaže se, da razširitev te matrike na ostale procese predstavlja ozko grlo sistema. Podoben način povzporejanja opišemo tudi za GPE.

Sledila je še optimizacija s kolonijo mravelj. Omenili smo vzporedno implementacijo za okolje MPI, kjer je bilo povzporejanje izvedeno za kreacijo rešitev in ocenjevanje le-teh. Posodobitev feromonske matrike je izvajal en proces. Nato smo podali način povzporejanja

na GPE, ki izvaja vse korake algoritma vzporedno na grafični kartici in se tako izogne kopiranju podatkov na glavni pomnilnik. Snovali smo tak vzporedni algoritem, da smo izvedeli usklajevanje na osnovi porazdelitve podatkov. Celoten algoritem smo razdelili na dva disjunktna dela, pri čem je bila edina zahteva, da se drugi del začne izvajati šele potem, ko se je prvi del že zaključil. Prvi del algoritma je bila glavna iteracija sistema mravelj, drugi del pa seštevanje feromonskih matrik.

Na koncu smo podali empirične rezultate in psevdokodo za vzporedne implementacije algoritmov. Pokazali smo, da sta implementaciji simuliranega ohlajanja in navzkrižne entropije, ki sta bili povzeti po obstoječih implementacijah za CPE in okolji za porazdeljeno računanje MPI in PVM, počasnejši na GPE. Z uporabo ugotovitev, ki smo jih dobili z negativnim poskusom povzporejanja prejšnjih dveh algoritmov, smo implementirali vzporedni sistem mravelj, ki pa je bil napisan za potrebe GPE in je veliko hitrejši od implementacije na CPE.

Za v bodoče se odpira kar nekaj zanimivih odprtih vprašanj. Na primer povzporejanje iskanja s prepovedmi po ideji, ki smo jo podali v nalogi. Vzporedni algoritem sistema mravelj bi lahko tudi testirali na novejših karticah, ki vsebujejo dosti večje število multiprocesorjev. Dobljene rezultate bi lahko primerjali še z implementacijo v okolju OpenCL, ki omogoča uporabo AMD-jevih GPE in nenazadnje tudi mešanih sistemov. Poleg tega bi bilo zanimivo videti izboljšave, ki jih prinaša Fermi arhitektura, saj vključuje veliko izboljšav pri dostopu do pomnilnika ter vsebuje globalni predpomnilnik.

Literatura

[1] (2009, avg.) NVIDIA CUDA Programming Guide 2.3.1. Dostopno na: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[2] (2009, jul.) NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3. Dostopno na: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[3] (2008, apr.) Dr. Dobb's - CUDA, Supercomputing for the Masses. Dostopno na: http://www.drdobbs.com/high-performance-computing/207200659

[4] (2008, nov.) The NVIDIA CUDA Debugger, User Manual, Version 2.1 Beta. Dostopno na: http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf

[5] (2010, jun.) The OpenCL Specification, Version: 1.1, Document Revision: 33. Dostopno na: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[6] C. Blum in A. Roli, Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, *ACM Computing Surveys* **35** (2003), 268–308.

[7] A. Kirkpatrick, C. D. Gelatt, Jr in M. P. Vecchi, Optimization by simulated annealing, *Science* **220** (1983), 671–680.

[8] M. R. Garey in D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, New York: W. H. Freeman (1983).

[9] S. Skiena, Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley (1990), 218–219.

[10] S. S. Skiena, The Algorithm Design Manual, New York: Springer-Verlag (1997), 312–314.

[11] R. Diekmann, R. Lüling in J. Simon, Problem Independent Distributed Simulated Annealing and its Applications, R. V. Vidal (ed.): *Applied Simulated Annealing*, Lecture Notes in Economics and Mathematical Systems, Springer LNEMS 396 (1993), 17–44.

[12] M. Dorigo, Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italy (1992).

[13] M. Dorigo in L.M. Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, *IEEE Transactions on Evolutionary Computation*, volume 1, (1997) 53–66.

[14] T. Stützle in H.H. Hoos, MAX MIN Ant System, *Future Generation Computer Systems*, volume 16, (2000) 889–914.

[15] (1993, sept.) DIMACS Instances. Dostopno na: `ftp://dimacs.rutgers.edu/pub/challenge`

[16] M. Dorigo, Ant colony optimization, *Scholarpedia* **3** volume 2 (2007), 1461.

[17] E. Onbaşoğlu in L. Özdamar, Parallel simulated annealing algorithms in global optimization, *Journal of Global Optimization* **19** (2001), 27–50.

[18] M. Dorigo, V. Maniezzo, in A. Colorni, Ant System: Optimization by a colony of cooperating agents, *IEEE Transactions on Systems, Man, and Cybernetics — Part B* **26** (1996), 29–41.

[19] F. Glover in M. Laguna, Tabu Search, *Kluwer Academic Publishers* (1997).

[20] E. Taillard, Robust Taboo Search for the Quadratic Assignment Problem, *Parallel Computing* **17** (1991), 443455.

[21] F. Glover, Tabu search Part II, *ORSA J. Comput.* **2** (1990), 4–32.

[22] R. Battiti in M. Protasi, A history-base heuristic for MAX-SAT, *ACM J. Exper. Algor.* **2** (1997).

[23] R. Battiti in G. Tecchiolli, The reactive tabu search, *ORSA J. Comput.* **6** (1994), 126–140.

[24] A. Janiak, W. Janiak in M. Lichtenstein, Tabu Search on GPU, *Journal of Universal Computer Science* **14** (2008), 2416–2427.

[25] P. Delisle, M. Krakecki, M. Gravel in C. Gagné, Parallel implementation of an ant colony optimization metaheuristic with OpenMP, *International conference of parallel architectures and complication techniques (PACT), Proceedings of the third European workshop on OpenMP* (1997), 8–12.

[26] Q. Lü, Z-H. Bai, X-Y. Xia Leader-Based Parallel Cross Entropy Algorithm for Maximum Clique Problem, *Journal of Software* **19** (2008), 2899–2907.

[27] G. Alexe, G. Bhanot in A. Climescu-Haulica, A Cross Entropy Algorithm for Classification with δ–Patterns, *DMTCS proceedings* **AG** (2006), 399–402.

[28] G. E. Evans, J. M. Keith in D. P. Kroese, Parallel Cross-Entropy Optimization, *Proceedings of the 2007 Winter Simulation Conference* (2007), 2196–2202.

[29] A. Hertz in D. Kobler, A framework for the description of evolutionary algorithms, *European Journal of Operational Research* **126** (2000), 1–12.

[30] P. Pospichal in J. Jaros, GPU-based Acceleration of the Genetic Algorithm, *GECCO'09 Competition* (2009).

[31] W. N. Martin, J. Lienig in J. P. Cohoon, Handbook of Evolutionary Computation, *IOP Publishing Ltd and Oxford University Press* (1997).

[32] R. Cvahte, Povzporejanje metahevristik za NP-polne probleme, *Dela FRI* (2010).

[33] J. R. A. Allwright in D. B. Carpenter, A distributed implementation of simulated annealing for the travelling salesman problem, *Parallel Computing* **10** (1989), 335–338.

[34] M. Snir, S. Otto, S. Huss-Lederman, D. Walker in J. Dongarra, MPI: The Complete Reference, The MIT Press, Cambridge, Massachusetts, (1998).

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek in V. Sunderam, PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing, MIT Press, Cambridge, MA, (1995).

[36] W. Feng in P. Balaji, Tools and environments for multicore and many-core architectures, *Computer* **12** (2009), 26–27.

[37] (2008, avg.) TSPLIB. Dostopno na: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

[38] (2010, avg.) Open64. Dostopno na: `http://www.open64.net/`

[39] (2010, avg.) Next Generation CUDA Architecture, Code Named Fermi – NVIDIA. Dostopno na: `http://www.nvidia.com/object/fermi_architecture.html`

[40] Arvind, D. August, K. Pingali, D. Chiou, R. Sendag, J. J. Yi, Programming multicores: Do applications programmers need to write explicitly parallel programs? *IEEE Micro* **30** (2010), 19–33.

[41] (2010, avg.) Islovar. Dostopno na: `http://www.islovar.org/`

Priloge

- Zgoščenka s celotno uporabljeno kodo