Rinnakkaisohjelmoinnin projekti

Kokonaislukujen järjestäminen

Kasper Hirvikoski 04.02.90
Santeri Hiltunen 01.08.89
Emmi Otava 24.01.88

Lisäpisteet: Kasper (2p)

Esittely

Tehtävänä oli toteuttaa kokonaislukujen järjestämisalgoritmi rinnakkaisesti. Algoritmin tulisi loppupelissä järjestää 58 miljoonaa 64-bittistä kokonaislukua nousevaan järjestykseen mahdollisimman tehokkaasti. Luvut tulisi lukea annetusta testidatasta, joka koostuu Little-Endian tavujärjestyksessä esitetyistä 64-bittisistä kokonaisluvuista. Luvut ovat etumerkitsemättömässä muodossa, mutta koska Java ei tulkitse etumerkitsemättömiä kokonaislukuja hyvin, toteutimme lukujen luvun ohjeiden mukaan etumerkillisessä muodossa. Kukaan ryhmässämme ei ollut aikaisemmin toteuttanut rinnakkaisia algoritmeja, joten lähdimme liikkeelle puhtaalta pöydältä.
Varsinkin Javan rinnakkaisuutta käsittelevä dokumentti osoittautui erittäin hyödylliseksi projektin toteuttamisen tueksi.

Sarjallisen algoritmin suunnittelu

Ryhdyimme suunnittelemaan ensin sarjallista toteutusta lukujen järjestämiseen. Päätimme käyttää jako-ja-voitto (divide-and-conquer) periaatteeseen perustuvaa järjestysalgoritmia, sillä tiesimme niiden soveltuvan luonteeltaan erittäin hyvin rinnakkaiseen toteutukseen. Valitsimme ensiksi Mergesortin, koska sen aikavaativuus on stabiilimbi ja pahimmassakin tapauksessa vain ***n log n***. Aluksi varsinkin pienimuotoista piinaa aiheutti pseudokoodien yleinen käytäntö, jossa taulukoiden ensimmäinen indeksi alkaa ykkösestä. Erittäin hyödylliseksi lähdemateriaaliksi muodoistui Matti Luukkaisen viime kevään Tietorakenteiden kurssin luentokalvot. Algoritmi kuitenkin hahmottui taulukoiden piirtelemisen ja tuskailun jälkeen. Toteutus järjesti sarjallisesti 58 miljoonaa lukua noin 17 sekunnissa, mutta toteutuksesta johtuen se käytti varsin runsaasti muistia. Rekursiivinen Mergesort loi jokaisella rekursiosyvyydellään kaksi uutta aputaulukkoa edellisellä tasolla jaetusta taulukosta. Pohdimme aluksi mahdollisuutta, että voisimme refaktoroida toteutuksen niin, että se käyttäisi maksimissaan vain kaksi kertaa järjestettävän taulukon pituuden verran tilaa. Toinen olisi ollut järjestettävä taulukko ja toinen väliaikainen aputaulukko. Tämä olisi silti tuonut haastetta rinnakkaiseen toteutukseen, sillä Mergesort olisi vaatinut synkroonisuutta taulukon

järjestämiseen. Osataulukoiden järjestäminen olisi pitänyt toteuttaa oikeassa järjestykssä taulukon paloittelemisen jälkeen. Emme löytäneet tähän ongelmaan alkuvaiheessa kovin hyvää ratkaisua tai lähdemateriaalia. Myös in-place Mergesortin toteutukseen tuntui löytyvän jokseenkin huonosti referenssimateriaalia.

Päätimme muuttaa lähestymistapaamme ja toteuttaa järjestämisen Quicksortilla. Tämä siltikin vaikkakin pahimmassa tapauksessa järjestäminen veisi aikavaativuudeltaan n^2. Keskimäärin kuitenkin Quicksortkin toimisi aikavaativuudella $n \log n$, joten Quicksort ei todennäköisesti tulisi olemaan Mergesorttia paljonkaan tai jopa ollenkaan huonompi suorityskyvyltään. Quicksort toi rinnakkaisuutta ajatellen yhden erittäin tärkeän ja hyödyllisen edun, sillä Quicksort jakaa ja järjestää osataulukot täysin riippumatta toisten osataulukoiden järjestämisestä, rinnakkaiseen toteutukseen ei tarvittaisi ollenkaan synkronisuutta tätä ajatellen. Säikeet voisivat hoitaa siis järjestämisen täysin toisistaan riippumatta. Voisimme olla myös varmoja siitä, että järjestäminen ei sotkisi järjestettävää taulukkoa kesken järjestämisen. Toteutimme myös Quicksortin rekursiivisesti. Toteutus ei käytä aputaulukoita, vaan järjestäminen tapahtuu käyttäen itse järjestettävää taulukkoa (niin sanottu In-place Quicksort). Valitsemme taulukon jakokohdan, eli pivotin, taulukon keskikohdasta tai sen lähettyviltä.

Niin Quicksortissa kuin Mergesortissakin on se ongelma, että mitä pienemmäksi järjestettävä osuus taulukosta käy, sitä tehottomammaksi algoritmi käy. Lopulta algoritmi käsittelee tai järjestää vain muutamaa alkiota. Silti jokaisesta vaiheesta täytyy tehdä uusi rekursiosyvyys tai rinnakkaisuutta ajatellen uusi säie. Optimisaatioksi päätimme toteuttaa jo varhaisessa vaiheessa pienten osataulukoiden järjestämisen lisäysjärjestämisellä. Lisäysjärjestäminen toimii erittäin tehokkaasti varsinkin pienillä taulukoilla. Sinnikkään testauksen jälkeen löysimme hyvän raja-arvon, jonka jälkeen järjestäminen kannattaa suorittaa lisäysjärjestämisellä. Sarjallinen algoritmi järjestää alle 50:n pituiset osataulukot lisäysjärjestämisellä. Sarjallinen Quicksort -toteutus käytti noin 11 sekuntia 58 miljoonan luvun järjestämiseen. Optimisaation jälkeen toteutus järjesti luvut vain vähän yli 8 sekunnissa. Optimisaatio osoittautui siis varsin merkittäväksi tekijäksi tehokkuuden kannalta.

Lukujen lukeminen

Sarjallisen Quicksortin jälkeen ryhdyimme toteuttamaan testidatan lukemista. Käytämme lukijassamme (LitteEndianReader) Javan tarjoamaa BufferedInputStreamia. Lukemisen

tehostamiseksi luemme tiedostosta vain 1024 tavua kerrallaan. Muutamme yksitellen jokaisen 64-bitin pituisen binäärin Little-Endian muodosta longiksi käyttäen apuna Javan ByteBufferia. Lukija palauttaa lopuksi viitteen luettujen longien taulukkoon, jonka voimme helposti järjestää algoritmillamme.

Yllätyimme tässä vaiheessa kuinka haasteelliseksi muodostui lukijan saamien lukujen varmistaminen oikeiksi tavujärjestyksen muuntamisen jälkeen. Varsinkin siksi, että Java käyttää Big-Endian formaattia vakiona. Laskimme kuitenkin käsin testidatan binääridatasta muutaman ensimmäisen luvun ja totesimme niiden vastaavan lukijan antamia lukuja sallitun etumerkillisyyden huomioon ottaen.

Tulosten kirjoittaminen

Olemme toteuttaneet ohjelmaan myös yksinkertaisen PrintWriterin, joka tulostaa halutun määrän järjestetyn taulukon tuloksista parametrina annettuun tiedostoon, kuten tehtävänannossa on pyydetty.

Rinnakkainen toteutus

Mietimme pitkään lähestymistapaa Quicksortin rinnakkaisuuden toteuttamiseen. Javan lähdemateriaali rinnakkaisuuden toteuttamiseen toi tähän erittäin suuren avun. Ymmärsimme jo heti alussa, että meidän pitäisi irroittaa sarjallisesta järjestämisestä itse järjestäminen omiksi pieniksi tehtäviksi, joita voisimme ajaa omissa säikessä. Ensimmäinen versiomme rinnakkaisesta toteutuksesta oli hyvin yksinkertainen, mutta toimiva. Normaalisti Quicksort jakaa tehtävän pienempiin osiin rekursion avulla. Rekursion sijaan loimme tehtävälle uuden säikeen. Tämä toteutus toimi kohtuullisesti pienemmillä taulukoilla, mutta oli aivan liian hidas suurempien kanssa. Testidatan kanssa toteutus loi yli 50 tuhatta säiettä, eli kaikki aika meni niiden luomiseen.

Tämän kokeilun jälkeen päätimme tehdä QuicksortTaskin, joka hoitaa itse osataulukon järjestämisen. Tämä luokka toteuttaa Javan Runnable-rajapinnan. Näin voimme luoda luokasta useita instansseja, joita voimme ajaa eri säikeissä. Käytännössä toteutus oli erittäin helppo toteuttaa. Sarjallinen järjestämisalgoritmi täytyi vain siirtää luokan run-metodin alle. Sen lisäksi

rekursion sijaan täytyi uusien osataulukoiden järjestäminen hoitaa luomalla aina uusi säie/ tehtävä itse tehtävän sisällä. Luokan instansseille piti myös pystyä ilmoittamaan tieto missä kohtaa taulukkoa järjestäminen täytyi hoitaa. Luokalle yksinkertaisesti annetaan konstruktorissa osataulukon ensimmäinen ja viimeinen indeksi. Tämän pohjalta tätä tehtävää suorittava säie voi hoitaa osataulukon ja luoda uuden säikeen hoitamaan siitä edelleen hoidettavia osataulukkoja.

Mietimme aluksi, että toteutukseen tarvitsemme jonkinlaisen “työjonon”, josta tehtäviä otettaisiin ja suoritettaisiin vain järkevä määrä kerrallaan, sillä meidän täytyi hallita rekursiota niin, että säikeitä ei luotaisi tuhansia. Tämä olisi erittäin tehotonta. Huomasimme kuitenkin, että työpinon toteuttaminen itse on varsin työläs tehtävä.

Löysimme lopulta Javan API:sta tähän soveltuvan ExecutorServicen. ExecutorService käytännössä kapseloi työjonon (thread pool), jonka kokoa voi rajoittaa. Tehtäviä voi antaa työjonoon niin paljon kuin on tarve, mutta ExecutorService pitää huolen siitä, että vain tietty määrä niitä on ajossa rinnakkaisesti yhtä aikaa. Päädyimme tulokseen, että todennäköisesti optimaalisin määrä säikeitä, joita suoritettaisiin samanaikaisesti, olisi algoritmia suorittavan koneen loogisten prosessoreiden määrä. Tämän tiedon saa helposti Javan runtimelta. ExecutorServicestä löytyi myös kätevä alustusmetodi joka luo rajoitetun työjonon (fixed thread pool).

Toteutimme järjestämisen työtä hallitsevan ConcurrentQuickSort luokan. Tälle yksittäiset QuickSortTaskit ilmoittivat suoritetusta tehtävästä ja antoivat uusia tehtäviä toteuttavaksi. Työtä hallitseva luokka sitten ylläpiti nyt suorituksessa olevien tehtävien määrää ja tarvittaessa aina QuickSortTaskin pyydettäessä antoi ExecutorServicelle uuden taskin hoidettavaksi. Kun uusi taski annettiin ExecutorServicelle, kasvatettiin työssä olevien tehtävien määrää synkronisesti ja aina kun tehtävä kerrottiin suoritetuksi, vähennettiin työssä olevien tehtävien määrää synkronisesti. Niin kauan kuin tehtävien määrä oli suurempi kuin 0, algoritmi tiesi järjestyksen olevan kesken. Kun tehtävien määrä laski nollaan, järjestys oli valmistunut.

Iloksemme huomasimme kuinka helppoa Javassa rinnakkaisuuden toteuttaminen on korkealla tasolla. Jos jokin metodi teki jotain, jossa Mutex-ehdon täytyi toteutua, riitti metodille vain ilmaista synchronized määre. Näin Java hoitaa sen, että missään vaiheessa kaksi säiettä ei yritä asettaa tai noutaa samaa muuttujaa. Käytimme toteutuksessa myös busy-wait looppia järjestämisen tilan seuraamiseen. Käytännössä siis aina tehtävän valmistuttua laskimme tehtävämäärälaskuria ja ilmoitimme notifylla kaikkia niitä säikeitä, jotka odottivat tehtävämäärälaskurin muutoksia. Jos tehtävämäärä oli vieläkin yli 0, jäimme odottamaan taas tehtäviä.

Loppujen lopuksi toteutuksesta tuli erittäin yksinkertainen, mutta vaikka toteutus järjesti rinnakkaisesti 58 miljoonaa lukua oikein, oli se jopa hitaampi kuin sarjallinen toteutus, joka järjesti luvut 8 sekunnissa. Rinnakkainen toteutus kulutti järjestämiseen noin 9 sekuntia. Tässä vaiheessa ryhmässämme iski pieni epätoivo.

Sinnikkään testaamisen jälkeen havaitsimme, että vain kasvattamalla raja-arvoja, jonka jälkeen järjestimme osataulukon lisäysjärjestämisellä, nopeutui rinnakkainen toteutus merkittävästi. Kun raja-arvo oli asetettu 700 lukuun, järjestäminen kesti vain noin 3,8-4,2 sekuntia. Myöhemmin havaitsimme, että optimaalisin raja-arvo sijaitsi noin 1000 luvun kohdalla. Tällöin saimme, jopa ajoittain Ukko laskentaklusterilla järjestämisarvoksi parhaimmillaan vain vähän yli 3 sekuntia. *(Sivuhuomautuksena, on yllättävää miten paljon laskenta-ajat voivat Ukkolla vaihdella. Täysin sama järjestäminen saattoi jopa heittää +-1 sekunnin eri ajokerralla.)* Vaikka tämä oli vielä kaukana viime vuotisesta huipputuloksesta 1,72 sekuntia, oli toteutuksemme jo yli 50% nopeampi sarjallista toteutusta. Emme silti olleet valmiita tyytymään tulokseen, sillä hyvähenkinen kilpailu ja lisäpisteet nostivat motivaatiotamme saavuttaa vielä parempi aika.

Ryhdyimme tällöin pitämään 1000 luvun raja-arvoa liian suurena, varsinkin jos järjestettävien taulukkojen koko olisi merkittävästi pienempi. Rinnakkainen toteutus järjesti esimerkiksi keskimäärin yhden ja viiden miljoonan kokoiset taulukot keskimäärin hieman hitaammin kuin sarjallinen toteutus johtuen juuri raja-arvon korkeasta luvusta. Päätimme, että raja-arvosta täytyisi tehdä sopeutuva.

Toteutimme tämän jälkeen neljännen ratkaisun rinnakkaiseen järjestämiseen. Tämä sattuu myös olemaan meidän viimeinen ja tehokkain toteutus, jonka olemme palauttaneet tämän raportin ohella. Päätimme tässä ratkaisussa hallita säikeitä itse. Käytämme Javan säikeiden start ja join-toiminnallisuutta, ja rajoitamme rekursiota tässäkin toteutuksessa loogisten prosessoreiden määrän perusteella. Toteutus käyttää QuickSorttia, jonka rekursiosyvyyttä ja säikeiden määrää on rajoitettu loogisten prosessoreiden perusteella. Kun rekursio on niin syvä, että sen matalimmalla tasolla on säikeitä loogisten prosessoreiden verran, ajetaan javan omaa järjestämismetodia. Rekursion syvyyden kasvattaminen ei tässä toteutuksessa vaikuta juuri ollenkaan suorituskykyyn. Syvyyden pienentäminen taas hidastaa järjestämistä huomattavasti. Luotuja säikeitä myös odotetaan niin kauan kuin ne ovat valmistuneet. Näin hallitsemme luotujen säikeiden määrää. Säikeitä muodostuu hieman yli loogisten prossesoreiden määrän. Kyseisestä toteutuksesta tuli hieman nopeampi kuin ExecutorServiceä käyttävä toteutus, mutta se ennenkaikkea suoriutuu huomattavasti

nopeammin myös pienemmistä taulukoista.

Olemme lisänneet projektin palautuksen yhteyteen osan aikaisemmista toteutuksista selityksen ja mielenkiinnon tueksi.

Käyttöohjeet

Projektimme käyttää Maven-projektinhallintaa. Olemme kuitenkin ohjeiden mukaan toteuttaneet start.sh ja compile.sh skriptit projektin ajamiseen. Skriptit löytyvät projektin juuresta. Ukkolta löytyy Maven, joten suoritus ei ole ongelma. Ensimmäinen kääntö- ja suorituskerta voi kuitenkin viedä jonkin aikaa, sillä Mavenin tarvitsee ladata automaattisesti sen tarvitsevat riippuvuudet. Komennot ajetaan projektin juuressa.

Ajattavan jar-paketin kääntäminen
./compile.sh

Rinnakkaisen järjestämisen ajaminen testidatalla (tulostaa ensimmäiset 1000 lukua projektin juureen results-parallel.txt-tiedostoon, säädettävissä skriptissä)
./start.sh

Sarjallisen järjestämisen ajaminen testidatalla (tulostaa ensimmäiset 1000 lukua projektin juureen results-serial.txt-tiedostoon, säädettävissä skriptissä)
./start-serial.sh

Projektin testit voi myös ajaa komennolla. Komento ajaa sekä sarjallisen että rinnakkaisen toteutuksen testit ja vertaa niistä saatuja tuloksia Javan omaan Arrays.sortin tulokseen. Testien ajaminen vie jonkin aikaa.
mvn test

Testaus

Toteutimme jo heti alusta lähtien JUnit-yksikkötestejä toteutuksen järjelliseen ja toistuvaan testaukseen. Testit, jotka ovat nähtävissä palautetun projektin lähdekoodista,

testaavat toteutettuja järjestysalgoritmeja erilaisilla rajatapauksilla. Aluksi testasimme järjestystä vain pienillä taulukoilla, mutta jo hyvin varhaisessa vaiheessa projektia lisäsimme testaukseen satunnaisen 58 miljoonan luvun taulukon sietokykytestauksen. Tällä testasimme pystyikö toteutuksemme ylipäätään järjestämään 58 miljoonaa lukua. Lopulta toteutimme myös annetun testidatan järjestämisen. Järjestämme taulukon myös Javan API:n tarjoaman Arrays.sort-metodin avulla. Vertaamme lopuksi toteutuksemme ja Javan järjestysalgoritmin tuloksia, jotta voimme todeta suurella varmuudella todeta järjestyksemme oikeellisuuden päteväksi.

Oikeellisuus, suorituskyky, skaalautuvuus

Quicksortista johtuen rinnakkaisuuden hallinta on varsin helppoa. Säikeet eivät tarvitse synkronisuutta keskenään ja voivat toteuttaa järjestämisen toisista riippumatta. Tämän pohjalta voimme sanoa synkronisuuden säilyvän. Myös kattava testaus ja järjestyksen tulosten vertaaminen Javan API:n tarjoamiin järjestysmetodeihin varmentaa algoritmin toimivuutta. Olemme, myös aikaisemmissa toteutuksissa, huolehtineet rinnakkaisuuden vaatimuksista jo yllä mainituin keinoin. Toteutuksessamme oikeastaan tärkeintä on ollut löytää tapa tehokkaasti rajoittaa säikeiden määrää ja optimoida Quicksortin toimintaa. Olemmekin siis hyvissä mielin todenneet toteutuksen toimivaksi.

Vaikka suorituskyky ei ylläkään viime vuoden huipputulokseen 1,72s, on algoritmimme suorityskyky noin 50% nopeampi kuin sarjallinen toteutus. Useista optimisaatioista huolimatta, emme kuitenkaan usko päässemme ennätykseen kiinni. Tosin emme ole siitä erityisen harmissammekaan. Tähän dokumenttiin on liitetty myös eri toteutuksiemme keskimääräiset suoritusajat Ukko laskentaklusterilta. Sarjallisen ja rinnakkaisen algoritmimme suhde on noin 7500ms/3600ms eli rinnakkainen on noin kaksi kertaa niin nopea kuin sarjallinen.

Koska toteutuksemme selvittää dynaamisesti loogisten prosessoreiden määrän, skaalautuu algoritmi hyvin eri alustoille. Mitä enemmän loogisia prosessoreita on, sitä tehokkaammin järjestäminen todennäköisesti toimii. Tästä tietenkään emme voi olla täysin varmoja. Algoritmi toimii hyvin myös vähätehoisemmilla koneilla, joissa prosessoreita on vähemmän.

Yhteenveto

Ryhmämme suoriutui projektista kohtalaisen hyvin. Pidimme aina tarvittaessa jopa useamman tunnin pituisia palavereja. Kuitenkin, koska rinnakkainen ohjelmointi oli kaikille uusi asia, vei projekti varsin paljon aikaa. Olemme kuitenkin oppineet projektin edetessä paljon uusia asioita. Varsinkin Javan erittäin ohjelmoijaystävällinen rinnakkaisuuden hallinta on yllättänyt ryhmämme positiivisesti.

Rinnakkaisuus on hieman kaksipiippuinen asia. Teoriassa rinnakkaisuuden hallinta ja sen takana olevat ideologiat ovat varsin yksinkertainen asia, mutta kuten olemme projektin toteutuksessa huomanneet, asiat eivät kuitenkaan käytännössä välttämättä ole niin helppoja. On kuitenkin selvää, että rinnakkaisuus tuo erittäin suuria etuja tiettyihin tehtäviin ja siksi on myös kriittisesti tärkeää varsinkin nykymaailmassa, jossa lähes jokainen kone sisältää moniytimisiä prosessoreita tai jopa useita prosessoreita. On siis erittäin mukavaa, että olemme saaneet projektistamme paljon irti.

Lähteet

http://docs.oracle.com/javase/tutorial/essential/concurrency/
http://www.cs.helsinki.fi/group/java/k11/tira/tira2011.pdf
http://en.wikipedia.org/wiki/Quicksort

Liitteet

Suorituskyvyn testitulokset

ExecutorService toteutus

```
tshiltun@ukko018:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin 1
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2092ms.
Threshold: 50 Median time: 9198
Threshold: 100 Median time: 6229
Threshold: 150 Median time: 5469
Threshold: 200 Median time: 4877
Threshold: 250 Median time: 4572
Threshold: 300 Median time: 4407
```

```
Threshold: 350 Median time: 4164
Threshold: 400 Median time: 4146
Threshold: 450 Median time: 3860
Threshold: 500 Median time: 3747
Threshold: 550 Median time: 3693
Threshold: 600 Median time: 3602
Threshold: 650 Median time: 3545
Threshold: 700 Median time: 3508
Threshold: 750 Median time: 3472
Threshold: 800 Median time: 3438
Threshold: 850 Median time: 3382
Threshold: 900 Median time: 3353
Threshold: 950 Median time: 3279
Threshold: 1000 Median time: 3326
Threshold: 1050 Median time: 3286
Threshold: 1100 Median time: 3300
Threshold: 1150 Median time: 3333
Threshold: 1200 Median time: 3431
Threshold: 1250 Median time: 3297
Threshold: 1300 Median time: 3360
Threshold: 1350 Median time: 3443
Threshold: 1400 Median time: 3403
Threshold: 1450 Median time: 3434
Threshold: 1500 Median time: 3377
Threshold: 1550 Median time: 3328
Threshold: 1600 Median time: 3445
Threshold: 1650 Median time: 3422
Threshold: 1700 Median time: 3382
Threshold: 1750 Median time: 3362
Threshold: 1800 Median time: 3427
Threshold: 1850 Median time: 3477
Threshold: 1900 Median time: 3491
Threshold: 1950 Median time: 3467
Threshold: 2000 Median time: 3460
```

ExecutorService, jossa järjestetään thresholdia pienemmät taulukot järjestetään javan sortilla insertion sortin sijaan

```
tshiltun@ukko210:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin 1
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2115ms.
Threshold: 50 Median time: 9571
Threshold: 100 Median time: 7418
Threshold: 150 Median time: 5254
Threshold: 200 Median time: 4777
Threshold: 250 Median time: 4559
Threshold: 300 Median time: 4509
```

```
Threshold: 350 Median time: 4190
Threshold: 400 Median time: 4087
Threshold: 450 Median time: 3951
Threshold: 500 Median time: 3861
Threshold: 550 Median time: 3870
Threshold: 600 Median time: 3722
Threshold: 650 Median time: 3747
Threshold: 700 Median time: 3652
Threshold: 750 Median time: 3728
Threshold: 800 Median time: 3148
Threshold: 850 Median time: 3110
Threshold: 900 Median time: 3494
Threshold: 950 Median time: 3700
Threshold: 1000 Median time: 3629
Threshold: 1050 Median time: 3247
Threshold: 1100 Median time: 3429
Threshold: 1150 Median time: 3394
Threshold: 1200 Median time: 2893
Threshold: 1250 Median time: 3027
Threshold: 1300 Median time: 3281
Threshold: 1350 Median time: 3272
Threshold: 1400 Median time: 2836
Threshold: 1450 Median time: 2911
Threshold: 1500 Median time: 3255
Threshold: 1550 Median time: 3268
Threshold: 1600 Median time: 3237
Threshold: 1650 Median time: 3199
Threshold: 1700 Median time: 3044
Threshold: 1750 Median time: 3154
Threshold: 1800 Median time: 3316
Threshold: 1850 Median time: 3124
Threshold: 1900 Median time: 3154
Threshold: 1950 Median time: 3123
Threshold: 2000 Median time: 2975
```

ExecutorService toteutus, jossa sarana-alkio on kolmen alkion mediaani

```
tshiltun@ukko210:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin 1
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2098ms.
Threshold: 50 Median time: 8700
Threshold: 100 Median time: 6726
Threshold: 150 Median time: 4920
Threshold: 200 Median time: 4978
Threshold: 250 Median time: 4625
Threshold: 300 Median time: 4430
Threshold: 350 Median time: 4316
Threshold: 400 Median time: 4116
Threshold: 450 Median time: 4066
```

```
Threshold: 500 Median time: 3925
Threshold: 550 Median time: 4031
Threshold: 600 Median time: 3831
Threshold: 650 Median time: 3826
Threshold: 700 Median time: 3741
Threshold: 750 Median time: 3839
Threshold: 800 Median time: 3806
Threshold: 850 Median time: 3864
Threshold: 900 Median time: 3593
Threshold: 950 Median time: 3635
Threshold: 1000 Median time: 3640
Threshold: 1050 Median time: 3553
Threshold: 1100 Median time: 3590
Threshold: 1150 Median time: 3542
Threshold: 1200 Median time: 3454
Threshold: 1250 Median time: 3523
Threshold: 1300 Median time: 3474
Threshold: 1350 Median time: 3632
Threshold: 1400 Median time: 3524
Threshold: 1450 Median time: 3521
Threshold: 1500 Median time: 3522
Threshold: 1550 Median time: 3560
Threshold: 1600 Median time: 3578
Threshold: 1650 Median time: 3507
Threshold: 1700 Median time: 3610
Threshold: 1750 Median time: 3625
Threshold: 1800 Median time: 3558
Threshold: 1850 Median time: 3580
Threshold: 1900 Median time: 3625
Threshold: 1950 Median time: 3681
Threshold: 2000 Median time: 3634
```

Lopullinen toteutus

```
tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2100ms.
Sorting longs...
Sorted longs in 3100ms.

tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2130ms.
Sorting longs...
Sorted longs in 3152ms.

tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
```

```
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2101ms.
Sorting longs...
Sorted longs in 3193ms.


tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2103ms.
Sorting longs...
Sorted longs in 3157ms.

tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2104ms.
Sorting longs...
Sorted longs in 3198ms.

tshiltun@ukko057:~/bin/RIO-sort/Sorter$ java -jar target/Sorter-1.0-
SNAPSHOT.jar /cs/fs/home/kerola/rio_testdata/uint64-keys.bin
Reading file "/cs/fs/home/kerola/rio_testdata/uint64-keys.bin"...
Read file in 2165ms.
Sorting longs...
Sorted longs in 3168ms.
```