Tampereen yliopisto

Viljami Irri

RINNAKKAISUUDEN VERTAILU: C++ JA PYTHON

Toteutuksen ja nopeutuksen tarkastelu

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Maarit Harsu
Huhtikuu 2023

TIIVISTELMÄ

Viljami Irri: Rinnakkaisuuden vertailu: C++ ja Python
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Huhtikuu 2023

Rinnakkaista suoritusta esiintyy lähes kaikkialla nykypäivänä, esimerkiksi tieteen ja teollisuuden aloilla, pelialalla ja sisällöntuotannossa. Rinnakkaisissa sovelluksissa pyritään tavallisesti mahdollisimman korkeaan suorituskykyyn, mikä auttaa alati kasvavien datamäärien käsittelynopeudessa. Rinnakkaisuuden merkityksellisyys on myös nähtävissä suoritinmarkkinoilla - suorittimien ydinmäärät ovat nousseet erityisesti viime vuosina.

Rinnakkaisuuden toteutustavat riippuvat käytetystä ohjelmointikielestä. Tässä työssä tarkastellaan Python- ja C++-kieliä. Python-kieli on tarkoitettu aloittelijaystävälliseksi ja helposti luettavaksi skriptikieleksi, joka hyödyntää monenlaisia kirjastoja joustavuuden takaamiseksi. Python-ohjelmat ajetaan virtuaaliympäristössä, joka tekee niistä alustariippumattomia. C++-kieli on tunnetusti hankala ohjelmointikieli, joka antaa ohjelmoijalle paljon vapautta matalan tason ohjelmointiin liittyen ja on suunniteltu suorituskyky- ja tehokkuuspainotteiseksi. Työssä vertaillaan näiden kahden ohjelmointikielen rinnakkaisia ohjelmatoteutuksia. Työn tarkoituksena on selvittää kielten rinnakkaisuuden toteutustapoja ja tehokkuutta.

Tutkielmassa tutustutaan rinnakkaisuuden teoreettiseen taustaan keskittyen rinnakkaisuuden toteutustapoihin, haasteisiin, tehokkuuteen vaikuttaviin asioihin sekä suorituskyvyn erilaisiin näkökulmiin. Työssä tarkasteltavia ohjelmointikieliä vertailtiin kirjoittamalla vastaavat ohjelmatoteutukset ja vertailemalla ohjelmatoteutuksien suorituskykyä keskittyen erityisesti rinnakkaisuudesta saatuun nopeutukseen. Ohjelmatoteutuksien rinnakkainen osuus oli matriisin kertolaskualgoritmi. Valitut eri sarakemitan matriisit ajettiin ohjelmatoteutuksilla, ja mittaustuloksia valikoitiin Best-K-metodin avulla. Tulokset herättivät pohdintaa muun muassa prosessien ja säikeiden yleiskustannusten erosta ja mahdollisesta jatkotutkimuksesta.

Mittausten perusteella tehokkuuden ero on merkittävä tarkasteltavien ohjelmointikielten välillä. Eroon vaikuttaa eniten Pythonin tulkkaus, joka hidastaa toteutetun ohjelman suorituskykyä verrattuna C++:n vastaavaan ohjelmatoteutukseen, joka on käännetty ennen ohjelman ajoa. Kielten skaalautuvuus on hyvin samankaltainen rinnakkaisessa kontekstissa. Tehokkuuseroa voidaan pienentää Pythonissa sovelluskehysten avulla.

Avainsanat: Rinnakkaisuus, parallelismi, matriisilaskenta, C++, Python, säie

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

GIL	engl. Global Interpreter Lock, CPythonin toteutuksen osa, joka rajoittaa säikeiden toimintaa ja varmistaa säieturvallisuuden (engl. thread safety)
SMT	engl. Simultaneous multithreading, usean säikeen suorittaminen laitteistotuella

1. JOHDANTO

Rinnakkainen suoritus on nykypäivänä suosittua etenkin tieteellisessä laskennassa sekä pelikehityksessä. Rinnakkaisuuden suosio näkyy myös nykypäivän suoritinmarkkinoilla – viime vuosina suorittimien ydinten määrä on kasvanut reilusti. Rinnakkainen suoritus parantaa merkittävästi ohjelman suorituskykyä, mutta sen toteuttaminen on erittäin virhealtista. Tästä syystä jo ohjelman suunnitteluvaiheessa täytyy ottaa rinnakkaisuuden tuomat haasteet huomioon.

Rinnakkaista suoritusta on olemassa kahdenlaista: asynkronista rinnakkaisuutta (engl. concurrency) sekä samanaikaista suoritusta (engl. parallelism). Asynkronisen rinnakkaisuuden suoritus ei välttämättä tapahdu samanaikaisesti muun ohjelman rinnalla, vaan peräkkäisesti. Tässä työssä rinnakkaisella suorituksella tarkoitetaan samanaikaista suoritusta eli parallelismia ja tätä käsitellään asynkronisen rinnakkaisuuden sijasta.

Tässä työssä tutkitaan C++- ja Python-kielten rinnakkaisen ohjelmatoteutuksen eroja tehokkuuden näkökulmasta. Työtä varten on suoritettu empiirinen tutkimus matriisilaskennan avulla. Tutkimus keskittyy ohjelmien toteutukseen sekä niiden rinnakkaisuuden nopeutukseen. Työssä tuodaan esille ohjelman tehokkuuteen vaikuttavia asioita, kuten kielten rinnakkaisuuden toteutustavat ja käytettävän prosessorin spesifikaatio. Työn tavoitteena on selvittää kielten rinnakkaisuuden toteutustapoja ja tehokkuutta.

Tutkimuksessa selvisi, että molemmat kielet skaalautuvat samankaltaisesti rinnakkaisessa kontekstissa. Python-kieli on tosin laskennalliselta tehokkuudeltaan heikompi kuin C++-kieli. Pythonille on tarjolla sovelluskehyksiä, joiden avulla tehokkuutta voidaan parantaa merkittävästi.

Työssä käydään ensin läpi tarvittava rinnakkaisuuden teoria. Työtä varten tehdyt ohjelmatoteutukset esitellään ja niiden toteutukseen perehdytään tarkemmin. Lisäksi käytetyt teknologiat, mittausmenetelmät ja -laitteisto esitellään. Mittaustuloksia analysoidaan ja verrataan keskenään. Tulosten pohjalta ehdotetaan vaihtoehtoja rinnakkaisuuden toteutukseen sekä esitetään mahdollisia jatkotutkimuksen aiheita.

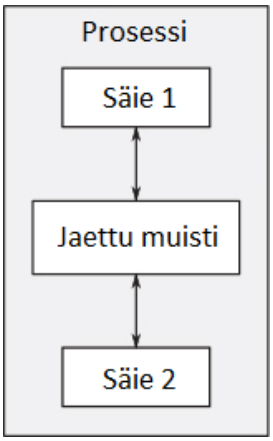
2. RINNAKKAISUUDEN TAUSTAA

Rinnakkaisuus tarkoittaa ohjelmointitapaa, jossa käytetään erilaisia tekniikoita ohjelman eri osien samanaikaiseen suoritukseen. Rinnakkaisuutta voidaan toteuttaa eri tavoilla eri ohjelmointikielissä. Tässä luvussa käydään läpi rinnakkaisuuden teoriaa, huomioitavia haasteita sekä rinnakkaisen ohjelman tehokkuuteen vaikuttavia asioita.

2.1 Prosessit ja säikeet

Rinnakkainen suoritus voidaan toteuttaa prosessien tai säikeiden avulla. Prosessi on yksinkertaisuudessaan ohjelman ilmentymä, jota ajetaan tietokoneessa. Jokaisella prosessilla on oma muistialueensa. Prosessien käyttöön liittyy luontaista yleiskustannusta esimerkiksi prosessien käynnistämisen ja käyttöjärjestelmän sisäisten resurssien jakamisen yhteydessä (Williams, 2019). Prosessien välinen kommunikointi tapahtuu esimerkiksi signaalien, sokettien tai tiedostojen avulla, mikä on usein hidasta, koska käyttöjärjestelmät suojaavat prosesseja välttääkseen toisen prosessin datan muokkauksen toisesta prosessista (Williams, 2019).

Säie (engl. thread) on ohjelmaa suorittava olio, joka on osa prosessia. Prosessilla voi olla monta säiettä, mutta niitä täytyy olla vähintään yksi kappale. Koska säikeet toimivat samassa prosessissa, ne jakavat saman muistialueen (Williams, 2019). Säikeillä on siis pääsy samoihin resursseihin keskenään. Säikeiden käynnistämiseen ja niiden väliseen kommunikointiin liittyvä matala yleiskustannus verrattuna prosessien vastaaviin yleiskustannuksen lähteisiin on tehnyt säikeistä suositumman lähestymistavan rinnakkaisuutta hyödyntävissä sovelluksissa esimerkiksi C++-kielessä, vaikka säikeiden jaettu muistialue voi aiheuttaa vaikeuksia (Williams, 2019). Kuvasta 1 nähdään, miten säikeiden välinen kommunikaatio tapahtuu.



Kuva 1. *Säikeiden kommunikaatio prosessissa (Williams, 2019).*

Rinnakkaisuudesta saatava hyöty ei välttämättä ole odotuksien mukainen, sillä säikeiden käynnistämiseen liittyy luontaista yleiskustannusta. Käyttöjärjestelmän täytyy allokoida resursseja ja lisätä uusi säie vuorottajaan. Mikäli säikeen suorittama tehtävä valmistuu nopeasti, tehtävän suorittamiseen kulunut aika voi olla pienempi kuin säikeen käynnistykseen liittyvä yleiskustannus. (Williams, 2019) Välillä säikeen luomatta jättäminen voi siis olla hyödyllistä ohjelman suorituskyvyn kannalta.

2.2 Rinnakkaisuudessa huomioitavat haasteet

Jaettujen resurssien kanssa täytyy huomioida tilanteet, joissa monet säikeet yrittävät tehdä samoihin resursseihin muokkauksia samanaikaisesti. Näissä tilanteissa ohjelman toiminta saattaa olla virheellistä, mikäli tarpeellisia varotoimenpiteitä ei noudateta. Tätä voidaan havainnollistaa seuraavalla esimerkillä.

Kuvitellaan tilanne, jossa kaksi säiettä yrittävät muokata samaa kokonaislukua samanaikaisesti nostaen tämän arvoa 1:llä. Molemmat säikeet lukevat kokonaisluvun nykyisen arvon, 2. Ensimmäinen säie nostaa arvoa 1:llä ja sijoittaa kokonaisluvun arvoksi 3. Toinen säie luki kokonaisluvun arvon ennen ensimmäisen säikeen toimintaa. Tästä syystä toisen säikeen summaus tuottaa myös kokonaisluvun arvoksi 3. Molemmat säikeet nostivat arvoa 1:llä, mutta koska molemmat säikeet lukivat kokonaisluvun arvon ennen sen muokkausta, ohjelma tuotti virheellisen tuloksen.

Kuvailtu tilanne on esimerkki kilpailutilanteesta, joka tapahtuu ainoastaan jaettua resurssia muokatessa. Kilpailutilanteet voidaan välttää rajoittamalla säikeiden pääsyä jaettuun resurssiin esimerkiksi lukon avulla. Lukko ympäröi koodilohkoa ja rajoittaa säikeiden pääsyä siihen niin, että vain yhdellä säikeellä on pääsy koodilohkoon kerrallaan (Williams, 2019). Lukittua aluetta koodissa kutsutaan kriittiseksi lohkoksi. Lukkojen käyttö

aiheuttaa muun muassa välimuistihuteja (engl. cache miss) (Dud's & S'ndor, 2012). Tämä rajoittaminen tuo esiin uusia haasteita, kuten säikeiden nälkiintymisen.

Nälkiintyminen on tila, jossa säie ei pysty jatkamaan suorittamista muiden säikeiden toiminnan vuoksi. Jos toinen säie suorittaa operaatioita ”väärällä” ajoituksella, toinen säie saattaa edetä suorituksessaan, kun taas ensimmäinen säie joutuu jatkuvasti suorittamaan operaationsa uudelleen (Williams, 2019). Muut säikeet voivat esimerkiksi ottaa lukon haltuunsa ennen kuin jo odottava säie ehtii ottaa sen itselleen haltuun. Tästä syystä säie joutuu odottamaan edelleen heikentäen ohjelman suorituskykyä, koska säie ei suorita ohjelmaa odottaessaan.

Kilpailutilanteita voidaan myös välttää, jos suoritettavat operaatiot ovat atomisia. Atomiset operaatiot ovat välittömästi suoritettavia, keskeyttämättömiä operaatioita (Williams, 2019). Säikeen suorittaessa atomista operaatiota muut säikeet eivät voi nähdä operaatiota keskeneräisessä tilassa (Williams, 2019). Atomisten operaatioiden avulla voidaan eliminoida lukoista aiheutuvaa yleiskustannusta sekä niistä käytävää kilpailua (Ichnowski & Alterovitz, 2014). Kuitenkin atomiset operaatiot lukottomassa koodissa voivat olla paljon hitaampia kuin ei-atomiset operaatiot lukollisessa ratkaisussa, ja todennäköisesti niitä tarvitaan enemmän samaa toiminnallisuutta varten (Williams, 2019). Atomisten operaatioiden käytöllä ei siis aina voida parantaa ohjelman suorituskykyä. Suorituskykyä voidaan kuitenkin arvioida esimerkiksi huonoimmalla odotusajalla, keskiarvoisella odotusajalla tai ohjelman suoritusajalla (Williams, 2019). Tämän työn ohjelmatoteutuksissa operaatiot eivät ole atomisia, mutta niissä on onnistuttu välttämään jaettujen resurssien yllä mainittuja ongelmia jakamalla säikeille omat vastuualueensa suoritettavasta työstä.

2.3 Tehokkuuteen vaikuttavat asiat

Rinnakkaisessa kontekstissa ohjelmien tehokkuuteen on hyvin monta vaikuttavaa asiaa. Tässä luvussa esitellään tämän työn kannalta oleellisimmat aiheet.

2.3.1 Kääntäminen vs. tulkkaus

C++-ohjelmointikieli on käännetty kieli tarkoittaen sitä, että C++-koodi käännetään konekoodiksi ennen kuin ohjelman suoritus tapahtuu. Python-kieli on vastaavasti tulkattu kieli, eli kirjoitettua koodia tulkitaan ajon aikana suorituksen etenemisen myötä. Tämä on merkittävä tekijä ohjelmointikielien tehokkuuden välillä, sillä tulkkauksesta aiheutuu yleiskustannusta koko ohjelman ajon ajan. Tulkkauksen vaikutus suorituskykyyn näkyy sekä sarjallisissa että rinnakkaisissa toteutuksissa.

Python-kielen rinnakkaisuus on toteutettu tulkkauksen asettamien rajoitteiden mukaisesti. Tulkkiin liittyvä GIL-mekanismi (engl. Global Interpreter Lock) estää säikeiden rinnakkaisen suorituksen, koska Python-tavukoodia voidaan ajaa vain yhdellä säikeellä kerrallaan (Python, 2020). Pythonissa säikeet ovat siis lähinnä hyödyllisiä asynkronisen suorituksen sovelluksissa – ei rinnakkaislaskennassa. Esimerkiksi I/O-operaatiot ja kuvankäsittely tapahtuvat GIL-mekanismin ulkopuolella, jolloin siitä ei aiheudu pullonkaulaa, mutta monisäikeiset ohjelmat käyttävät mekanismia Python-tavukoodin tulkkaamiseen (Python, 2020). Tämä rajoite voidaan välttää käyttämällä prosesseja säikeiden sijasta. On olemassa Python-toteutuksia, joissa GIL ei ole käytössä, esimerkiksi Jython ja IronPython (Python, 2020). Nämä toteutukset jätetään kuitenkin tämän työn ulkopuolelle. C++-kielessä säikeet voivat ajaa koodia samanaikaisesti hyödyntäen yhteistä muistialuetta, mikä parantaa Pythonin toteutukseen nähden suorituskykyä. Pythonilla kulunut aika funktion kutsumiseen kahdelta säikeeltä voi olla kaksinkertainen verrattuna kuluneeseen aikaan, kun funktiota kutsutaan kahdesti yhdellä säikeellä (Python, 2020).

2.3.2 Suoritin

Suoritin eli prosessori on tietokoneen osa, joka suorittaa ohjelmakoodissa esiintyviä käskyjä. Prosessorin rakenne ja ominaisuudet vaikuttavat merkittävästi varsinkin rinnakkaisten ohjelmien suorituskykyyn. Tärkein prosessorin ominaisuus rinnakkaisen suorituskyvyn kannalta on sen ydinten määrä. Ydinten määrä vaikuttaa siihen, kuinka monta konekäskyä prosessori voi suorittaa samanaikaisesti. Modernin prosessorin yksittäinen ydin pystyy usein ajamaan kahta säiettä kerrallaan SMT-tekniikan (engl. Simultaneous multithreading) avulla. SMT-tekniikka on laitteistopohjainen ratkaisu monisäikeistykselle. SMT-tekniikka parantaa prosessorin laskennallista nopeutta ja suoritustehoa. Prosessorin ydinten määrä on siis yksi merkittävimmistä rajoittavista tekijöistä rinnakkaisuudesta saatavaan hyötyyn.

Suorittimen kellotaajuus kertoo, kuinka monta tilanvaihdosta suoritin kykenee tekemään sekunnissa. Mitä korkeampi taajuus on, sitä enemmän käskyjä voidaan suorittaa tietyssä ajassa. Kellotaajuus vaikuttaa merkittävästi varsinkin sarjallisen ohjelman tehokkuuteen, sillä käytössä on vain yksi ydin kerrallaan, jolloin yksittäisen ytimen suorituskyky korostuu.

Suorittimella on välimuistia, joka on tietokoneen keskusmuistia moninkertaisesti nopeampaa. Välimuistilla on yleensä monia eri tasoja, esimerkiksi L1-, L2- ja L3-muisti. Moderneilla laitteilla nopeimman välimuistin, L1, muistihuti kustantaa noin 10 kellosykliä ja

tämä kustannus kasvaa hitaammilla välimuisteilla (Valgrind, 2022). Tässä työssä toteutettujen ohjelmien välimuistin käyttöä ei ole analysoitu ja tästä syystä välimuistin vaikutukset jätetään tämän työn ulkopuolelle.

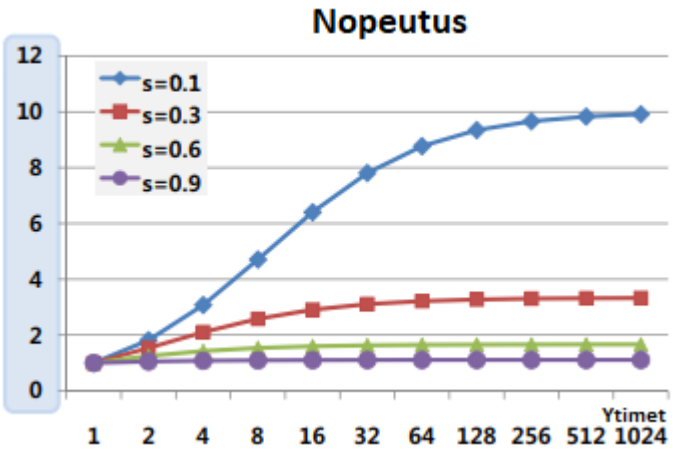
2.4 Rinnakkaisen suorituskyvyn suureet

Tässä työssä rinnakkaisuusasteella tarkoitetaan ohjelman tai algoritmin osaa, joka on rinnakkaistettavissa. Rinnakkaisen ohjelman koko koodia ei yleensä ole tapana ajaa rinnakkaisesti, vaan ainoastaan osat, joissa rinnakkaisten tekniikoiden käyttöä voidaan perustella tehokkuuden näkökulmasta. On mahdollista, että jotakin ohjelman koodin osaa ei voida suorittaa rinnakkaisesti, mikä vähentää rinnakkaisuusastetta.

Suoritusaika on ohjelman konkreettinen suorituskyvyn mittari. Mitä nopeammin ohjelma kykenee tuottamaan halutun lopputuloksen, sitä tehokkaampi se on. Suoritusajassa nähdään merkittäviä muutoksia sarjallisten ja rinnakkaisten ohjelmien välillä varsinkin, kun ohjelman rinnakkaisuusaste on korkea. Parhaimmillaan rinnakkaisesta toteutuksesta saatava nopeutus on superlineaarinen. Nopeutus on superlineaarista, jos ohjelman suorituskyky on parempi kuin lineaarisen nopeutuksen raja (Gusev & Ristov, 2014). Käytännössä saavutetun nopeutuksen täytyy olla moninkertaisempi kuin ohjelmaa ajavien suorittajien määrä. Esimerkiksi matriisin kertolaskulle superlineaarinen nopeutus on mahdollinen saavuttaa jaetun muistin moniprosessorilla (engl. shared memory multiprocessor) (Gusev & Ristov, 2014).

Virrankulutus on varsinkin nykypäivänä erittäin tärkeä aspekti, joka täytyy ottaa huomioon ohjelmaa suunniteltaessa ja sen ajavan laitteiston määrittelemisessä. Rajoitetusti skaalautuvat sovellukset eivät välttämättä parane suorituskyvyltään, vaikka ajavien ydinten määrää lisättäisiin (Conoci et al., 2021). Ydinten lisääminen saattaa kuitenkin johtaa heikompaan suorituskykyyn, jonka seurauksena energiatehokkuus heikkenee korkeamman virrankulutuksen takia (Conoci et al., 2021). Tätä haastetta ei käsitellä tämän työn mittauksissa, sillä toteutetut ohjelmat ovat pienikokoisia. Virrankulutus olisi kuitenkin hyvä ottaa huomioon suuremman skaalan ohjelmissa varsinkin, jos ohjelmaa ajetaan hyvin paljon.

Nopeutus on myös tärkeä mittari ohjelman suorituskyvyn kannalta. Suorittajien lisääminen ohjelmaan usein parantaa suorituskykyä, mutta tämä hyöty pienenee suorittajaa kohti niitä lisätessä. Tätä ilmiötä kutsutaan vähenevän tuoton laiksi ja sitä voidaan havainnollistaa kuvan 2 avulla.



Kuva 2. *Nopeutus ydintä kohti (Lee et al., 2009).*

Kuvassa 2 on tutkittu Amdahlin lain avulla teoreettista nopeutusta, kun *s* on sarjallisen koodin osuus ajetun ohjelman koodista. Amdahlin laki on malli rinnakkaisen ja peräkkäisen ohjelman nopeuseron arviointiin ja ennustamiseen. Tätä käsitellään tarkemmin luvussa 3.3. Kuvasta 2 huomataan, että nopeutus ydintä kohti pienenee ydinten kokonaismäärän kasvaessa. Jossakin vaiheessa nopeutus on niin pientä ydintä kohti, että ydinten lisääminen ei ole enää perusteltua tehokkuuden kannalta. Jokaisen ohjelman kohdalla tämä olisi hyvä ottaa huomioon. Tämän työn mittauksissa tarkastellaan suorittajien antamaa nopeutusta.

3. MITTAUSJÄRJESTELYT

Tässä luvussa käydään läpi mittauksissa käytetty laitteisto ja ohjelmien toiminnan kannalta oleellisia koodipätkiä, optimointeja ja teknologioita. Lopuksi kerrotaan käytetyistä mittausmenetelmistä.

3.1 Laitteisto

Työssä käytettävän tietokoneen käyttöjärjestelmä on Windows 10. Tietokoneelle on asennettu Pythonin versio 3.7.9 ja käytössä on tavallinen CPython-tulkki. Tietokoneen C++-kääntäjäksi on valittu 64-bittinen MinGW-kääntäjä, jonka versio on 8.1.0.

Tietokoneen prosessori on malliltaan AMD Ryzen 5 1600. Tämän kuusiytimisen prosessorin kellotaajuus on 3,20 GHz ja säikeiden määrä on 12 (AMD, 2023). Mikäli ydinten määrä tai kellotaajuus olisi suurempi, ohjelmatoteutuksilla saataisiin parempia tuloksia kuin tässä työssä. Prosessorin ydinten määrä rajoittaa ohjelmissa luotujen säikeiden antamaa hyötyä, kun ohjelmassa käytetään tätä määrää enemmän säikeitä.

3.2 Ohjelmatoteutukset

Tämän työn ohjelmatoteutukset käyttävät algoritminaan matriisien kertolaskua. Algoritmin kaava voidaan ilmaista muodossa

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj},$$

jossa *c* on tulosmatriisi, *a* ja *b* ovat tulon matriisit, *i* on rivin numero, *j* on sarakkeen numero, *k* on näitä indeksoiva muuttuja ja *m* on muuttujan *k* maksimiarvo eli matriisin rivitai sarakemitta. Tämän algoritmin valinta perustuu sen tärkeään rooliin lineaarialgebrassa sekä sen rinnakkaiseen soveltuvuuteen (Liu et al., 2012). Kyseisen algoritmin toteutus on myös samankaltainen tässä työssä käsiteltävien ohjelmointikielien välillä.

Ohjelmissa 1 ja 2 on esitelty ohjelmatoteutuksien käyttämät matriisin kertolaskualgoritmit C++-kielellä. Molempien kielien ohjelmatoteutuksien kokonaisuuksia voi tarkastella syvemmin GitHubista (Irri, 2023).

```
void sequential_multiplication(){
    for(int i=0;i<SIZE;i++){
            int i_offset = SIZE*i;
            for(int j=0;j<SIZE;j++){
                int j_offset = SIZE*j;
                int total = 0;
                for(int k=0;k<SIZE;k++){
                    total += flat_a[i_offset+k] * flat_b[j_offset+k];
                }
                matrix_result[i_offset+j] = total;
            }
        }
}

```

Ohjelma 1. *Sarjallinen matriisin kertolasku.*

Ohjelmissa käytettävät matriisit ovat määritelty globaaleiksi muuttujiksi. Koska ohjelmien matriisit ovat muokattu yksiulotteisiksi listoiksi niiden indeksoimiseen tarvitaan kokonaislukumuuttujia *i_offset* ja *j_offset*. Kokonaislukumuuttuja *total* toimii tulosmatriisin yhden alkion tuloksena. Tähän muuttujaan summataan kerrottavien matriisien alkioiden kertolaskuja. Lopulta muuttujan arvo sijoitetaan tulosmatriisiin.

```
void parallel_multiplication(int dimension_start, int dimen-
sion_stop){
    for(int i=dimension_start;i<dimension_stop;i++){
        int i_offset = SIZE*i;
        for(int j=0;j<SIZE;j++){
            int j_offset = SIZE*j;
            int total = 0;
            for(int k=0;k<SIZE;k++){
                total += flat_a[i_offset+k] * flat_b[j_offset+k];
            }
            matrix_result[i_offset+j] = total;
        }
    }
}
```

Ohjelma 2. *Rinnakkainen matriisin kertolasku.*

Huomataan, että algoritmit ovat samanlaiset, mutta rinnakkaisessa algoritmissa algoritmin suorittajalle täytyy määrittää parametreina käsiteltävät matriisien ulottuvuudet. Annettuja parametreja hyödynnetään matriisien indeksoinnissa. Tässä työssä siis säikeille on jaettu matriiseista tietyt osat, jotta voidaan välttää luvussa 2.2 esiteltyjä haasteita, esimerkiksi kilpailutilanteita. Jaetut matriisien osat on pyritty pitämään mahdollisimman samankokoisina suorittajien välillä, jotta ohjelman rinnakkaisuusaste olisi mahdollisimman korkea.

Tämän työn ohjelmissa matriisit ovat aina neliön mallisia yksinkertaisuuden vuoksi. Laskentaa on optimoitu muokkaamalla matriisit yhteen ulottuvuuteen sopiviksi, sekä toinen matriiseista sarakepainotteiseksi (engl. column-major). Näiden optimointien avulla matriiseista etsitty lukuarvo löytyy yksittäisen indeksin avulla ja indeksointia varten tarvitsee suorittaa vähemmän laskentaa.

C++-ohjelmatoteutuksessa rinnakkaisuus on toteutettu standardikirjaston std::thread-luokalla, joka mahdollistaa usean funktion samanaikaisen suorituksen (Cppreference, 2022). Kyseisen luokan oliot vastaavat toiminnaltaan säikeitä. Aiemmin mainitun GIL-mekanismin takia Pythonin ohjelmatoteutuksessa on käytetty multiprocessing-kirjaston Process-luokkaa, joka kykenee samanaikaiseen suoritukseen kuten std::thread-luokka. Process-luokan oliot vastaavat toiminnaltaan prosesseja.

3.3 Mittausmenetelmät

Suoritusajan mittaus toteutetaan ohjelmissa itsessään. Sarjallisissa toteutuksissa suoritusaika sisältää vain matriisilaskentaan kuluneen ajan. Rinnakkaisissa toteutuksissa myös säikeiden tai prosessien aiheuttama yleiskustannus huomioidaan suoritusajassa. Kun matriisien laskuoperaatio on suoritettu, ajan mittaus lopetetaan.

C++-toteutuksessa suoritusajan mittaaminen tapahtuu std::chrono-kirjaston high_resolution_clock-luokan avulla. Matriisioperaatiota ennen ja sen jälkeen otetaan ylös ajankohdat, joiden erotuksena saadaan matriisioperaatioon kulunut aika. Python-toteutuksessa suoritusajan mittaus tapahtuu samankaltaisesti time-moduulin avulla.

Tulosten analysoinnissa hyödynnetään Best-K-metodia. Metodissa valitaan K määrä luotettavimpia tuloksia kaikista tuloksista (Yao et al., 2019). Tässä työssä ohjelman suoritusten joukosta valitaan neljä nopeinta aikaa, koska ohjelman kokemat hidastukset ovat aina ulkoisten tekijöiden aiheuttamia, joten ohjelman todellinen suorituskyky on lähimpänä parhaita tuloksia. Näiden parhaiden tulosten keskinäisen eron maksimiarvoksi on valittu alle puoli sekuntia, jotta tuloksia voidaan pitää luotettavana. Tästä syystä joitakin ohjelman suorituskertoja, jotka ovat olleet epätavallisen nopeita muihin suorituskertoihin verrattuna on jätetty huomioimatta. Epätavallisen nopeat tulokset ovat mahdollisia, jos käyttöjärjestelmän tai jonkin muun taustaprosessin toiminnan aiheuttama kuormitus on hetkellisesti matalampaa kuin muilla mittauskerroilla. On kuitenkin mahdollista, että nämä epätavallisen nopeat tulokset toistuisivat, mikäli mittauskertoja lisättäisiin. Tässä työssä kaikilla säie- tai prosessimäärillä on suoritettu vähintään 20 mittausta kummallakin kielellä ja tarvittaessa mittauskertoja on lisätty yksittäisissä tapauksissa, mikäli Best-K-metodin asettamiin rajoihin ei olla päästy.

Vertailussa olevien ohjelmointikielien laskennallisen tehokkuuden eron vuoksi valitut matriisien koot ovat erilaiset. Python-kielen tapauksessa matriisien rivi- ja sarakemitaksi valittiin 800 ja C++-kielen tapauksessa 5500. Ohjelmien suoritusajat yksittäisen suorittajan tapauksessa valituilla mitoilla ovat alle puolen sekunnin välillä toisistaan, joten vertailu on mielekästä. Esimerkiksi C++-kielellä ja matriisin mitalla 800 suoritettavaan matriisilaskuun kului noin kolmasosa sekuntia yhdeltä suorittajalta, jolloin eri säiemäärillä suoritettavia mittauksia ei voida tehdä mielekkäästi.

Tuloksia voidaan myös verrata Amdahlin lain antamaan ennusteeseen nopeutuksen osalta. Tulosta voidaan käyttää suuntaa antavana ennusteena, mutta tähän on kuitenkin syytä suhtautua varauksella. Tätä mallia on kritisoitu vuosien varrella eri seikoista. Mallin paikkansapitävyyttä on kyseenalaistettu (Dévai, 2017). Myös mallin huomioimia parametreja on kritisoitu (Zheng et al., 2015). Nopeutus voidaan Amdahlin lain avulla laskea kaavalla

$$\text{Nopeutus} = \frac{1}{(1-p) + p/N},$$

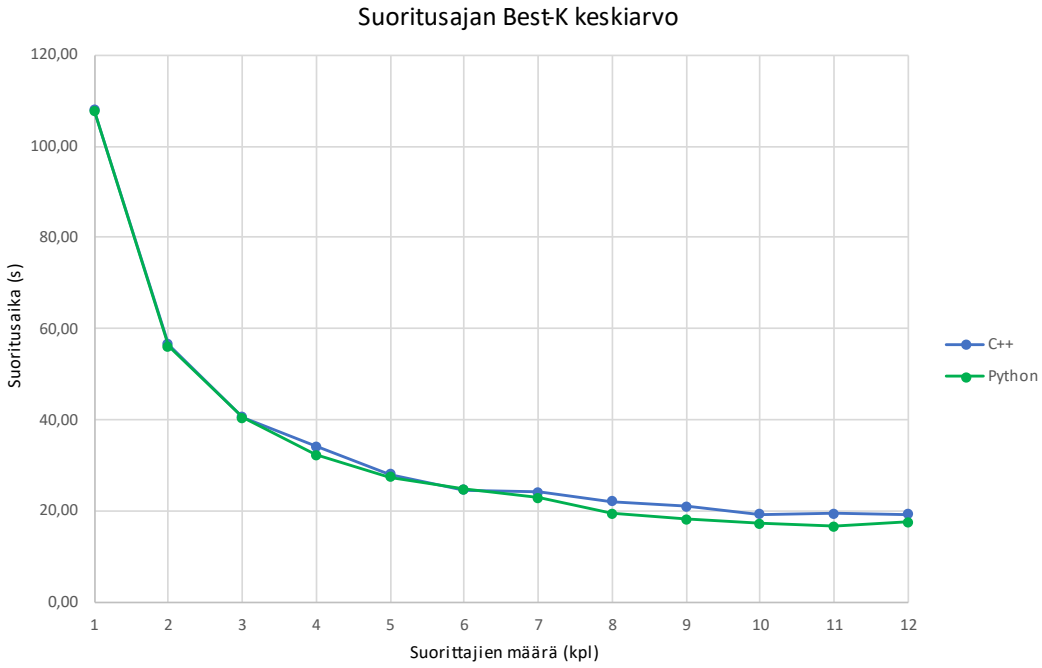
jossa *p* on rinnakkaistettavan koodin osuus ja *N* on koodia suorittavien ydinten määrä. Tässä työssä rinnakkaistettavan koodin osuus vaihtelee sen mukaan, kuinka nopeasti säikeet tai prosessit käynnistyvät. Osuus pienenee, jos jokin suorittajista on muita jäljessä. Koodin rinnakkaisesti suoritettavaksi osuudeksi voidaan valita tässä 0,95. Tämä valinta perustuu toteutuneisiin mittaustuloksiin. Kuten edellä mainittiin, työssä käytettävällä prosessorilla on kuusi ydintä, joten tätä ydinmäärää seuraavat ennusteet eivät päde tältä osin.

4. TULOSTEN ANALYSOINTI

Tässä luvussa esitellään mittauksissa saadut tulokset ja niitä analysoidaan suoritusajan, saavutetun nopeutuksen ja suoritusajan keskihajonnan kannoilta. Ohjelmointikielten tuloksia vertaillaan keskenään ja tulosten luotettavuutta arvioidaan. Lopuksi tulosten pohjalta ehdotetaan lähteisiin ja pohdintoihin perustuen vaihtoehtoja rinnakkaisuuden toteuttamiseen.

4.1 Tulosten vertailu

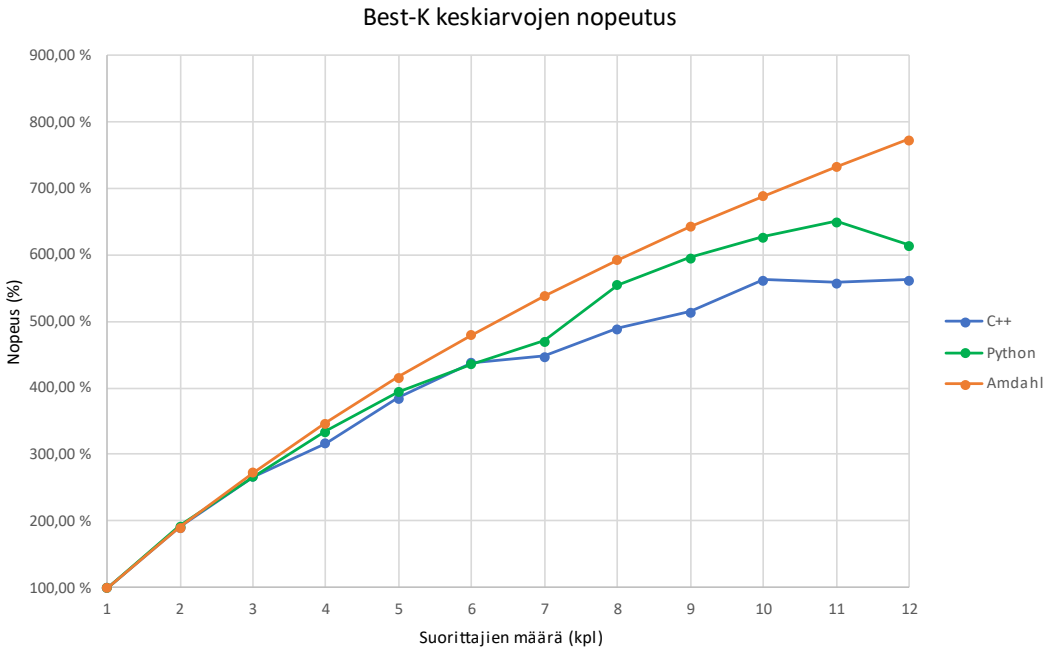
Mittauksissa toteutuneista suoritusajoista on otettu neljän parhaan tuloksen keskiarvo säie- ja prosessimääräkohtaisesti. Nämä tulokset ovat esillä kuvassa 3.



Kuva 3. *Suoritusaikojen Best-K keskiarvo.*

Kuvan 3 kuvaajasta voidaan tulkita, että molemmat kielet nopeutuvat hyvin samankaltaisesti suorittajien määrää lisättäessä. Vastoin odotuksia Python-kieli näyttää suoriutuvan hieman nopeammin kuin C++-kieli varsinkin, kun suorittajien määrä on korkeampi kuin kuusi. Tämä nähtävissä oleva ero johtunee C++-kielen kohdalla käyttöjärjestelmän vuoronnuksesta. Vuoronnuksen takia on mahdollista, että ensimmäinen säie on ehtinyt suorittamaan matriisikertolaskun vastuualuettaan jo pitkälle, ennen kuin kaikki säikeet ovat luotu. On kuitenkin hyvä muistaa, että tässä C++-kieli käsittelee moninkertaisesti suurempia matriiseja, joten tehokkuudeltaan kieli on parempi.

Seuraavaksi kuvassa 4 esitetään kielien rinnakkaisten toteutuksien antama nopeutus sekä Amdahlin lain ennuste.

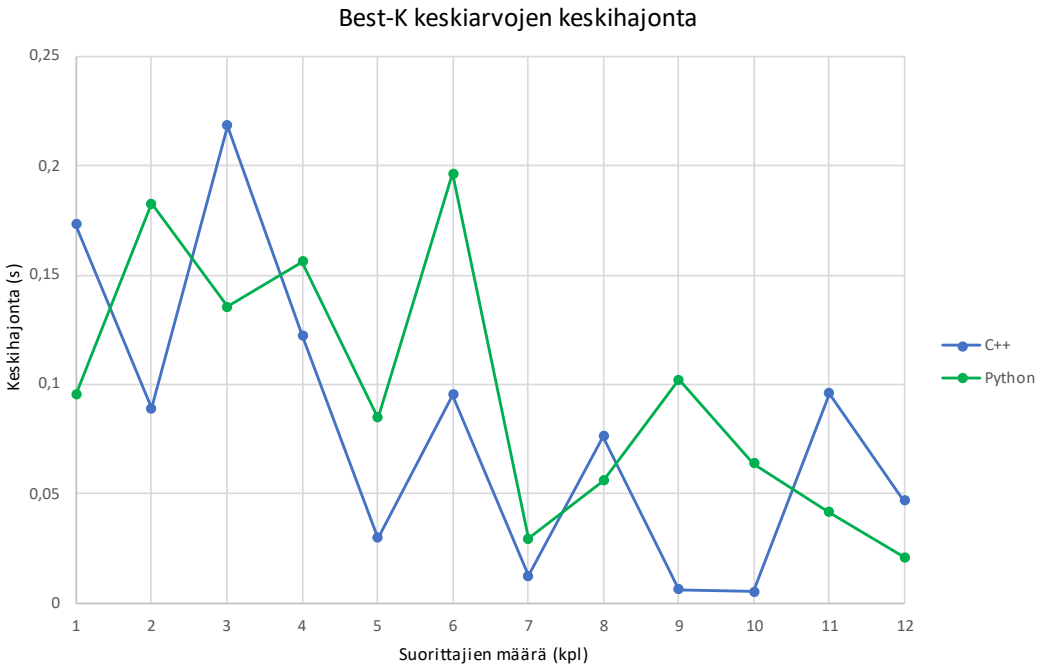


Kuva 4. *Best-K keskiarvojen nopeutus.*

Kuvasta 4 huomataan, että Python-kieli näyttää saavan suuremman nopeutuksen suorittajien lisäämisestä, joten se hyötyy rinnakkaisesta toteutuksesta enemmän kuin C++-kieli. Saavutetun nopeutuksen ero on kuitenkin hyvin pieni ja on myös selitettävissä käyttöjärjestelmän vuoronnuksen avulla. Kuvan 4 kuvaajista huomataan myös, kuinka yksittäisestä suorittajasta saatava nopeushyöty pienenee suorittajien määrän kasvaessa. Varsinkin C++-kielellä kuuden suorittajan jälkeen suorittajista saatu hyöty pienenee vahvasti, mikä kielii työssä käytetyn prosessorin ydinten määrästä. Python-kielellä saatava nopeushyöty näyttää heikkenevän merkittävästi vasta kahdeksan suorittajan jälkeen. Tämä saattaa johtua Python-kielen matalammasta laskennallisesta tehosta, jolloin prosessorin ydinten kuormitus kasvaa hitaammin suorittajia lisätessä. Eroa kielten välillä voi myös selittää vaihtelu tietokoneen yleisessä kuormituksessa mittausten aikana, tosin tätä pyrittiin minimoimaan.

Amdahlin lain ennuste näyttää pitävän paikkansa kolmeen suorittajaan asti, mutta tämän jälkeen todelliset tulokset alkavat hidastumaan ennusteeseen nähden. Kuuden suorittajan jälkeen varsinkin C++-kielen suoritusajat eroavat merkittävästi ennusteesta. Amdahlin laki olettaa, että uudet suorittajat vastaavat aina uutta prosessorin ydintä, joten tämä eroavaisuus oli odotettavissa. Prosessien ja säikeiden käynnistämisestä aiheutuva yleiskustannus suurenee suorittajien määrän kasvaessa, jolloin rinnakkaistettu osuus koodista pienenee. Tämä kasvattaa myös eroa ennusteen ja todellisten tulosten välillä.

Kuvassa 5 on esitetty Best-K keskiarvojen keskihajonta.



Kuva 5. *Best-K keskiarvojen keskihajonta.*

Kuvan 5 kuvaajista huomataan, että saatujen mittaustuloksien keskihajonta on vähäistä, mikä tukee tuloksien luotettavuutta. Tietokoneen muun kuormituksen hetkellinen vaihtelu voi aiheuttaa huomattavia muutoksia tässä mitattavien ohjelmien suoritusaikaan, minkä takia mittauksissa pyrittiin minimoimaan ulkoisten hidastusten vaikutusta. Tuloksista huomataan, että pienemmillä määrillä suorittajia keskihajonta on hieman korkeampaa. Kun suorittajia on vähän, niiden käynnistymisestä aiheutuva yleiskustannus on suuremmassa asemassa, sillä suorittajat tekevät suuremman työn kuin tilanteissa, missä suorittajia on enemmän. Kielten keskihajonnassa ei ole merkittävää eroa, joten molemmat kielistä toimivat vakaasti.

4.2 Vaihtoehtoja rinnakkaisuuteen

Python-kieli ei vastaa laskennalliselta teholtaan C++-kieltä tuloksien perusteella, mutta tämä ei näytä vaikuttavan rinnakkaisuuden skaalautuvuudessa. Pythonin tehokkuutta voidaan parantaa käyttämällä erilaisia ohjelmistokehyksiä kuten CharmPy:ä. Galvez et al. (2018) ovat näyttäneet, että CharmPy:n avulla voidaan kirjoittaa Python-kielisiä rinnakkaisia ohjelmia, jotka skaalautuvat erittäin korkeille ydinmäärille supertietokoneilla ja suoriutuvat samankaltaisesti kuin MPI-pohjaiset tai C++-kielelliset ratkaisut. CharmPy perustuu Charm++:aan, ja sitä ajetaan C++-ajonaikaisessa ympäristössä (Galvez et al.,

2018). CharmPy ajaa siis Python-koodin lopulta C++-koodina ilman, että suorituskyvyssä on merkittävää eroa. Täten ohjelmoija voi kirjoittaa ohjelmatoteutuksensa Python-kielellä ja saavuttaa suorituskyvyllisesti tuloksia, jotka vastaavat C++-kielellisiä ohjelmatoteutuksia.

C++-kielellä rinnakkaisuuden toteutus voi olla monimutkaista, mutta tähän löytyy myös ohjelmoijan työtä helpottavia työkaluja. OpenMP-ohjelmointirajapinta tukee C-, C++- ja Fortran-kieliä rinnakkaisten sovellusten kirjoittamisessa (OpenMP, 2023). OpenMP määrittää siirrettävän ja skaalautuvan mallin varustettuna yksinkertaisella ja joustavalla käyttöliittymällä (OpenMP, 2023). OpenMP:n avulla voidaan vähentää rinnakkaisuuden toteutukseen tarvittavaa koodin määrää.

Monet muut ohjelmointikielet kuten Java kykenevät myös rinnakkaisiin toteutuksiin, joten ohjelmoijalla on runsaasti valinnanvaraa työkalujen suhteen. Kaikki teknologiat eivät ole alustariippumattomia, joten työkalun valinta täytyy miettiä tilannekohtaisesti. Mikäli ohjelmatoteutus ei ole suorituskykykriittinen, ohjelman kirjoittamisessa korostuu mahdollisesti rinnakkaisuuden hallinnoinnin rajapinta tai rinnakkaisuuden toteuttamisen yksinkertaisuus.

5. POHDINTAA

Mittaustulosten pohjalta voidaan todeta, että Pythonin rinnakkaisuuskirjastot kuten työssä käytetty multiprocessing-kirjasto tarjoavat toimivan ratkaisun rinnakkaisuudelle. Suorituskykykriittisissä sovelluksissa on kuitenkin suositeltavampaa käyttää rinnakkaisuuden toteutukseen C++:aa Pythonin sijasta. Tämä näkyi erityisesti käytettyjen matriisien kokojen eroissa. Näyttäisi kuitenkin siltä, että suorituskyvyn erosta kielten välillä päästään eroon erilaisia sovelluskehyksiä käyttämällä.

Prosessien ja säikeiden käytön yleiskustannusten ero jäi tässä työssä epäselväksi. Mittaustulosten perusteella prosessit skaalautuivat paremmin kuin säikeet, mutta tämä saattoi johtua kielten laskennallisen tehon eroista. Hypoteesina oli, että C++-kielen ratkaisu skaalautuisi paremmin mittauksissa, mutta todellisuudessa tilanne oli päinvastoin. Matriisien kokoeroilla ei ollut vaikutusta ohjelmien skaalautuvuuteen, minkä seurauksena epäilykset muistinhallinnan vaikutuksista nopeutuseroon kielten välillä hälventyivät. Mikäli mittaus haluttaisiin kohdistaa vain laskentatehoon, onnistuisi tämä estettä (engl. barrier) käyttämällä. Este on työkalu säikeiden tai prosessien synkronoinnille (Williams, 2019). Estettä käyttämällä suoritusajasta jäisi pois säikeiden ja prosessien käynnistämisestä aiheutuva yleiskustannus.

Olisi mielenkiintoista nähdä, miten tulokset eroaisivat, kun käytössä olisi tehokkaampi tai heikompi prosessori. Jos käytössä olisi moniytimisempi prosessori, voitaisiin myös lisätä mittauksissa käytettävien prosessien ja säikeiden määrää. Tuloksista nähtäisiin, kuinka vahva suorittimen vaikutus on. Mahdollisesti tutkimuksessa voitaisiin myös tarkastella prosessoriarkkitehtuurien vaikutusta.

Tämän työn mittaus voitaisiin toistaa käyttämällä Pythonille tarjolla olevia erilaisia sovelluskehyksiä. Tuloksista voitaisiin nähdä, ovatko toteutukset kielten välillä tehokkuudeltaan lähes samat. Pythonin tulkkauksesta aiheutuva yleiskustannus mietityttää varsinkin tässä tapauksessa. Sovelluskehystä käytettäessä ohjelmassa on kuitenkin pakosti niin sanottu ylimääräinen välikäsi. Tähän liittyen kielten välillä voitaisiin vertailla yleiskustannusta.

6. YHTEENVETO

Tässä työssä tutustuttiin tarkemmin Python- ja C++-ohjelmointikielien rinnakkaisuuden toteutukseen ja tehokkuuteen. Python-kielessä säikeet olivat tarkoitettu asynkroniseen rinnakkaisuuteen, joten samanaikaista suoritusta varten täytyi käyttää prosesseja. C++-kielessä säikeitä voitiin käyttää samanaikaiseen suoritukseen.

Tehokkuudeltaan C++-kielinen ohjelmatoteutus oli moninkertaisesti tehokkaampi kuin Python-kielellä tehty vastaava ohjelmatoteutus, mutta niiden skaalautuvuus rinnakkaisessa kontekstissa oli hyvin samankaltainen. Merkittävin tekijä tehokkuudessa kielten välillä oli käännettävyydessä ja tulkkauksessa. Ohjelman kääntäminen tapahtuu ennen ohjelman ajoa, kun taas tulkkaus tapahtuu ohjelman ajon aikana, mikä vaikuttaa ohjelmassa esiintyvään yleiskustannukseen suuresti. Python-kielellä skaalautuvuus näytti olevan korkeammilla suorittajamäärillä jopa hieman parempi, mitä voidaan selittää mahdollisesti käyttöjärjestelmän vuoronnuksen tai Python-kielen heikomman laskentatehon, ja täten matalamman suorittimen kuormituksen avulla.

Käsitellyille kielille on tarjolla erilaisia ohjelmistokehyksiä, joiden avulla suorituskykyerot kielten välillä poistuvat lähes täysin. Tällöin valinta ohjelmointikielien välillä keskittyy luultavimmin alustalta siirrettävyyteen, rinnakkaisuuden hallinnoinnin rajapintaan tai rinnakkaisuuden toteuttamisen yksinkertaisuuteen.

Aiheesta voisi tehdä runsaasti jatkotutkimusta. Kielille tarjolla olevia ohjelmistokehyksiä voitaisiin vertailla keskenään. Kielien vertailussa voitaisiin keskittyä moniin eri asioihin kuten yleiskustannukseen, virrankulutukseen tai laskentatehoon. Myös käytettävän suorittimen spesifikaatioon pohjautuvia vaikutuksia voitaisiin tutkia kielten välillä.

7. LÄHTEET

AMD. (2023). AMD Ryzen™ 5 1600 Processor. https://www.amd.com/en/products/cpu/amd-ryzen-5-1600 (Haettu 3.3.2023)

Conoci, S., Di Sanzo, P., Pellegrini, A., Ciciani, B. & Quaglia, F. (2021). On power capping and performance optimization of multithreaded applications. *Concurrency and Computation: Practice and Experience, 33*(13), e6205. https://doi.org/10.1002/cpe.6205

Cppreference. (2022). std::thread. https://en.cppreference.com/w/cpp/thread/thread (Haettu 3.3.2023)

Dévai, F. (2017). The Refutation of Amdahl's Law and Its Variants. *Computational Science and Its Applications – ICCSA 2017*. https://doi.org/10.1007/978-3-319-62395-5_33

Dud's, Á & S'ndor, J. (2012). Cache Performance and Efficiency Factors of Parallel Data Structures. *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. https://doi.org/10.1109/ISPA.2012.87

Galvez, J., Senthil, K. & Kale, L. (2018). CharmPy: A Python Parallel Programming Model. *2018 IEEE International Conference on Cluster Computing,* 423-433. https://doi.org/10.1109/CLUSTER.2018.00059

Gusev, M. & Ristov, S. (2014). A Superlinear Speedup Region for Matrix Multiplication. *Concurrency and Computation Practice and Experience, 26*(11), 1847-1868. https://doi.org/10.1002/cpe.3102

Ichnowski, J. & Alterovitz, R. (2014). Scalable Multicore Motion Planning Using Lock-Free Concurrency. *IEEE Transactions on Robotics, 30*(5), 1123-1136. https://doi.org/10.1109/TRO.2014.2331091

Irri, V. (2023). Matrix multiplication. https://github.com/irriv/matrix_multiplication

Lee, J.-G., Jung, E. & Shin, W. (2009). An Asymptotic Performance/Energy Analysis and Optimization of Multi-core Architectures. *Distributed Computing and Networking. ICDCN 2009*. https://doi.org/10.1007/978-3-540-92295-7_13

Liu, J., Chi, L., Gong, C., Xu H., Jiang, J., Yan, Y. & Hu, Q. (2012). High-Performance Matrix Multiply on a Massively Multithreaded Fiteng1000 Processor. *Algorithms*

and Architectures for Parallel Processing, ICA3PP 2012. https://doi.org/10.1007/978-3-642-33065-0_18

OpenMP. (2023). The OpenMP API specification for parallel programming. https://www.openmp.org/

Python. (2020). Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock

Valgrind. (2022). 5. Cachegrind: a cache and branch-prediction profiler. https://valgrind.org/docs/manual/cg-manual.html (Haettu 3.3.2023)

Williams, A. (2019). C++ Concurrency in Action, Second Edition. Manning Publications.

Yao, Y., Shang, J. & Wang, Q. (2019). Optimised multi-hypothesis tracking algorithm based on the two-dimensional constraints and manoeuvre detection. *The Journal of Engineering, 2019*(21), 7677-7682. https://doi.org/10.1049/joe.2019.0750

Zheng, Z., Yu L. & Lan Z. (2015). Reliability-Aware Speedup Models for Parallel Applications with Coordinated Checkpointing/Restart. *IEEE Transactions on Computers, 64*(5), 1402-1415. https://doi.org/10.1109/TC.2014.2317182