Game of Life -projektin raportti

Juho Kallio
Kai Kontio
Ville Niemi

Johdanto

Ongelman kuvaus

Conwayn Game of Life muodostuu äärettömästä kaksiulotteisesta soluautomaatista, jossa solun tila voi olla elävä tai kuollut. Solujen tilat muuttuvat sukupolvesta toiseen yksinkertaisten sääntöjen perusteella, vierussolujen tilasta riippuen.

Tässä kurssiprojektissa toteutetussa Game of Lifessa ei käytetä ääretöntä soluruudukkoa, vaan soluruudukon koko ja alkutila annetaan ohjelmalle syötteenä. Joissain toteutuksissa nähtyä äärellisen soluruudukon reunojen ympäri “käärimistä” ei käytetä, vaan soluruudukon ulkopuolella laskennallisesti olevat naapurit katsotaan aina kuolleiksi.

Game of Lifen säännöt seuraavan solusukupolven laskemiselle ovat seuraavat:

1. Elävä solu, jolla on vähemmän kuin kaksi elävää naapuria, kuolee
2. Elävä solu, jolla on kaksi tai kolme elävää naapuria, säilyy elävänä
3. Elävä solu, jolla on enemmän kuin kolme elävää naapuria, kuolee
4. Kuollut solu, jolla on tasan kolme elävää naapuria, herää henkiin

Rinnakkaisuusongelman ratkaisu

Ratkaisussa käytetään kahta soluruudukkoa, joista toinen on luku- ja toinen kirjoituspuskuri. Lukupuskurissa on nykyinen solusukupolvi ja kirjoituspuskuriin muodostetaan tuleva solusukupolvi. Kun uusi sukupolvi on kokonaisuudessaan laskettu, tulee kirjoituspuskurista uusi lukupuskuri ja päinvastoin.

Puskureiden käyttö helpottaa säikeiden lukujen ja kirjoitusten synkronointia. Koska lukupuskuria ei muuteta uutta sukupolvea laskettaessa, ei siihen tehtyjä lukuoperaatioita tarvitse synkronoida. Kirjoituspuskurista jaetaan säikeille palasia synkronoidusti rivi kerrallaan ja säikeet kirjoittavat lukupuskuriin ainoastaan omille riveilleen.

Soluruudukoiden toteutus

Soluruudukot on käytännössä toteutettu yksiuloitteisina short -tyyppisinä taulukoina, joissa ykkönen merkitsee elävää solua ja nolla kuollutta. Periaatteessa byte -tyyppinen taulukko riittäisi, mutta käytännössä Javassa byte -tyypeillä aritmeettinen operointi on tehty suhteettoman

vaikeaksi. Yksiulotteisen taulukon käyttö kaksiulotteisen sijaan nosti ohjelman suorituskykyä noin 25%.

Jotta vältytään reunatapausten tunnistamiselta ja käsittelyltä laskentaa suorittavissa säikeissä, käytetään soluruudukot toteuttavissa taulukoissa puskurointia taulukon alussa, lopussa ja jokaisen rivin lopussa. Puskurisolut ovat aina kuolleita ja niiden tarkoitus on kuvata varsinaisen soluruudukon ulkopuolella olevia soluja, joita kuitenkin tarvitaan, kun soluruudukon reunasolujen vierussolujen tilaa halutaan tutkia.

Taulukoiden rakenne N*N kokoiselle soluruudukolle:

```
Taulukon ensimmäinen indeksi
|
<- puskuri N+2 solua ->
<- soluruudukon rivi 1, N solua  + 1 solu puskuria ->
<- soluruudukon rivi 2, N solua  + 1 solu puskuria ->
<- soluruudukon rivi 3, N solua  + 1 solu puskuria ->
...
<- soluruudukon rivi N, N solua  + 1 solu puskuria ->
<- puskuri N+2 solua ->
                      |
Taulukon viimeinen indeksi
```

Laskentasäikeet ja synkronointi

Ohjelman käyttämien laskentasäikeiden määrä voidaan antaa parametrina käynnistysskriptille tai vaihtoehtoisesti, jos parametria ei anneta, ohjelma voi dynaamisesti päätellä JVM:lle käytettävissä olevien prosessoriydinten määrän ja käyttää niitä kaikkia.

Ohjelma käyttää säikeiden synkronointiin kahta eri ratkaisua. Soluruudukosta jaetaan laskentasäikeille paloja rivi kerrallaan Javan AtomicIntegerin avulla. Uuden sukupolven laskennan alkaessa AtomicInteger osoittaa ensimmäiseen riviin. Laskentasäikeet suorittavat looppia, jossa AtomicIntegerin getAndIncrementillä pyydetään laskettavan rivin numero ja prosessoidaan rivin seuraavan sukupolven tila. Soluruudukon prosessointi riveittäin oli noin 10% nopeampaa kuin soluruudukon jako säikeille samansuuruisiin alueisiin ja näiden prosessointi.

Solusukupolvien välillä tapahtuva soluruudukkopuskureiden vaihto edellyttää, että kaikki laskentasäikeet ovat lopettaneet seuraavan sukupolven laskennan. Tähän käytetään Javan CyclicBarrieria. Laskentasäie menee CyclicBarrierille odottelemaan, kun säie saa getAndIncrementillä rivin, joka on suurempi kuin soluruudukon rivien määrä. Kun kaikki laskentasäikeet ovat CyclicBarrierilla odottamassa, kutsuu CyclicBarrier metodia, joka vaihtaa luku- ja kirjoituspuskurit ja nollaa seuraavaa prosessoitavaa riviä osoittavan AtomicIntegerin. Tämän jälkeen, mikäli laskettu sukupolvi ei ollut viimeinen, CyclicBarrier vapauttaa säikeet

laskemaan seuraavaa sukupolvea.

On huomioitava, että Game of Lifen ehtojen tarkistusjärjestyksellä on merkitystä ohjelmaa suoritettaessa. Selvästi yleisin tapaus (n. 94%), eli alle kaksi elävää naapuria, kannattaa tarkastaa ensin. Sen jälkeen saman periaatteen mukaisesti loput ehdot niiden toteutumistodennäköisyyden mukaisessa järjestyksessä.

Toinen merkityksellinen asia ehtoja tarkistettaessa on, että prosessoitavan solun tila edellisessä sukupolvessa on merkityksellinen vain, jos sillä on tasan kaksi elävää naapuria. Eli kolmessa tapauksessa neljästä voimme päätellä solun tilan seuraavassa sukupolvessa tekemättä muistinoutoa kyseiselle solulle.

Sarjallinen ratkaisu

Ohjelma voidaan suorittaa vain yhdellä laskentasäikeellä, jolloin ratkaisu on sarjallinen.

Oikeellisuus

Poissulkevuusehto

Sukupolvien välissä tapahtuva luku- ja kirjoituspuskurien vaihto on kriittinen vaihe. Tällöin laskentasäikeet eivät saa olla kirjoittamassa tai lukemassa. CyclicBarrier pitää huolta, että laskentasäikeet ovat odottamassa, kun vaihto tehdään ja vapauttaa ne seuraavan sukupolven laskentaan vaihdon jälkeen.

Toinen kriittinen vaihe on prosessoitavien rivien jakaminen laskentasäikeille. Javan AtomicInteger pitää huolta, että kaksi säiettä ei prosessoi samaa riviä yhtä aikaa.

Saman rivin samanaikainen prosessointi ei luku- ja kirjoituspuskurien ansiosta johtaisi väärään lopputulokseen, vaan ongelma on suorituskyvyn heikkeneminen ja epätodennäköisessä skenaariossa tapahtuva elolukko.

Nälkiintyminen ja lukkiutuminen

Laskentasäikeiden kriittinen vaihe on uuden rivinumeron saaminen AtomicIntegeriltä. AtomicInteger käyttää prosessorin tarjoamia atomisia operaatoita, jos mahdollista, eikä siis blokkaa säikeitä. Mikäli rautatason tukea ei ole saatavilla, saattavat säikeet olla ohimenevästi blokattuna getAndIncrementin kohdalla mutta pääsevät lopulta jatkamaan, viimeistään kun getAndIncrementin lukkoa pitävä säie lopettaa.

Käyttöohjeet

Ohjelma on pakattu zip -pakettiin, jonka purkautuu riorage12 -kansioon. Kansiosta löytyy “compile.sh” ja “start.sh” skriptit, joilla ohjelman voi kääntää ja ajaa laitoksen Ukko -klusterilla.

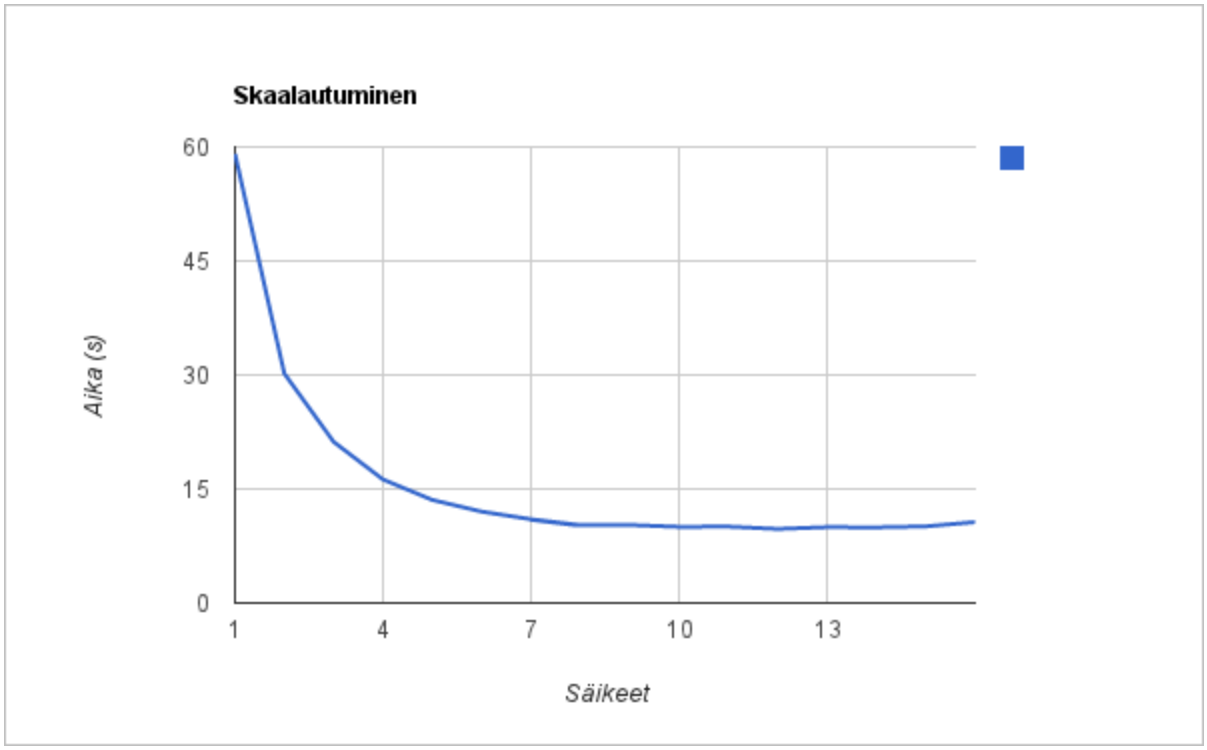
Alihakemistossa src on ohjelman lähdekoodi ja alihakemistossa datasets on

suorituskykykilpailun input tiedostonimellä “life_800_10000.txt” ja ohjelman tuottama output tiedostossa “life_results.txt”.

Käynnistysskriptille voi halutessaan antaa parametreina input tiedoston (-i), output tiedoston (-o) ja säikeiden määrän (-t).

Mikäli parametreja ei anneta käytetään skriptin oletusparametreja, joissa input luetaan tiedostosta "/home/fs/kerola/rio_testdata/life_800_10000.txt", output kirjoitetaan tiedostoon "/tmp/life_results.txt" ja ohjelma käyttää kaikkia JVM:lle näkyviä ytimiä.

Suorituskyky ja skaalautuminen



Säikeet	1	2	3	4	5	6	7	8
Aika (ms)	59114	30195	21203	16252	13559	12022	11017	10238
Säikeet	9	10	11	12	13	14	15	16
Aika (ms)	10267	10019	10048	9742	9979	9950	10079	10642

Laitoksen laskentaklusterilla (Ukko node 018) ajettujen suorituskykytestien perusteella, ohjelma skaalautuu odotusten mukaisesti. Aina kahdeksaan säikeeseen asti suorituskyky paranee merkittävästi mutta tämän jälkeen tasaantuu. Voimme tehdä johtopäätöksen, että Intelin HyperThreadingista ei ole juuri etua tätä ohjelmaa ajettaessa.

Ohjelma skaalautuu tätäkin suuremmalle prosessorimäärälle kivuttomasti yksittäisen

laskentaosuuden ollessa yksi soluruudukon rivi. Tosin prosessorimäärän ylittäessä soluruudukon rivimäärän ei säikeiden lisääminen rivimäärää suuremmaksi tuo kuin säikeiden aiheuttamaa yleisrasitetta.

Rinnakkaistuksella saavutetuksi nopeuseduksi saatiin kaavalla “paras yhden säikeen suoritusaika / paras rinnakkainen aika”, 59114 ms / 9742 ms = 6,07.

Nopeimmaksi ohjelman suoritusajaksi saatiin suorituskykytesteissä 9742 ms.

```
@ukko018:~/riorage12$ ./start.sh
Using data set at: /home/fs/kerola/rio_testdata/life_800_10000.txt
Initializing game state... done.
Grid size: 800x800 Game steps: 10000
Starting game with 12 worker threads...
Used 9742ms on 10000 game steps.
Saving the results to: /tmp/life_results.txt ...done.
Bye!
Thu Apr 26 14:16:52 EEST 2012
```

Yhteenveto

Ohjelman toteutus muuttui projektin aikana tasasuuruisiin laskentaosiin jaetuista kaksiulotteisista taulukoista nykyiseen yksiulotteisiin taulukoihin ja riviperustaiseen laskentajakoon.

Soluruudukon jakaminen laskentasäikeille riveinä toi samalla koodiin yksinkertaisuutta ja suoritukseen tehokkuutta. Toisaalta kaksiulotteisesta taulukosta luopuminen toi ohjelmakoodiin monimutkaisuutta mutta huomattavasti suorituskykyä.

Suurin yllätys projektissa oli yksiulotteisen taulukon suuri suorituskykyero kaksiulotteiseen verrattuna. Luulisi kääntäjän osaavan optimoida tällaisen perustapauksen aika hyvin, mutta toisin kävi. Toisaalta vähemmän suorituskykyintensiivisessä ohjelmassa kaksiulotteinen taulukko on varmasti oikea valinta jo ohjelmakoodin luettavuuden ja ylläpidettävyyden takia.

Skaalautumisessa ei tullut suuria yllätyksiä, vaan ukolla ajetut testit olivat linjassa odotusten ja projektin jäsenten omilla koneillaan ajamien testien kanssa. HyperThreadingista oli odotettua vähemmän hyötyä.