Tampereen yliopisto

Jussi Lähteenmäki

RINNAKKAISTEN JÄRJESTYSALGORITMIEN AJAN- JA ENERGIANKÄYTTÖ

Kandidaatin työ
Tieto- ja sähkötekniikan tiedekunta
Tarkastajat: Prof. Terhi Kilamo
Heinäkuu 2020

TIIVISTELMÄ

Jussi Lähteenmäki: Rinnakkaisten järjestysalgoritmien ajan- ja energiankäyttö
Kandidaatin työ
Tampereen yliopisto
Tietotekniikan kandidaatin tutkinto-ohjelma
Heinäkuu 2020

Järjestysalgoritmit ovat laaja-alaisesti käytettyjä erilaisissa sovelluksissa ja niitä on tutkittu laaja-alaisesti jo vuosikymmenien ajan varsinkin ajankäytön kannalta. Koska prosessorien suoritusnopeudet alkavat saavuttaa teoreettista huippuaan, rinnakkaislaskennasta on tullut tärkeä työkalu suoritusnopeuden kasvattamiseen. Tämän työn tavoitteena on selvittää mitkä järjestysalgoritmit hyötyivät suoritusnopeudeltaan rinnakkaistetusta toteutuksesta, sekä mikä oli rinnakkaistettujen algoritmien energiankäyttö verrattuna sarjallistettuihin versioihin.

Työ on kirjallisuustutkielma, jossa käydään läpi tutkimuksia rinnakkaistetuista järjestysalgoritmeista, jotka on toteutettu eri teknologioilla sekä eri alustoilla. Käytettyjä teknologioita olivat OpenCL, CUDA sekä OpenMP, ja tukitut algoritmit olivat bitonic sort, odd-even sort, quicksort, rank sort sekä shell sort. Työssä käydään läpi rinnakkaislaskennan sekä järjestysalgoritmien teoriaa, jonka jälkeen vertaillaan tutkimusten tuloksista mitkä algoritmit ja millä syöteko'oilla hyötyivät rinnakkaistetusta toteutuksista, miten ne vertautuivat toisiinsa ja mikä oli rinnakkaistetun version energiankäyttö verrattuna sarjallistettuun versioon. Työssä huomattiin, että näytönohjaimia hyödyntävien järjestysalgoritmien energiankulutuksesta on niukalti tutkimusta.

Tutkimuksista selvisi, että varsinkin algoritmin datariippuvuudet vaikuttivat rinnakkaistamisesta saatuun hyötyyn. Esimerkiksi bitonic sort nopeutui huomattavasti rank sortiin ja odd-even sortiin verrattuna OpenCL:llä toteteutetuilla näytönohjaimia hyödyntävissä sovelluksissa kaikilla syöteko'oilla. Esimerkiksi 217:llä alkiolla bitonic sortin suoritusaika oli alle sekuntin verrattuna odd-even sortin sekä ranked sortin yli 100:aan sekuntiin. OpenMP:llä toteutetuista moniytimistä prosessoria hyödyntävistä algoritmeista quicksort hyötyi eniten vasta tarpeeksi suurilla syöteko'oilla. 10000:lla alkiolla quicksortin ajankäyttö kasvoi neljää ja kahdeksaa ydintä hyödyntäessä. Nopeutus kuitenkin kasvoi alkiomäärän kasvaessa yli 100000:een.

Energiankäytöstä löytyi vain OpenMP:llä toteutettu moniytimistä prosessoria hyödyntäviä algoritmeja tutkinut tutkimus. Tutkimuksessa todettiin, että jos algoritmin rinnakkaistoteutus oli ajallisesti tehokkaampi kuin sarjallinen tutkimus, niin se oli myös energiankäytöltään tehokkaampi. Syyksi arvioitiin, että koko tietokoneen energiankäyttö on huomattavaan suuri osa verrattuna useamman ytimen energiankäyttöön.

Avainsanat: Järjestysalgoritmit, rinnakkaislaskenta, energiankäyttö, ajankäyttö, tehokkuus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

SISÄLLYSLUETTELO

1 Johdanto 1
2 Rinnakkaislaskenta 2
3 Järjestysalgoritmit 4
4 Rinnakkaistetut järjestysalgoritmit 5
4.1 Käytetyt toteutukset ja testialustat 5
4.2 Ajankäyttö 6
4.3 Energiankäyttö 10
5 Yhteenveto 12
Lähteet 13

LYHENTEET JA MERKINNÄT

CUDA	Nvidian kehittämä ohjelmointirajapinta näytönohjaimille
OpenCL	Khronos Groupin kehittämä ja ylläpitämä ohjelmistokehys rinnakkaislaskentaan
OpenMP	OpenMP Architecture Review Boardin ylläpitämä rajapinta rinnakkaislaskennalle C, C++ ja Fortran kielille

1 JOHDANTO

Noin 30 vuoden ajan järjestysalgoritmit ovat olleet hyvin laajasti tutkittuja ja niiden käyttö on laaja-alaista eri sovelluksissa. Näytönohjainten käytön yleistyessä yleishyödyllisten sovellusten käytössä ovat myös rinnakkaislaskentaa hyödyntävien järjestysalgoritmien toteutukset yleistyneet. [1, p. 173] Ei voida kuitenkaan olettaa, että rinnakkaislaskennan nopeutus on suoraan verrannollinen käytettyjen laskentaytimien määrään, koska syötteen siirtäminen prosessorilta oheislaitteelle vie aikaa, eikä laskennan jakaminen oheislaitteen laskentaytimille ole triviaalia. On myös otettava huomioon, että syötekokojen kasvaessa ei voida keskittyä pelkästään ajalliseen tehokkuuteen, vaan myös energiankäytön tehokkuuteen tulee kiinnittää huomiota[2, p. 1]. Kun laskentaan lisätään laitteita, niin myös energiankäyttö kasvaa. Energiankäyttö on ollut monien palvelinkeskuksien sekä korkean suorituskyvyn laskentasovelluksien päällimmäisenä huolena [2, p. 1]. Tämän työ selvittää vastaukset seuraaviin kysymyksiin:

1. Mitkä algoritmit hyötyvät rinnakkaislaskennasta?
2. Mikä täytyy olla syötekoon, että rinnakkaislaskennasta saadaan hyötyä?
3. Kuinka paljon näytönohjaimella ajettu algoritmi kuluttaa energiaa verrattuna tavanomaisempaan sekventiaaliseen toteutukseen?

Työ tutkii miten eri järjestysalgoritimien toteutukset hyötyvät rinnakkaislaskennan käytöstä ja miten näytönohjaimen käyttö vaikuttaa sovelluksen energiankäyttöön. Luvussa 2 käydään läpi rinnakkaislaskennan teoriaa sekä heterogeenisten laskenta-alustojen toimintaa. Luvussa 3 käydään läpi verrattavat järjestysalgoritmit ja niiden toteutukset, minkä jälkeen luvussa 4 vertaillaan rinnakkaislaskentaa käyttävien toteutuksien tehoa niin ajallisesti kuin energiankäytön puolesta. 5. luku on yhteenveto työstä ja sen tuloksista.

2 RINNAKKAISLASKENTA

Rinnakkaislaskennalla tarkoitetaan, että usea laskentaydin suorittaa samaan aikaan saman tehtävän eri osia. Rinnakkaisuus voi olla bitti-, data-, käsky- tai tehtävätasolla. Rinnakkaislaskennan motiivina on nopeuttaa työn suoritusaikaa. Nopeutusta rajoittaa kuitenkin Amdahlin laki. Jos P on rinnakkaistettava osuus ohjelmasta ja (1 - P) on sarjallistettuna pysyvä osuus ohjelmasta, niin suurin mahdollinen nopeutus, joka saadaan aikaan N:llä prosessorilla on,

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

josta nähdään, ettei rinnakkaislaskennan suoritus nopeudu lineaarisesti saatavilla olevien laskentaydinten mukaan. Nopeutusta rajoittaa myös rinnakkaistettavan koodin osuus kokonaisuudesta. Jos 95 %:ia koodista on rinnakkaistettavissa, niin teoreettinen suurin mahdollinen nopeutus äärettömällä määrällä prosessoreja on 20-kertainen alkuperäiseen verrattuna. Realistisemmalla kahdeksalla ytimellä suurin mahdollinen nopeutus on kuusinkertainen. [2, p. 2]

Kun ohjelmaa jaetaan rinnakkaistettavaksi, niin on tärkeää määritellä termit siitä, miten rinnakkaistettavuutta voidaan mitata. Zecena ja muut määrittelevät tehtävien rakeisuuden (*granularity*): kuinka paljon laskentaa tehtävän pitää tehdä, että se on valmis. Tehtävän karkeus on siis tapa mitata sen laskennan määrää. Erittäin suureen hienojakoisuuteen ei kannata pyrkiä yleiskustannuksien kasvun takia.

Heterogeenisellä alustalla tarkoitetaan tietokonetta, joka suorittaa ohjelmaa isäntänä toimivan prosessorin lisäksi erikoistetulla laitteella, jolle siirretään prosessorilta dataa suoritettavaksi. Suoritettuaan laskennan laskenta-alusta palauttaa tuloksen prosessorille. Esimerkkinä tästä ovat näytönohjaimet. Dongarra ja Lastovetsky määrittävät, että heterogeenisille alustoille rinnakkaistetussa ja hajautetussa laskennassa pätee aina, että niillä on useampi prosessori, sekä kommunikointiverkko, joka yhdistää prosessorit. Järjestelmän täytyy myös olla omistautunut vain yhden sovelluksen suorittamiseen, eli kaikkien sen resurssien tulee olla varattu saman sovelluksen käyttöön. Tämä voidaan toteuttaa jakamalla järjestelmä loogisesti alijärjestelmiin, jotka eivät kuvaudu fyysisesti samoille osille. [3, p. 3]

Tässä työssä keskitytään *vendor-designed* -tyyppisistä heterogeenisistä järjestelmistä näytönohjaimiin (GPU). Tämänkaltaiset rakenteet ovat Dongarran ja Lastovetskyn mukaan avanneet uusia mahdollisuuksia suoritusnopeudessa sekä energiankulutuksessa, mutta eivät ilman haasteita sovelluksien käyttäjille ja ohjelmistokehittäjille [3, p. 6]. Khan

sekä muut määrittelevät GPU:t useampiytimisinä prosessoreina, jotka tukevat tuhansia yhtäaikaisessa ajossa olevia säikeitä [1, p. 173].

GPU:t ovat ohjelmoitavissa tekijän tarjoaman rajapinnan kautta, kuten NVIDIAn CUDA, ATI:n FireStream ja Khronos Groupin määrittelemä OpenCL. Vaikka säikeiden hallinta toteutetaan laitteistotasolla, niin ohjelmistokehittäjän tehtävä on jakaa kokonaistyö yhdeksi tai useammaksi työtehtäväksi [1, p. 173]. Koska näytönohjain on erillinen laite, eivätkä osa isäntäprosessoria, täytyy ottaa huomioon myös tiedonsiirtoon kuluva aika. Tämän takia tulee isäntälaitteen ja näytönohjaimen välistä datan synkronointia vältellä sen ollessa mahdollista. [1, p. 174].

3 JÄRJESTYSALGORITMIT

Järjestysalgoritmilla tarkoitetaan ohjelmaa, joka ottaa syötteenä n-alkioisen taulukon A ja järjestää sen alkiot järjestykseen [4, p. 5]. Zecena ja muut toteavat, että järjestysalgoritmit ovat kaikkialla läsnä tietotekniikassa ja ovat tämän vuoksi hyvin tutkittuja ajankäytön suhteen [2, p. 1]. Tähän työhön koottuja rinnakkaisesti toteutettuja algoritmeja ovat Bitonic Sort, OddEven Sort, Shellsort sekä Quicksort [1] [5], [2].

OddEven Sort on iteratiivinen järjestysalgoritmi, jonka ajankäyttö on luokkaa $O(n^2)$ joka on varsin tehokas pienillä datamäärillä[2, p. 1]. Algoritmi vaiheittain vuorotellen vertailee kaikkia parillisissa tai parittomissa indekseissä olevia alkioita niiden vierekkäisissä indekseissä oleviin alkiohin. Joka vaiheessa vertailuväliä kasvatetaan suuremmaksi, kunnes se kattaa koko taulukon. [2, p. 2]

Shellsort on tehokkuudeltaan $O(n^{3/2})$ ja toimii tehokkaasti keskisuurista suurien datamäärien kanssa [2, p. 1]. Siinä jokaista taulukon alkiota verrataan hyppien aina tietyn välin verran taulukkoa ja verraten jokaisessa näissä indekseissä olevaan alkioon. Tämän jälkeen seuraavaa alkiota verrataan samalla väliarvolla vastaaviin alkiohin. Zecenan ja muiden toteutuksessa väli alkoi noin yhden kolmasosan kokoisena taulukosta ja pienentyi yhteen. [2, p. 2]

Bitonic sort on algoritmi, joka järjestää mielivaltaisen sekvenssin bitoniseksi sekvenssiksi (*bitonic sequence*) ja on ajankäytöltään luokkaa $O(n\log(n)^2)$. Bitoninen sekvenssi joko monotonisesti kasvaa tai pienenee saavuttaen jonkun maksimin tai minimin, minkä jälkeen sekvenssi taas monotonisesti kasvaa tai pienenee. Esimerkiksi lukujonot 3 5 8 9 7 4 2 1 ja 5 8 9 7 4 2 1 3 ovat bitonisia. Ensimmäinen kasvaa 3:sta 9:ksään ja laskee tämän jälkeen 1:een. Toinen luku sarja kasvaa 5:stä 9:ään, minkä jälkeen se pienenee 1:een ja lopuksi nousee takaisin 3:een. [1, p. 175]

Quicksort on hajota ja hallitse -tyyppinen algoritmi, joka jakaa rekursiivisesti taulukon kahteen alitaulukkoon [4, p. 170]. Jokainen vasemman alitaulukon alkio on pienempi kuin(*pivot*-alkio ja oikean alitaulukon alkiot ovat suurempia tai yhtäsuuria kuin *pivot*-alkio. Tämän jälkeen jakaminen suoritetaan rekursiivisesti syntyneille alitaulukoille kunnes alitaulukkojen koko on yksi ja alkuperäinen taulukko on täten järjestyksessä. [2, p. 2]

4 RINNAKKAISTETUT JÄRJESTYSALGORITMIT

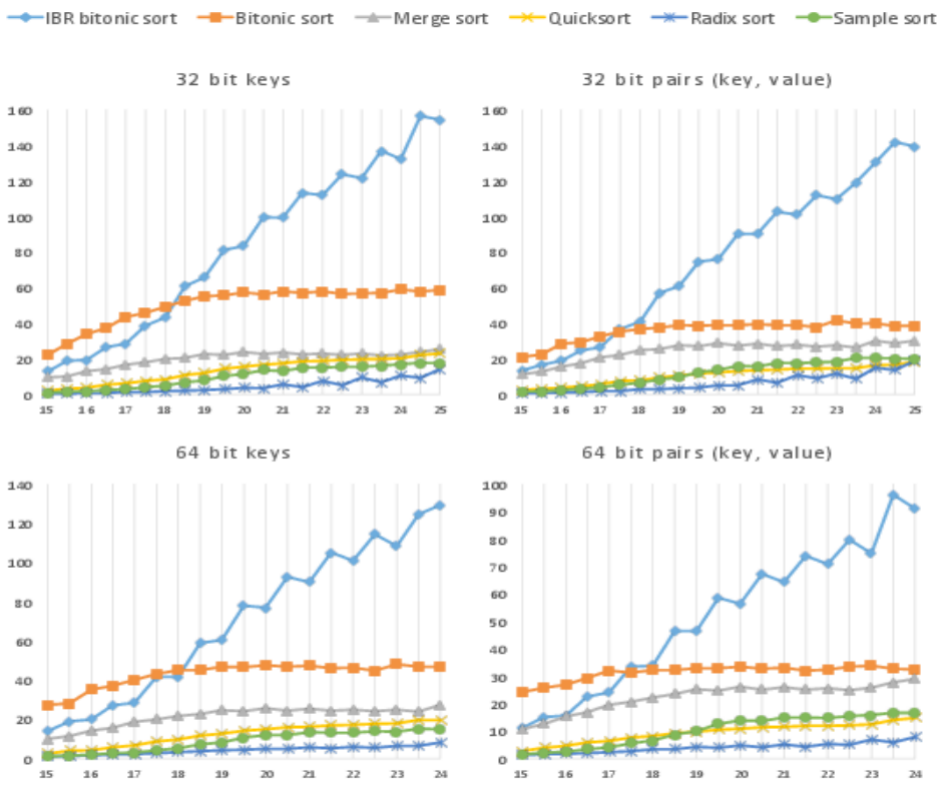
Kun järjestysalgoritmia toteutetaan rinnakkaistetusti näytönohjaimella, on otettava huomioon miten ohjelma jaetaan eri laskentaytimille, ja että datan synkronointi näytönohjaimen sekä isäntäprosessorin välillä minimoidaan tiedonsiirtoon kuluvan ajan takia. Jotkin ohjelmointikehykset kuten OpenCL eivät mahdollista muistin dynaamista varausta kernelien sisällä, joten täytyy muistinkäytöstä olla tieto jo sovellusta ohjelmoitaessa. [1, p. 176] Myös moniytimisellä prosessorilla prosessoreiden välinen datan synkronointi vie aikaa [2, p. 3]. Koska tämän työn tutkimuskysymyksenä on saman algoritmin rinnakkaistoteutuksen vertaaminen sekventiaaliseen toteutukseen, ei eri tutkimusten tuloksia verrata toisiinsa.

4.1 Käytetyt toteutukset ja testialustat

Khan ja muut toteuttivat rinnakkaistetut algoritminsa OpenCL:llä ja käyttivät useampaa eri testialustaa. Testialustoina toimivat NVIDIA Quadro 6000, NVIDIA GeForce GTX 260 ja NVIDIA GeForce GT 320M. Sekventiaaliset algoritmit suoritettiin 2,66 GHz:n Intel Core2Quad CPU 8400:lla, jolla oli 4GB RAM. Syötteenä käytettiin prosessorilla satunnaisesti generoituja liukulukuja, jotka noudattivat joko tasa- tai eksponenttijakaumaa. Ajankulutukseen laskettiin vain kernelin taulukon järjestämiseen käyttämä aika, eikä esimerkiksi muistin kopiomista. [1, p. 176]

Zecena ja muut käyttivät testialustana kahta neliytimistä AMD 2380 Opteron prosessoria, jotka jakoivat muistinsa. Ohjelmointikielenä toimi C/C++, jonka kanssa käytettiin OpenMP-laajennuksen pragmoja rinnakkaislaskennan mahdollistamiseen. Kaikki algoritmit suoritettiin dynaamisesti varatuille taulukoille, joille annettiin satunnaisesti generoitu syöte. [2, p. 2] Testit suoritettiin rinnakkaistetuille toteutuksille käyttäen 1:ä, 2:a, 4:ää sekä 8:aa ydintä [2, p. 3].

Bozidar ja Dobravec käyttivät CUDA:a ja heidän rinnakkaistettujen algoritmiensa alustana toimi GeForce GTX670 näytönohjain 2GB:n muistilla. Tuloksia verrattiin Intel Core i7-3770K 3.5GHz prosessorilla suoritettuihin sekventiaalisiin versiohin. Syötteinä käytettiin 32-bittisiä avaimia, 32-bittisiä avain-arvo pareja, 64-bittisiä avaimia sekä 64-bittisiä avain-arvo pareja. Syötteitä kehitettiin eri jakaumilla. [5, p. 5]



Kuva 4.1. *Algoritmien nopeutukset eri syötteillä Bozidarin ja Dobravecin toteutuksilla. X-akselilla syötteen koko binäärisenä logaritmina, y-akselilla nopeutus. [1, p. 176]*

4.2 Ajankäyttö

Bitonic sort osoittautui hyvin nopeutuvaksi rinnakkaistettuna Bozidar ja Dobravecin sekä Khanin ja muiden totetuksissa. Kuten kuvasta 4.1 nähdään, että Bozidarin ja Dobravecin toteutuksen nopeutus kasvoi 32-bittisillä avaimilla syötteen kokoon 2^{19} asti ja pysyi tasaisesti yli 50-kertaisena. Muilla syötteillä nopeutus ei ollut näin korkea. [5, p. 6] Khan ja muut toteavat, että Bitonic Sortin datariippumattomuus soveltuu hyvin näytönohjaimille. Quadro 6000 -näytönohjaimella Bitonic Sort järjesti 2^{25} kokoisen taulukon 1,97:ssä sekunnissa ja 2^{15} kokoisen 0,0012:ssa sekunnissa. GTX 260:lla ja GT 320M:llä 2^{15} kokoisen taulukon järjestäminen kesti 0.003 sekuntia sekä 0.016 sekuntia vastaavasti. Keskimääräiseksi nopeutukseksi Khan ja muut saivat Bitonic Sortille 18,93 Quadro 6000:lla ja 10,11 GTX 260:llä. Khanin ja muiden Bitonic sortin ajoajat eri alustoilla löytyy kuvasta 4.2. [1, p. 176]

Khan ja muut totesivat, että OddEven sort sopeutuu myös hyvin näytönohjaimella toteutettavaksi datariippumattomuutensa ansiosta. 2^{15} kokoisella syötteellä OddEven Sort järjesti taulukon Quadro 6000:lla 0,23:ssa sekunnissa, GTX 260:lla 0,7:ssä sekunnissa ja GT 320M:llä 2,17:ssä sekunnissa. Keskimääräinen nopeutus Khanin ja muiden OddEven Sortille oli 2,73 GTX 260:lla ja 12,11 Quadro 6000:lla. Khanin ja muiden OddEven sortin ajoajat eri alustoilla löytyy kuvasta 4.3. [1, p. 176] Zecenan ja muiden moniytimisellä prosessorilla suorittama versio OddEven sortista oli varsin tehoton. Muihin ryhmän

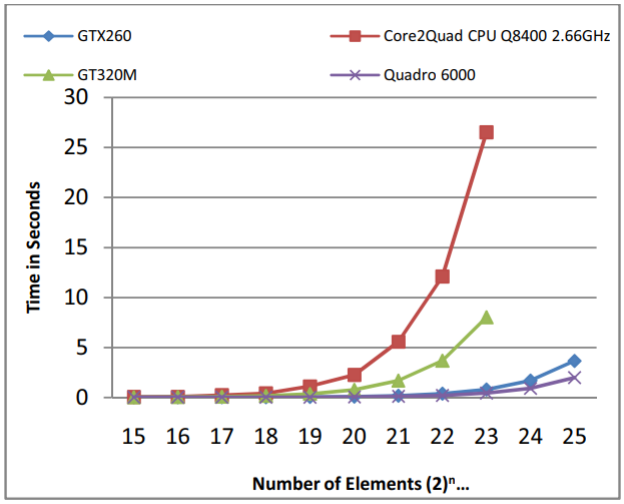


Figure 3. Sorting Time (Bitonic Sort)

Kuva 4.2. *Bitonic sortin ajankulutus eri alustoilla Khanin ja muiden toteutuksilla [1, p. 176].*

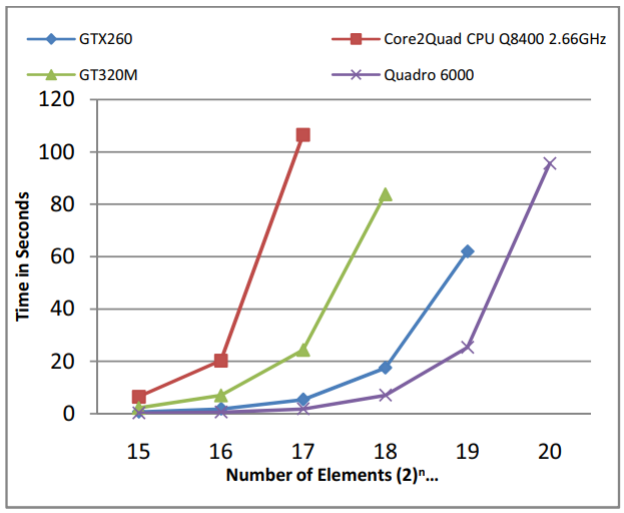


Figure 4. Sorting Time (Odd-Even Sort)

Kuva 4.3. *OddEven sortin ajankulutus eri alustoilla Khanin ja muiden toteutuksilla [1, p. 176].*

tutkimiin algoritmeihin verrattuna OddEven Sortin käyttämä aika oli useampia kertaluokkia suurempi syötekoosta riippumatta, ja testejä ei edes ajettu 10 000 000:n ja 100 000 000:n kokoisilla taulukoilla. Tulokset löytyvät kuvasta 4.4. Vaikka algoritmi itsessään oli hidas, niin rinnakkaislaskenta kuitenkin nopeutti sitä huomattavasti. Kahdeksalla ytimellä nopeutus oli yli 7-kertainen syötteen koon ollessa 100 000–1 000 000. Kuvassa 4.5 näkyy nopeutuksen suuruus. [2, p. 3]

Bozidarin ja Dobravecin toteutus Quicksortista ei saavuttanut suurta nopeutusta. Kaikilla syötetyypeillä nopeutus pysyi alle 20-kertaisena ja 64-bittisillä avain-arvo pareilla, nopeutus pysyi keskimäärin alle 10-kertaisena. Tulokset löytyvät kuvasta 4.1. [5, p. 6] Zecenan ja muiden toteutus Quicksortista hyötyi rinnakkaislaskennasta syötteen koon ollessa tarpeeksi suuri. 10 000:llä alkiolla Quicksort oli hitaampi kuin sekventiaalinen ver-

Sorting Algorithms Performance (1 machine, 1-8 cores)									
Data	Algorithm	Time (s) 1 Core	Total Energy (J)	Time (s) 2 Cores	Total Energy (J)	Time (s) 4 Cores	Total Energy (J)	Time (s) 8 Cores	Total Energy (J)
10,000	Odd-Even	1.99	268.6J	1.00	134.8J	0.53	71.3J	0.29	39.0J
	Shellsort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.01	1.3J
	Quicksort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.04	5.4J
100,000	Odd-Even	199.57	27176.6J	99.26	13804.8J	53.05	7473.6J	27.31	4021.8J
	Shellsort	0.23	30.9J	0.14	18.8J	0.09	12.1J	0.07	9.4J
	Quicksort	0.07	9.4J	0.04	5.4J	0.03	4.0J	0.06	8.1J
500,000	Odd-Even	4968.73	677369.7J	2500.20	345726.8J	1302.90	184657.0J	675.19	100622.9J
	Shellsort	1.67	269.2J	0.93	134.9J	0.55	74.0J	0.37	49.8J
	Quicksort	0.37	49.8J	0.19	25.5J	0.11	14.8J	0.09	12.1J
1,000,000	Odd-Even	20091.48	2780763.0J	9998.62	1382570.4J	5232.57	743229.9J	2716.37	406571.3J
	Shellsort	3.69	539.6J	2.06	270.8J	1.16	135.7J	0.76	102.2J
	Quicksort	0.76	102.2J	0.40	53.8J	0.23	30.9J	0.15	20.2J
10,000,000	Shellsort	62.61	8591.2J	32.95	4563.7J	18.12	2713.2J	10.03	1497.1J
	Quicksort	9.30	1358.3J	4.79	684.6J	2.68	422.2J	1.64	282.6J
100,000,000	Shellsort	1115.57	152286.0J	575.33	79914.9J	298.02	42670.3J	157.18	23524.7J
	Quicksort	167.76	23024.2J	86.00	12067.0J	45.54	6541.4J	25.82	3783.8J

Fig. 4. Performance of sorting algorithms.

Kuva 4.4. *Algoritmien energian- sekä ajankäyttö eri Zecenan ja muiden toteutuksilla [2, p. 4].*

Speedups (over serial)					
Data	Algorithm	Serial 1 Core	Parallel 2 Cores	4 Cores	8 Cores
10,000	Odd-Even	1x	1.98x	3.75x	6.79x
	Shellsort	1x	1.55x	2.00x	2.33x
	Quicksort	1x	0.00x	0.85x	0.17x
100,000	Odd-Even	1x	2.01x	3.76x	7.31x
	Shellsort	1x	1.65x	2.47x	3.52x
	Quicksort	1x	1.81x	2.48x	1.16x
500,000	Odd-Even	1x	1.99x	3.81x	7.36x
	Shellsort	1x	1.79x	3.03x	4.16x
	Quicksort	1x	1.89x	3.30x	4.02x
1,000,000	Odd-Even	1x	2.01x	3.84x	7.40x
	Shellsort	1x	1.79x	3.18x	4.81x
	Quicksort	1x	1.90x	3.34x	5.21x
10,000,000	Shellsort	1x	1.90x	3.45x	6.24x
	Quicksort	1x	1.94x	3.47x	5.67x
100,000,000	Shellsort	1x	1.94x	3.74x	7.10x
	Quicksort	1x	1.95x	3.68x	6.50x

Fig. 5. Speedup of sorting algorithms.

Kuva 4.5. *OddEven Sortin, Shellsortin ja Quicksortin nopeutuksien vertailu Zecenan ja muiden toteutuksilla [2, p. 5].*

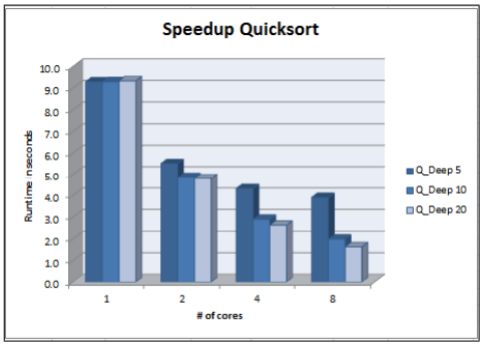


Fig. 9. Performance comparison of different Quicksort depths.

Kuva 4.6. *Rekursion syvyyden vaikutus ajankäyttöön Quicksortissa Zecenan ja muiden toteutuksella [2, p. 4].*

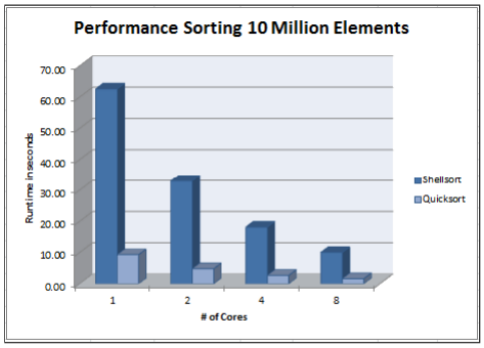
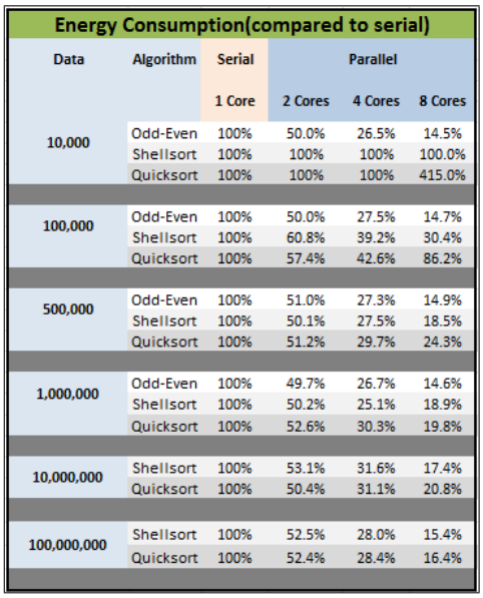


Fig. 7. Performance comparison of Shellsort and Qucksort.

Kuva 4.7. *Shellsortin ja Quicksortin ajankäytön vertailu 10 000 000:lla alkiolla Zecenan ja muiden toteutuksilla[2, p. 5].*

sio johtuen rinnakkaislaskennan rinnakkaistamisen ja synkronoinnin yleiskustannuksien vuoksi. Syötteen koon kasvaessa nopeutus kuitenkin kasvoi, mutta on huomattava, että 100 000:lla alkiolla neljän ytimen käyttö tuotti suuremman nopeutuksen kuin kahdeksan ydintä. Quicksortin nopeutus kuitenkin kasvoi syötteen koon kasvaessa. [2, p. 3] Testin tulokset löytyvät kuvasta 4.4 ja nopeutus kuvasta 4.5. Kasvattamalla rekursion syvyyttä Zecena ja muut testasivat Quicksortin rakeisuuden vaikutusta suorituskykyyn. Rekursion syvyyksinä käytettiin arvoja 5, 10, 15 ja 20. Testeistä huomattiin, että rekursion syvyyden kasvu nopeutti suoritusta. [2, p. 5] Tulokset rekursion syvyyden vaikutuksesta ajankäyttöön löytyvät kuvassa 4.6.

Zecena ja muiden toteutus Shellsortista tarvitsi myös tarpeeksi suuren syötekoon hyötyäkseen rinnakkaislaskennasta. Shellsort kuitenkin saavutti suurempia nopeutuksia tietyillä konfiguraatioilla kuin Quicksort kuten kuvasta 4.5 nähdään. Kuten Quicksort, Shellsortin nopeutus kasvoi syötteen koon kasvaessa. [2, p. 3] Shellsortin ja Quicksortin ajankäyttöä 10 000 000:lla alkiolla on vertailtu kuvassa 4.7.



Energy Consumption(compared to serial)					
Data	Algorithm	Serial	Parallel		
		1 Core	2 Cores	4 Cores	8 Cores
10,000	Odd-Even	100%	50.0%	26.5%	14.5%
	Shellsort	100%	100%	100%	100.0%
	Quicksort	100%	100%	100%	415.0%
100,000	Odd-Even	100%	50.0%	27.5%	14.7%
	Shellsort	100%	60.8%	39.2%	30.4%
	Quicksort	100%	57.4%	42.6%	86.2%
500,000	Odd-Even	100%	51.0%	27.3%	14.9%
	Shellsort	100%	50.1%	27.5%	18.5%
	Quicksort	100%	51.2%	29.7%	24.3%
1,000,000	Odd-Even	100%	49.7%	26.7%	14.6%
	Shellsort	100%	50.2%	25.1%	18.9%
	Quicksort	100%	52.6%	30.3%	19.8%
10,000,000	Shellsort	100%	53.1%	31.6%	17.4%
	Quicksort	100%	50.4%	31.1%	20.8%
100,000,000	Shellsort	100%	52.5%	28.0%	15.4%
	Quicksort	100%	52.4%	28.4%	16.4%

Fig. 6. Energy consumption of sorting algorithms.

Kuva 4.8. *Algoritmien energiankäyttö Zecenan ja muiden toteutuksilla [2, p. 4].*

4.3 Energiankäyttö

Ainoastaan Zecena ja muut vertasivat toteutuksiensa energiankäyttöä työssään. Kuten ajan suhteen, heidän toteutuksensa OddEven Sortista jätettiin lopullisesta vertailusta pois huomattavan tehottomuuden takia [2, p. 5]. Kuten kuvasta 4.4 näkyy, jo 100 000:lle alkiolla OddEven Sort vei tuhansia jouleja energiaa verrattuna Shellsortin kymmeniin ja Quicksortin alle kymmeneen jouleen.

Tuloksista huomataan, että algoritmien energiankäyttö laskee ytimien määrän kasvaessa ja ajankäytön laskiessa. Poikkeuksena on kuitenkin 10 000:n alkion testeillä Shellsortilla ja Quicksortilla, koska rinnakkaistamisen ja synkronoinnin yleiskustannukset ovat suuremmat kuin rinnakkaistuksesta saatu hyöty. 10 000:lla alkiolla käyttäen kahdeksaa ydintä Quicksort kulutti yli nelinkertaisen määrän energiasta verrattuna yhden ytimen käyttöön kuten kuvasta 4.8 huomataan. [2, p. 3]

Kuvasta 4.8 nähdään myös, että useammasta ytimestä saatu hyöty kasvaa syötteen koon kasvaessa. 100 000:lla alkiolla ja kahdeksalla ytimellä Shellsort ja Quicksort käyttävät vastaavasti 80,4 % ja 86,2 % siitä energiasta, jonka ne käyttivät yhdellä ytimellä taulukon järjestämiseen. Tämä suhde pienenee Shellsortilla 15,4 %:iin ja Quicksortilla 16,4 %:iin. [2, p. 3]

Lopuksi työssä verrataan Quicksortin ja Shellsortin tuloksia. Shellsort oli Quicksortia tehokkaampi 10 000:lla alkiolla, mutta Quicksort oli tehokkaampi jo 100 000:lla alkiolla ja siitä suuremmilla syötteillä. Quicksort on myös huomattavasti energiatehokkaampi kaikilla ydin määrillä. Shellsortin ja Quicksortin energiankäyttöä 10 000 000:lla alkiolla on vertailtu kuvassa 4.9. [2, p. 5]

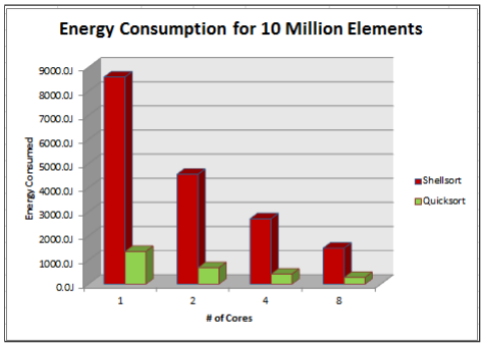


Fig. 8. Energy consumption of Shellsort and Quicksort.

Kuva 4.9. *Shellsortin ja Quicksortin energiankäytön vertailu 10 000 000:lla alkiolla Zecenan ja muiden toteutuksilla [2, p. 5].*

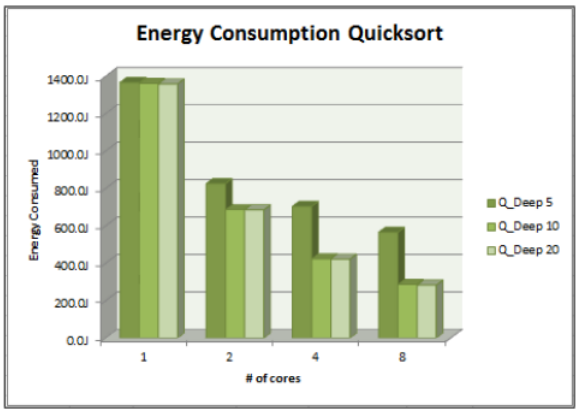


Fig. 10. Energy consumption of different Quicksort depths.

Kuva 4.10. *Quicksortin energiankäytön vertailu 10 000 000:lla alkiolla eri määrillä ytimiä ja eri rekursion syvyyksillä Zecenan ja muiden toteutuksella[2, p. 5].*

Lopulta Zecena ja muut vertailivat Quicksortin käyttäytymistä eri rakeisuuksilla vaihtamalla rekursion syvyyttä. 10 000 000 alkiota järjestettiin 5:n, 10:n, 15:n ja 20:n tason rekursiosyvyyksillä. Paras tulos saatiin syvyydellä 20, joka käytti energiaa noin 283,4 J:a verrattuna syvyydellä 5 kulutettuun 565,5 J:een. Tulokset löytyvät kuvasta 4.10. [2, p. 5]

5 YHTEENVETO

Tässä työssä käytiin läpi eri tutkimuksia rinnakkaislaskentaa hyödyntävistä algoritmeista ja niiden energiankäytöstä. Algoritmien ajankäyttö on pitkään tutkittu ja optimoitu asia, ja sekventiaalisen laskennan tehokkuuden kasvun hidastuessa on rinnakkaistaminen monesti paras tapa nopeuttaa laskentaa. Myös energiankulutus on nykypäivänä kriittinen asia niin tuotannollisesti kuin ekologisesti.

Työssä huomattiin, että monet algoritmit hyötyvät rinnakkaislaskennasta. Rinnakkaistetun version nopeutus riippuu kuitenkin siitä, miten hyvin laskentaa suorittava alusta tukee algoritmia, ja miten suuri syötteen koko on. Datariippumattomat algoritmit, kuten Bitonic Sort, hyötyivät näytönohjaimien arkkitehtuurista. Kuten Zecenan ja muiden Quicksort toteutuksesta huomattiin, syötteen koko on tärkeä kriteeri, kun päätetään sovelletaanko algoritmissa rinnakkaislaskentaa. Liian pieni syöte voi jopa hidastaa algoritmia, koska rinnakkaislaskennan yleiskustannukset vievät enemmän aikaa, kuin siitä saatu hyöty. On kuitenkin huomattava, että vaikka algoritmi nopeutuu huomattavasti, voi vähemmän nopeutuva algoritmi silti olla tehokkaampi.

Voidaan huomata, että moniytimisen prosessorin käyttäessä useampaa ydintä laskemiseen energiankäyttöä voi laskea jopa huomattavasti. Tämä johtuu siitä, että lisäytimien käyttämä energia on huomattavan pieni osa tietokoneen kokonaiskulutusta. Jatkokysymyksenä työlle on kuitenkin, että kuinka paljon näytönohjainten käyttö nostaa tai laskee energiankäyttöä, koska yksikään työ ei vertaillut sitä.

LÄHTEET

[1] Khan, F. G., Khan, O. U., Montrucchio, B. ja Giaccone, P. Analysis of Fast Parallel Sorting Algorithms for GPU Architectures'. *2011 Frontiers of Information Technology*. IEEE. 2011, 173–178.

[2] Zecena, I., Zong, Z., Ge, R., Jin, T., Chen, Z. ja Qiu, M. Energy consumption analysis of parallel sorting algorithms running on multicore systems. *2012 International Green Computing Conference (IGCC)*. IEEE. 2012, 1–6.

[3] Dongarra, J. ja Lastovetsky, A. L. *High performance heterogeneous computing*. Vol. 78. John Wiley & Sons, 2009.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L. ja Stein, C. *Introduction to algorithms*. MIT press, 2009.

[5] Bozidar, D. ja Dobravec, T. Comparison of parallel sorting algorithms. *arXiv preprint arXiv:1511.03404* (2015).