Kauppamatkustajaongelman ratkaiseminen geneettisellä algoritmilla

Markus Korpinen

Suurteholaskennan työkalut
Fysiikan laitos, Helsingin yliopisto

13. tammikuuta 2012

1 Johdanto

Kauppamatkustajaongelma on yksi esimerkki laskennallisesti huonosti skaalautuvasta ongelmasta. Kauppamatkustajaongelmassa pyritään löytämään usean eri kaupungin välille reitti, joka on lyhyempi kuin mikään muu reitti niin että palataan takaisin alkupisteeseen. Tehokkaita ratkaisuja lyhyimmän reitin löytämiseksi ei ole, mutta vähemmän tarkan ratkaisun kelvatessa on useitakin nopeita algoritmejä, kuten geneettiset algoritmit, jotka oikein käytettäessä ovat suhteellisen nopeita ja antavat tyydyttäviä vastauksia.

Tässä työssä kauppamatkustajaongelmaa pyritään ratkaisemaan geneettisten algoritmien avulla käyttäen apuna c-ohjelmointikieltä.

2 Algoritmi

Geneettiset algoritmit pyrkivät ratkaisemaan ongelman niin että satunnaisesti valittujen ratkaisujen joukosta muodestetaan populaatio, jonka parhaat yksilöt lisääntyvät keskenään muodostaen aina vain parempia yksilöitä. Tämän lisäksi yksilöt mutatoituvat satunnaisesti, jotta pystyttäisiin lieventämään sitä tosiasiaa, että satunnaisesti valittu populaatio on usein vain hyvin pieni murto-osa kaikkien eri vaihtoehtojen joukosta. Monen generaation jälkeen voidaan poimia tulos, jonka pitäisi olla lähellä oikeaa tulosta, mutta ei kuitenkaan voida olla varmoja onko tulos tarkin mahdollinen.

Kauppamatkustajaongelman ratkaisemiseksi geneettisillä algoritmeillä muodostetaan aluksi satunnaisesti valittujen reittien joukosta populaatio. Tämän reittijoukon lyhyimpiä reittejä yhdistelemällä muodestetaan reittejä , jotka ovat myöskin lyhyimpiä tai parhaimmassa tapauksessa lyhyimpiä reittejä. Reitit lajitellaan aina lyhyimmyys järjestykseen ja uudet reitit syrjäyttävät joukon huonoimmat reitit. Tämän lisäksi mutaatio tapahtuu niin, että tietyllä todennäköisyydellä reittien jotkin kaupungit vaihtavat paikkoja keskenään.

3 Rinnakkaistaminen

Ohjelma pyrittiin ohjelmoimaan niin, että root-prosessori huolehtii tiedoston lukemisesta, kaupunkien välisten etäisyyksien laskemisesta, ja geneettisen algoritmien asetuksien alustamisesta. Tämän jälkeen algoritmin rinnakkaistamista varten tarvittavat tiedot lähetetään *MPI_Broadcast*-funktiolla muille prosessoreille. Jokainen prosessori muodostaa oman populaationsa ja lähettää aina kymmenen generaation välein kahden parhaan yksilönsä lapsen seuraavalle prosessorille ja myöskin vastaanottaa edelliseltä prosessorilta *MPI_Sendrecv*-funktiolla. Näin populaatiot saadaan sekoittumaan. Riittävän monen generaation jälkeen prosessorit lähettävät parhaan yksilönsä root-prosessorille, joka valitsee näistä parhaimman ja tulostaa sen näkyviin.

4 Ohjelman rakenne

Ohjelma koostuu kolmesta osasta, jotka ovat pääohjelman **tsp_mpi** lisäksi moduulit **genetic** ja **cities**. Ohjelma avaa tiedoston, lukee kaupungit, laskee kaupunkien väliset etäisyydet, luo satunnaisen populaation reiteistä sekä lajittelee, pariuttaa, ja mutatoi

usean generaation ajan populaatiota. Lopuksi tulostetaan populaation vahvimmat yksilöt, eli lyhyimmät reitit.

4.1 Tsp_mpi.c

Pääohjelma kutsuu funktioita moduuleista **genetic** ja **cities**. Rinnakkaistamattoman pääohjelman nimi on **tsp_genetic.c**.

4.2 Cities

Moduuli kostuu tiedostosta cities.c, sekä sen header-tiedostosta cities.h. Cities.c sisältää seuraavat funktiot:

print_city tulostaa halutun kaupungin nimen ja koordinaatit komentoriville.

count_lines lukee tiedostosta kaupunkien lukumäärän. Funktio jättää huomiotta rivit jotka on merkattu #-merkillä, sekä tyhjät rivit.

read_cities lukee tiedostosta kaupungit koordinaatteineen ja tallentaa ne muistiin. Kaupunkien on oltava muodossa *nimi leveyspiiri pituuspiiri*. Leveyspiiri ja pituuspiiri luetaan asteissa.

to_radians muuttaa asteet radiaaneiksi.

distance ottaa parametreina lähtö– ja maalikaupungin ja palauttaa niiden välisen etäisyyden.

calculate_distances ottaa sisääntuloparametreina kaksiulotteisen taulukon, listan kaupungeista sekä kaupunkien lukumäärän. Funktio alustaa kaksiulotteisen taulukon distances ja laskee siihen kaikkien kaupunkin väliset etäisyydet sekä palauttaa paluuarvona kaksiulotteisen taulukon.

4.3 Genetic

Moduuli kostuu tiedostosta genetic.c, sekä sen header-tiedostosta genetic.h. Genetic.c sisältää seuraavat funktiot:

generate_random_population alustaa popupulaation ja reitit omiin tietotyyppeihinsä *Population* ja *Path*. Populaatio sisältää tiedon populaation lukumäärästä sekä osoitinmuuttujan taulukkoon reiteistä. Satunnaisesti valitut reitit tallennetaan tietotyypeiksi *Path* ja niiden mukana tallennetaan tieto reitin pituudesta.

calculate_fitness laskee yksittäisen reitin "kunnon"eli sen kuinka lyhyt reitti on.

generate_random_combination ottaa parametreina reitin ja asetukset, varaa muistista riittävästi tilaa reitille ja luo satunnaisen reitin kaupungeista.

compare_population on pikalajittelua (**qsort**) varten muodostettu vertailufunktio.

print_population tulostaa populaation niin että näkyviin tylee jokaisen reitin järjestysnumero ja pituus.

simple_breed_population on yksinkertaisempi versio suunnitellusta risteyttämisfunktiosta. Funktio risteyttää kymmenen populaation parasta niin että ensimmäinen ja toinen risteytyvät, kolmas ja neljäs risteytyvät jne. Risteyttämiseen käytetään funktiokutsua *mate*.

mate ottaa parametreina kaksi reittiä ja palauttaa niistä tehtävänannossa esitellyn tavan mukaisesti uuden reitin.

mutate_population aiheuttaa syötettyyn populaatioon mutaatioita vakion *mutationRate* mukaisesti. Mutaatio tapahtuu vaihtamalla satunnaisesti valitun kahden kaupungin paikkaa. Tämän lisäksi mutatoiduille reitelle lasketaan uudet pituudet funktiolla *path_distance*

path_distance laskee reitin pituuden käyttäen apunaan kaikkien kaupunkien väliset reitit sisältävää taulukkoa *distances*.

5 Käyttödokumentti

5.1 Rinnakkaistettu ajo

Ohjelman lähdekoodi käännetään kirjoittamalla komentoriville

```
$ make -f Makefile_mpi
```

ja tämän jälkeen ohjelman voi suorittaa käyttäen esimerkiksi neljää laskentaydintä komennolla

```
$ mpirun -np 4 ‘pwd‘/TSP
```

Koska en onnistunut saamaan rinnakkaistettua versiota ohjelmasta toimimaan kunnolla oli ohjelman ulosanti seuraava:

```
$ mpirun -np 4 ‘pwd‘/TSP
WARNING: Unable to read mpd.hosts or list of hosts isn’t
provided. MPI job will be run on the current machine only.
TaskID 1: Init random seed succesful...
TaskID 1: Init population succesful...
TaskID 2: Init random seed succesful...
TaskID 2: Init population succesful...
rank 3 in job 1 korundi.grid.helsinki.fi_57031 caused
collective abort of all ranks exit status of rank 3:
killed by signal 11
rank 2 in job 1 korundi.grid.helsinki.fi_57031 caused
collective abort of all ranks exit status of rank 2:
killed by signal 11
rank 1 in job 1 korundi.grid.helsinki.fi_57031 caused
collective abort of all ranks exit status of rank 1:
killed by signal 11
TaskID 3: Init random seed succesful...
TaskID 3: Init population succesful...
```

Ongelma todennäköisesti johtuu väärästä tavasta lähettää itsemääriteltyjä tietotyyppejä, mutta aika ei riittänyt kaikkien ongelmien selvittämiseen.

5.2 Rinnakkaistamaton ajo

Ohjelman lähdekoodi käännetään kirjoittamalla komentoriville

```
$ make
```

ja tämän jälkeen ohjelman voi suorittaa komennolla

```
$ ./TSP
```

Yhdessä ajossa ohjelma luki 22 kaupunkia ja tulosti

```
$ ./TSP

Number of cities: 22
Init cities succesful...
Init distances succesful...
Init GA config succesful...
Init random seed succesful...
Init population succesful...
Init mostFit succesful...
Shortest path of 1000th generation: 45994
Shortest path of 2000th generation: 40775
Shortest path of 3000th generation: 40775
Shortest path of 4000th generation: 40568
Shortest path of 5000th generation: 40568
Shortest path of 6000th generation: 40568
Shortest path of 7000th generation: 40568
Shortest path of 8000th generation: 40568
Shortest path of 9000th generation: 40568
Shortest path of 10000th generation: 40568
Shortest path is 40568

PARIISI (48.850000 2.350000)
MOSKOVA (55.750000 37.610000)
ESPOO (60.200000 24.650000)
...
```

6 Vertailuanalyysi

Rinnakkaistetun koodin toimimattomuuden vuoksi haluttua vertailuanalyysiä ei pystytty suorittamaan.

7 Johtopäätökset

Tehtävässä käytetyn 22 kaupungin muodostamia mahdollisia reittejä on 21! kappaletta eli $5,1 * 10^{19}$ analysoitavaa reittiä. Pienimmän mahdollisen reitin löytäminen käyttäen esimerkiksi pelkkää raakaa laskentatehoa kestää jo aika kauan ja sitä varten esimerkiksi nyt käytetty tuhannen reitin populaatio kymmenellä tuhannella generaatiolla on hyvin paljon nopeampi. Sen todennäköisyys että reitti olisi pienin mahdollinen ei tässä tapauksessa ole kovinkaan suuri ja se ilmenee sillä että tarkasteltaessa usean ajon tuloksia aina välillä löytyy taas kerran lyhyempi reitti kuin aikaisemmin. Jotta algoritmi toimisi paremmin olisi risteytymistä ja mutatoimista kehitettävä hieman niin että voisi olla mahdollista saada vahvempiakin mutaatioita ja että pariutumiseen myöskin tulisi hieman satunnuisuutta mukaan eikä niin että vain parhaat pääsevät pariutumaan. Mikäli vain samaan lokaaliin

minimiin joutuneet yksilöt pariutuvat keskenään niin silloin kuopasta pois pääseminen on hieman vaikeampaa. Geneettisten algoritmien käyttäminen mahdollistaa monien ongelmien tutkimisen kunhan vain luovutaan esimerkiksi siitä toivosta että pystyttäisiin aina löytämään pienin minimi.