Matlab-ohjelmakoodin vektorisointi ja rinnakkaistaminen Python-toteutuksena

Pro gradu -tutkielma
Turun yliopisto
Teknillinen tiedekunta
Data-analytiikka
2022
Jarkko Järvinen

TURUN YLIOPISTO
Teknillinen tiedekunta

JARKKO JÄRVINEN: Matlab-ohjelmakoodin vektorisointi ja rinnakkaistaminen Python-toteutuksena

Pro gradu -tutkielma, 63 s.
Syyskuu 2022

Tutkimus käsittelee Matlab-ohjelmointikielellä toteutetun konseptitoteutuksen ohjelmointikielen vaihtoprosessia Python-kielelle. Toteutuksen avulla tutkitaan maanpinnan muotoja pienessä mittakaavassa. Tavoitteena on tarjota toteutus palveluna tai yleiskäyttöisenä kirjastona helposti ylläpidettävänä kokonaisuutena, nopeampana ja skaalautuvana käyttäen vektorisointia, rinnakkaistamista ja moniydinprosessointia. Tutkimuskysymyksinä käsitellään ohjelmistokielten numeeristen ja operationaalisten erojen huomiointia toteutuksessa, rinnakkaistamisen ratkaisumalleja ja ohjelmistokielen vaihtoprosessin onnistumisen validointia.

Taustatutkimuksen aikana esitellään prosessit ja säikeet sekä niiden luomiseen liittyviä yleisiä käytänteitä, rinnakkaisuuteen liittyviä tekniikoita, haasteita ja niihin liittyviä kommunikointi- ja synkronointiratkaisumalleja, silmukoiden yleisiä vektorisointikäytäntöjä, matriisien indeksointi-, viipalointi- ja ikkunointitapoja Python-koodissa. Toteutusosiossa kuvataan ohjelmointikielen vaihtoprosessi käyttäen yksikkötestejä Matlab-toteutuksesta kerätyn testitiedon avulla, vaihtoprosessissa kohdattuja toteutushaasteita ja niissä käytettyjä ratkaisuja, ohjelmakoodin optimointikeinoja ja lopputuloksen koostamista.

Konseptitoteutuksen ohjelmointikieli vaihdettiin Python-kieleen käyttäen C/C++ -kielellä toteutettuja kirjastoja. Rinnakkaistamisella saavutettiin nopeutusta ja ratkaisuarkkitehtuuriehdotuksen mukaisesti toteutus voidaan tarjota pilvilaskentaparadigmaa hyödyntäen korkeasti saavutettavana skaalautuvana palveluna. Tutkimuksen tuloksien perusteella vektorisointi on tehokas keino saavuttaa nopeutusta. Vektorisointi edellyttää vektoritietoisuutta ja kehittäjän tulee ymmärtää käytetyt algoritmit täydellisesti, jotta vektorisoinnilla saavutetaan oikeanlainen tulos.

Avainsanat: Rinnakkaisohjelmointi, ohjelmointikielet, pilvipalvelut, optimointi

UNIVERSITY OF TURKU
Faculty of Technology

JARKKO JÄRVINEN: Vectorization and parallelization of Matlab implementation as a Python implementation

Master of Science Thesis, 63 p.
September 2022

The research will focus on the process of changing the programming language of the proof-of-concept Matlab implementation to the Python language. The software helps to study the small-scale forms of the earth's surface. The goal is to offer the implementation as a service or a maintainable general-purpose library, also make the software faster and scalable by using vectorization, parallelization, and multi-core processing. The research questions include consideration of the numerical and operational differences of software languages in the implementation, parallelization solution models and validation of the software language exchange process.

During the research, processes and threads are presented with general creation practices. Also described parallelism-related techniques and challenges, parallelism communication and synchronization solution models, general vectorization practices for loops, matrix indexing, slicing and windowing methods with practical Python code fragments. The implementation section describes the process of changing the programming language using unit tests with the test data collected from the Matlab implementation. Including the implementation challenges encountered during the programming language exchange process and the solutions used to solve challenges. Code optimization solutions and gathering the results of the calculations will be covered.

The programming language exchange process completed successfully by using Python libraries implemented in C/C++ language. The implementation can be offered as a highly accessible and scalable service by following the solution architecture proposal utilizing the cloud computing paradigm. Based on the research results vectorization is an effective solution to achieve performance improvement. Vectorization requires vector awareness and the full understanding of the algorithms used to achieve the optimal result.

Keywords: Parallel programming, programming languages, cloud services, optimization

SISÄLLYSLUETTELO

KUVAT

TAULUKOT

1 JOHDANTO

Tutkimus sai alkunsa Älykkäiden ja oppivien järjestelmien (prof. Heikkonen) tutkimusryhmän kiinnostuksesta maanpinnan mikrotopografiaan. Kyseessä on monitieteinen aihepiiri, jossa tutkitaan maanpinnan muotoja pienessä 5–200 cm mittakaavassa. Aiheesta oli aiemmin tehty Matlab-ohjelmakoodilla konseptitoteutus, jonka tavoitteena oli auttaa metsäojien, renkaanjälkien ja muiden urien syvyyksien selvittämisessä fotogrammetrisesta tai lentolaserkeilausdatasta. Näiden löytämisellä voidaan valvoa esimerkiksi metsäkoneiden ajon laatua ja selvittää metsäojien kunto. Sivutavoitteena on ollut luoda koko Suomesta geomorfologinen viitekehys käyttäen pilvilaskentaparadigmaa.

Konseptitoteutuksessa suoritetaan kaltevuusriippumaton mikrotopografinen näkymä maastoon, jossa jokaisen pikselin ympäriltä lasketaan kuudesta suunnasta minimi entropia. Halutut kuviot löydetään entropian ollessa pieni. Pikselidatasta poistetaan puut, jolloin saadaan siistimpi kaarevuuskuva. Tämän jälkeen kuva suodatetaan ja siitä lasketaan maaston kaltevuudet. Maastourien kulkureitit hyödyntäen Kalman-suodatinta ja konvoluutioneuroverkkoa. Toteutusta koestetaan fotogrammetrisia kuvia vasten, jotka olivat otettu Kouvolasta ja Sievistä kesällä 2020. Suunnitelmissa on toteuttaa aiemmin toteutetun konseptitoteutuksen tavoitteen täyttävä toteutus palvelu- tai kirjastomallisena, helpommin ylläpidettävänä kokonaisuutena Python-ohjelmointikielellä. Ohjelmointikielen vaihdon tavoitteena on saattaa ohjelmiston kehittäminen helpommaksi ja yleiskäyttöisemmäksi. Python-ohjelmointikielen osaajia löytyy kehittäjien keskuudesta helpommin kuin Matlab-kielen osaajia. Yhtenä laadullisena tavoitteena on käyttöönottaa yksikkötestaukset tuomaan helpotusta ja varmuutta jatkokehitykseen. Python-toteutuksen tavoitteena on nopeuttaa laskentaa ja lisätä skaalautuvuutta käyttäen vektorisointia, rinnakkaistamista ja moniydinprosessointia.

Teoriaosuudessa käydään läpi tutkintakysymyksien ongelmakenttiä ja pohditaan niihin mahdollisia ratkaisumalleja. Tutkimuskysymyksinä ovat, miten eri ohjelmistokielten välillä huomioidaan numeeriset ja operationaaliset erot sekä miten varmistutaan, että ohjelmistokielen vaihto on onnistunut. Taustatutkimuksessa kartoitetaan silmukoiden vektorisointia, matriisien indeksoinnin eri tapoja, rinnakkaistamisen haasteita ja eri

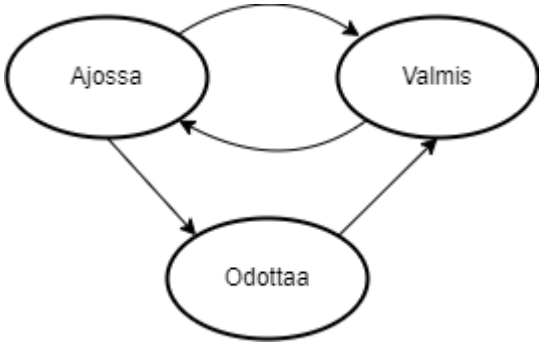
käytänteitä sekä käsiteittä, kuten triviaalisti rinnakkaistuva algoritmi. Toteutusosiossa käydään läpi kielen vaihtoa ohjelmistoprojektin strategisena jäsentäjänä, joista siirrytään käytännön koodifragmenttien esimerkkeihin. Tässä osiossa myös kuvataan prosessia, miten Matlab-toteutus muutetaan Python-toteutukseksi ja miten ratkaisumallit toimivat päähaasteiden kanssa. Tulokset osiossa analysoidaan ja kerrotaan mitä tutkimuksen aikana saavutettiin ja saavutettiinko uudella toteutuksella alkuperäisen kaltaista suoritusta sekä reflektoidaan onnistumisia ja epäonnistumisia. Lisäksi pohditaan mitä olisi voinut tehdä toisin ja mitä kannattaa huomioida mahdollisessa jatkotutkimuksessa. Tutkimuksessa käydään lopulta kehittyneempiä arkkitehtuurisia ratkaisuja läpi, joilla tutkimuksen aikana luotu Python-ohjelma voitaisiin tarjota paremmin saavutettavana ja skaalautuvana palveluna esimerkiksi pilvilaskentaparadigmaa hyödyntäen. Ratkaisuarkkitehtuuriehdotus tarjoaa ratkaisumallin, jonka avulla tämän kaltainen vaatimus voidaan saavuttaa.

2 TAUSTATUTKIMUS

Taustatutkimuksen aikana käydään läpi perusteoriaa prosesseista ja säikeistä, rinnakkaisuudesta ja vektorisoinnista. Prosessien ja säikeiden luomiseen löytyy monia hyväksi havaittuja käytäntöjä, joita suositellaan käytettävän ohjelmistoissa. Taustatutkimuksessa esitellään muutamia yleisimpiä käytänteitä. Yleisistä käytännöistä huolimatta rinnakkaisuus tuo mukanaan uusia haasteita ja rajoituksia, jotka täytyy ottaa huomioon ohjelmistoa suunniteltaessa. Näitä haasteita voidaan välttää käyttämällä kommunikointi ja synkronointiin rinnakkaisuudessa. Rinnakkaisuutta voidaan tavoitella myös vektorisoinnilla. Vektorisoinnin edellytykset voidaan selvittää riippuvuusanalyysilla ja profiloinnilla, jotta vältytään myöhemmiltä toteutusvaiheen yllätyksiltä. Vektorisointi voi olla mahdotonta suorittaa koodissa olevien riippuvuuksien takia ja koodi voidaan joutua uudelleenkirjoittamaan vektorisoinnin takia. Python on selkeä ja tehokas olio-ohjelmointikieli, joka sisältää tehokkaat indeksointi- ja viipalointitoiminnot sekä suorituskyvyltään hyvät profilointi- ja vertailuanalyysityökalut. Sisäkkäiset silmukat voivat aiheuttaa Python-koodissa pullonkauloja. Erilaisia C/C++ -kieleen pohjautuvia laajennuksia ja kirjastoja käyttäen silmukoiden vektorisointi ja rinnakkaistaminen sekä ikkunointitoiminnot voidaan järjestää Python-kielessä tehokkaasti.

2.1 Prosessit ja säikeet

Tietokoneella ajettavaa ohjelmaa kutsutaan prosessiksi. Tietokoneen yksittäisessä suorittimessa voidaan ajaa yhtä prosessia kerrallaan. Moniydinjärjestelmässä prosesseja voi olla prosessorin ytimien verran, mikä mahdollistaa tehokkaamman ajon. Prosessi sisältää tilatiedon, joka kuvaa prosessin suoritusta. Prosessilla on kolme perustilaa, jotka esitetään niiden välisine siirtymineen kuvassa (Kuva 1). Nämä tilat ovat Ajossa, Odottaa ja Valmis. Prosessi on Ajossa-tilassa, kun se suorittaa ohjelmaa. Valmis-tilassa prosessi on valmis suoritettavaksi, mutta käyttöjärjestelmä on päättänyt olla suorittamatta sitä tällä hetkellä. Odottaa-tilassa prosessi suorittanut jonkinlaisen toiminnan ja se on jäänyt odottamaan jonkin tapahtuman suorittamista. Uudet prosessit tulevat Valmis-tilaan ja päättyvät prosessit poistuvat Odottaa-tilan kautta. [1]



Kuva 1. Prosessien tilakaavio [1, s. 50]

Modernit käyttöjärjestelmät tukevat perinteistä prosessia kevyempää versiota prosessista eli säiettä. Yhdessä prosessissa voi olla yksi tai useampi säie. Säikeellä ei ole omia resursseja, vaan ne käyttävät prosessin resursseja. Prosessin tehtävänä on tarjota joukolle säikeille yhteinen muisti ja yhdistää ne loogisesti. Rinnakkainen ohjelma suorittaa useita säikeitä tai prosesseja yhtäaikaisesti. Nuo prosessit ja säikeet kommunikoivat keskenään sekä käyttävät yhteisiä tietorakenteita. Säikeet voidaan jakaa suoritinajan käytön perustella kahden tyyppisiksi. Suoritinsidonnaisilla säikeillä tarkoitetaan säikeitä, jotka käyttävät runsaasti suoritinaikaa, kuten teknisen laskennan ongelmia suorittavat säikeet. Siirräntäsidonnaiset säikeet taas tekevät suhteellisen usein suoritusaikaan nähden siirräntäpyyntöjä. Tällainen tilanne ilmenee, kun odotetaan käyttäjältä tulevaa syötettä. [1]

2.2 Vuoronnus

Seuraavaksi suoritettavan prosessin, säikeen tai työn valintamenetelmää kutsutaan termillä vuoronnus (scheduling). Jatkossa tämän kappaleen aikana pienintä valittavaa yksikköä kutsutaan sanalla säie. Valintaoperaation suorittaa vuorontaja (scheduler), jonka tavoitteena yleisesti on optimoida järjestelmän suorituskykyä. Tämä suoritetaan optimoimalla vastausaika, kokonaissuoritusteho (throughput) ja reiluus (fairness). Järjestelmän kokonaissuoritusteholla mitataan suoritettujen töiden määrää aikayksikköä kohti. Reiluudella pyritään suoritusaikaa jakamaan säikeiden välillä siten, ettei yksikään joudu odottamaan vuoroaan liian pitkään. Reiluus taas hankaloittaa vastausajan vaihteluvälin (jitter) laskentaa. Optimointitapa vaihtelee hieman erätöiden (batch job), osituskäytön ja reaaliaikajärjestelmien tapauksissa. Erätöissä järjestelmässä on suurempi

liikkumavara optimoida järjestelmän kokonaissuoritusta. Osituskäytössä pyritään ottamaan huomioon päätteillä odottavien käyttäjät, jotta käyttäjät saavat suunnilleen samantasoisen keskimääräisen vasteajan. Reaaliaikajärjestelmissä pyritään järjestelmän vasteajat pitämään aikarajojen puitteissa, jolloin ajonaikaiset prioriteetin vaihdokset eivät ole mahdollisia. Vuoronnuksen sanotaan olevan irrottamaton, kun ajossa olevaa prosessia ei koskaan vaihdeta toiseen ilman prosessin lupaa. Irrottavassa vuoronnuksessa prosessi voidaan vaihtaa kesken suorituksen prosessin lupaa kysymättä. [1]

Vuorontaja käyttää erilaisia menetelmiä säikeiden vuorontamisessa. Jonoperustaisessa menetelmässä (FIFO, first in first out) säikeet suoritetaan siinä järjestyksessä, jossa ne saapuvat Valmis-tilaan. Menetelmä on reilu, mutta suoritinsidonnaiset säikeet viivyttävät tarpeettomasti siirräntäsidonnaisten säikeiden etenemistä ja ikuinen silmukka pysäyttää muut säikeet. Kiertovuorottelumenetelmä (round robin) toimii kuten jonoperustainen menetelmä, mutta siihen on lisätty irrottavuutta ja reiluutta. Säie saa olla Ajossa-tilassa yhtäjaksoisesti sille annetun aikaviipaleen määräämän ajan verran. Näin jokainen säie saa vuorollaan aikaviipaleen määrittämän ajan toimia ja näin suoritinsidonnaiset säikeet eivät hidasta siirräntäsidonnaisia säikeitä. Säikeiden lukumäärän kasvaessa suorituksen aikana, kasvaa myös aikaviipaleen odotusaika. Säikeitä voidaan myös priorisoida. Kiinteän prioriteetin (fixed priority) menetelmässä suoritukseen valitaan korkeimman prioriteetin omaava säie. Vaihtelevan prioriteetin (dynamic priority) menetelmässä säikeen prioriteettia muutetaan ajan funktiona. Vuorottelumenetelmät ovat yleensä eri menetelmien yhdistelmiä. Käyttöjärjestelmissä käytetään normaalisti kiinteitä korkeita prioriteetteja käyttöjärjestelmän omille säikeille ja käyttäjän säikeissä sovelletaan vaihtelevaa prioriteettia yhdistettynä kiertovuorotteluun saman prioriteetin säikeiden kesken. [1]

2.3 Rinnakkaisuus

Ohjelmistoissa rinnakkaisuudella tarkoitetaan yhtä aikaa ajossa olevia prosesseja tai säikeitä, jotka kommunikoivat keskenään käsittelemällä yhteisiä tietorakenteita tai lähettämällä sanomia toiselleen. Rinnakkaisuudella yleensä pyritään saavuttamaan suoritusnopeutta, toteutuksen tehostamista modularisoinnilla tai hajauttaa kuormitusta esimerkiksi tietokoneverkkoon. Joskus jotkut ohjelmointiongelmat ovat luonteeltaan sellaisia, että niiden ratkaisu on järkevää hoitaa rinnakkaistamisella. [1]

Rinnakkaisuus tuo ohjelmiin neljä ongelmaa, jotka ovat poissulkeminen, synkronisointi, lukkiutuminen ja nälkiintyminen. Poissulkeminen muodostuu, kun useampi prosessi päivittää yhteistä tietorakennetta yhtä aikaa. Semafori on yksi keino estää poissulkemisongelma, jolla pyritään lukitsemaan jokin tietty kriittinen alue, esimerkiksi tiedon tallennus jaettuun muuttujaan. Synkronisointi ilmaantuu, kun prosessit pitäisi tahdistaa toistensa kanssa. Tämä ongelma voidaan yrittää ratkaista käyttämällä ajastettuja odotuksia, joissa prosessi odottaa annetun ajan tiedon saantia, jonka jälkeen se joko tekee virheen tai jatkaa toimintaansa. Lukkiutuminen tapahtuu, kun prosessit odottavat toisen prosessin tulosta, joka taas odottaa toisen prosessin tulosta ja niin edelleen. Lukkiutuminen voidaan joko laukaista, välttää tai estää. Laukaiseminen on ohjelmallisesti hankalaa. Yleensä käyttäjän vastuulle jää tuhota jokin lukossa oleva prosessi. Välttämisessä prosessin tarvitsemat kaikki resurssit varataan heti aluksi tai erilaisilla algoritmeilla prosessin aikana. Lukkiutumisen estäminen suoritetaan poistamalla jokin lukkiutumisen välttämättömistä ehdoista. Nälkiintymisellä tarkoitetaan ongelmaa, kun prosessi voisi jatkaa, mutta se ei jostain syystä saa suoritusaikaa. Tämä voidaan estää lisäämällä prosesseille synkronointikohtia, priorisoinnin vaihtelulla ja käyttämällä jonoissa FIFO-menetelmää. [1] [2]

Rinnakkaisuutta sisältävän ohjelmiston testaus voi olla hankalaa, mikäli sen toiminta on näennäisesti epädeterministinen. Testaus hankaloituu, kun ohjelman lopputulokseen voidaan vaikuttaa siinä suoritettavien tapahtumien keskinäisellä järjestyksellä tai niiden välisellä suoritusajalla. Tällöin testauksessa käytettävien yhdistelmien määrä ei ole rajallinen ja niiden lukumäärä voi nousta niin suureksi, että niitä voi olla lähes mahdoton luoda. Lisäksi ohjelman kehittäminen hankaloituu, sillä vikatilanteen toistaminen korjauksen testaamiseksi muuttuu mahdottomaksi. Virhetilanteet tulevat esiin ajallisesti paljon myöhemmin varsinaisen tapahtumisen jälkeen. Tällainen tilanne voi esiintyä, kun prosessi 1 tuhoaa prosessin 2 tallentaman tiedon, kun prosessi 2 odottaa jotain tapahtumaa. Kun prosessi 2 pääsee vihdoin jatkamaan suoritustaan, on sen tarvitsema data turmeltunut. Tämä voi sattua myös sarjallisessa koodissa, mutta rinnakkaisessa koodissa toiminnan ja tiedon oikeellisuuden varmistaminen on hankalampaa. [1]

Ohjelman rinnakkaistaminen edellyttää, että se voidaan jakaa rinnakkain ajettaviin osiin ja niissä loogisesti sarjamuotoiset operaatiot on aina suoritettava peräkkäin. Ohjelma voidaan jakaa rinnakkaisiin osiin funktionaalisesti tai tietoperustaisesti.

Funktionaalisessa osituksessa ohjelma jaetaan rinnakkain suoritettaviin säikeisiin esimerkiksi jakamalla laskennalliset tehtävät laskentasäikeiden kesken. Tietoperustaisessa osituksessa käytettävä tieto jaetaan säikeille, jotka suorittavat tehtäviä rinnakkaisesti. Jotkut ongelmat ovat triviaalisti rinnakkaistuvia, jotkut hieman vaikeampia, jotkut vaativat algoritmin kokonaan uudelleen kirjoittamista ja jotkut ovat mahdottomia rinnakkaistaa. [3]

2.3.1 Triviaalisti rinnakkaistuva algoritmi

Triviaalisti rinnakkaistuva algoritmi (embarrassingly parallel algorithm) tarkoittaa algoritmia, joka on hyvin helposti rinnakkaistettavissa. Yleensä tämä tarkoittaa sitä, että algoritmi on helppo jakaa osiin, jotka suoritetaan eri prosesseissa. Prosessien tuloksien yhdistäminen ei vaadi suurta käsittelyä, ja prosessit voivat toimia itsenäisesti ilman toisten kanssa kommunikointia. Rinnakkaistaminen muuttuu hieman vaikeammaksi, kun syötetaulukon arvojen perusteella lasketaan summaa, minimi-, maksimi- tai keskiarvoa.

```
Function sum(arr, len):
    res = 0
    For i = 1 To len:
       res += arr[i]
    EndFor
    return res
```

Tässä tapauksessa res-muuttujan arvo riippuu edellisestä arvosta, joten ongelmaa ei voida jakaa kahteen osaan. Voidaan kuitenkin hyödyntää + operaattorin assosiatiivisuutta, jolloin $(a + b) + c = a + (b + c)$. Tämä tarkoittaa, että yhteenlasku voidaan suorittaa missä tahansa järjestyksessä. Näin rinnakkaistaminen voidaan suorittaa esimerkiksi siten, että säikeessä 1 suoritetaan i arvot välillä 0 ja 999 ja säikeessä 2 suoritetaan i arvot välillä 1000 ja 1999 ja niin edelleen. Lopputulos saadaan laskemalla yhteen jokaisen säikeen tuottama summa. Käytännön läheisempänä esimerkkinä tämän tyyppistä lähestymistä voidaan käyttää kuvan prosessoinnissa, jossa suurikokoinen kuva jaetaan pienempiin osiin ja prosessointi suoritetaan rinnakkain pienemmille osille sen

sijaan, että suoritetaan prosessointi koko kuvalle kerralla. Prosessointien tulokset yhdistetään lopulta yhdeksi kuvaksi. [3] [4]

2.3.2 Rinnakkaistamisen haasteet

Rinnakkaistaminen muuttuu huomattavasti vaikeammaksi esimerkiksi, kun algoritmin silmukassa esiintyy kilpailutilanne. Kilpailutilanne ilmenee esimerkiksi, kun lukuja käsitellään rinnakkain ja val-taulukon alkiot indekseissä 1 ja 100 sisältävät saman arvon. Tällöin histogram-taulukko voi saada väärän arvon kilpailutilanteen vuoksi.

```
Function histogram(histogram, val, len):
    For i = 1 To len:
        index = val[i]
        histogram[index]++
    EndFor
```

Syötearvotaulukon val kaikki arvot tulee huomioida histogram-taulukon summaa laskiessa. Tämä voidaan ratkaista käyttämällä atomista taulukkoa tai säilyttämällä paikallisesti jokaisen rinnakkain ajettavan suorituksen histogram-taulukkoa lopulta yhdistäen niiden arvot yhdeksi taulukoksi. Ensimmäinen tapa on hitaampi atomisuuden hallinnan vuoksi, mutta tehokkaampi suurella määrällä rinnakkaisia suorituksia. Toinen lähestymistapa on nopeampi, mutta kuluttaa enemmän muistia ja lopussa on rinnakkaisten ajojen tuloksien yhdistämistoimenpide. [3]

Luonteeltaan peräkkäisiä algoritmeja on erittäin vaikea tai mahdoton rinnakkaistaa. Tällaisia algoritmeja käytetään usein kryptografiassa tehden salauksen rikkomisesta mahdotonta vain lisäämällä laskentakapasiteettia. Tällaisesta esimerkkinä hajautusalgoritmin ketjutus, jossa tiivistefunktiota h kutsutaan peräkkäin merkkijonolle x.

$$h^n(x) = h(h\left(\dots\left(h\left(h(x)\right)\right)\right))$$

2.3.3 Amdahlin laki

Amdahlin lain avulla voidaan määrittää teoreettisesti, kuinka paljon sarjallinen ohjelma nopeutuu käytettäessä rinnakkaisuutta. Tietoa voidaan käyttää päätöksen tekoon, kannattaako sarjallinen ohjelma rinnakkaistaa lisäämällä käytettävien suorittimien määrää vai jatketaanko sarjallisen ohjelman suorituskyvyn optimointia. Lisäksi voidaan selvittää, kuinka suuri osa sovelluksesta on rinnakkaistettava ennen kuin siitä saavutetaan merkittävä hyöty. Rinnakkaistamisesta saatavaa nopeutusta voidaan tarkastella vertaamalla sarjallisen suorituksen toteuttamista rinnakkaiseen suoritukseen. [4]

Olkoon $T(p)$ laskenta-aika, kun ongelman ratkaisemiseen käytetään rinnakkain p suoritinta ja $T(1)$ on sarjalliseen ratkaisemiseen kuluva aika. Suhteellinen nopeutus $S(p)$ on:

$$S(p) = \frac{T(1)}{T(p)}$$

Oletetaan, että ongelmasta osuus r voidaan rinnakkaistaa. Tällöin laskenta-aika $T(p)$, joka kuluu rinnakkaistettavaan osaan on $(1-r)\,T(1)$. Kun käytetään p suoritinta, laskenta aika on vähintään:

$$T(p) \geq (1-r)T(1) + \frac{rT(1)}{p}$$

ja suhteellinen nopeutus $S(p)$ on korkeintaan:

$$S(p) \leq \frac{T(1)}{(1-r)T(1) + \frac{rT(1)}{p}} = \frac{p}{(1-r)p + r}$$

Tämä tunnetaan nimellä Amdahlin laki.

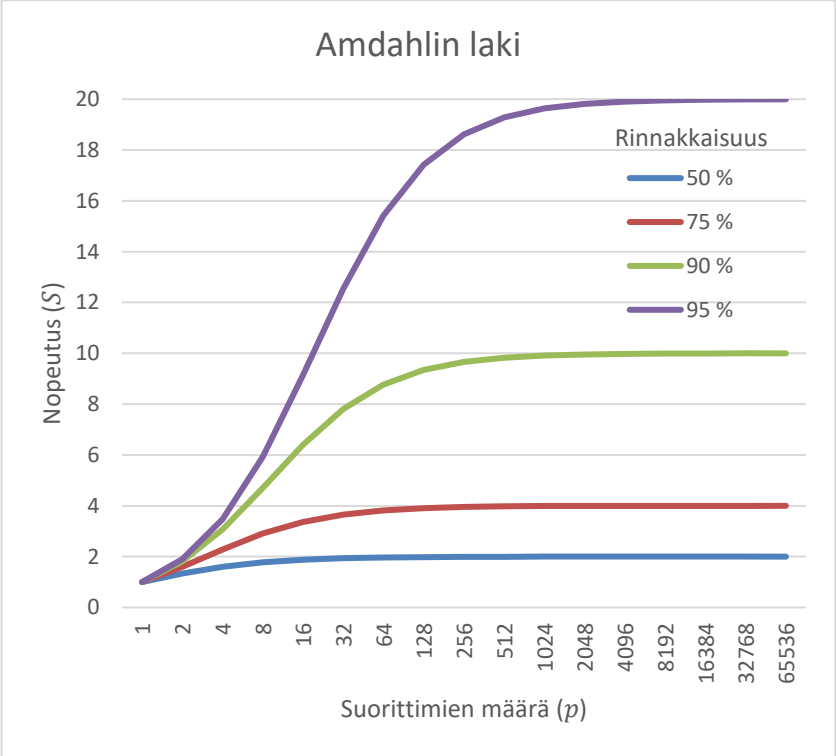
Amdahlin laki voidaan sovittaa esimerkkiin, jossa 40 % ongelmasta voidaan rinnakkaistaa käyttäen neljää suoritinta. Täten voidaan laskea nopeutuksen olevan:

$$S = \frac{4}{(1 - 0{,}4)4 + 0{,}4} \approx 1{,}43$$

Amdahlin laki asettaa siis suoritusajan nopeutumiselle ylärajaksi:

$$\begin{cases} s \leq \dfrac{1}{1-r} \\ \lim\limits_{p \to \infty} S = \dfrac{1}{1-r} \end{cases}$$

jossa $\frac{1}{1-r}$ on yläraja sille, kuinka paljon samanaikaisuudella ja rinnakkaisuudella voidaan parantaa ohjelman suoritusaikaa. Riippumatta siitä kuinka monta käytettävissä olevaa suoritinta järjestelmässä on, raja määräytyy ohjelman sarjallisen osan mukaan (Kuva 2).



Kuva 2. Nopeutuskäyrät eri rinnakkaistetuilla osuuksilla

2.3.4 Prosessien ja säikeiden luominen

Prosessien ja säikeiden luonti voidaan järjestää staattisesti tai dynaamisesti. Staattisessa mallissa jokin kiinteä määrä prosesseja tai säikeitä luodaan ohjelman alussa. Ne ovat käytössä koko ohjelman elinkaaren ajan ja ne lopetetaan ohjelman lopussa. Dynaamisessa mallissa ohjelman käynnistyessä on yksi aktiivinen prosessi tai säie, joka suorittaa pääohjelmaa. Pääohjelmassa prosesseja tai säikeitä voidaan luoda ja lopettaa missä tahansa vaiheessa ohjelman ajoa. Prosessien ja säikeiden hallintaan löytyy monia hyväksi havaittuja malleja, joita käyttämällä ohjelmistonkehitys helpottuu. Syntaktiset ja semanttiset yksityiskohdat vaihtelevat huomattavasti eri ohjelmointikielien ja kirjastojen välillä, mutta useimmat noudattavat yhtä seuraavista vaihtoehdoista. Vaihtoehdot ovat yhteinen aloitus, rinnakkaiset silmukat, esittelyiden työstämiseen liittyvä aloitus, haarautumat ja liittymät, implisiittinen vastaaminen, aikaistettu kuittaus, liukuhihna, tehtäväaltaat ja tuottaja–kuluttaja. [3] [5]

2.3.4.1 Yhteinen aloitus

Yhteinen aloitus (parbegin-parend, cobegin-coend) mahdollistaa usean operaation yhtäaikaisen suorittamisen. Jokainen yhteisen aloituksen lauseke stmt_1, …, stmt_n on yleensä aliohjelmakutsu kutsu, joista jokainen käynnistää säikeen. Säikeiden luonnin jälkeinen stmt_n+1-lauseke suoritetaan vasta, kun kaikki aiemmat käynnistetyt säikeet ovat lopettaneet työnsä tai ne on lopetettu. [5]

```
co-being
      stmt_1
      stmt_2
      …
      stmt_n
end
stmt_n+1
```

2.3.4.2 Rinnakkaiset silmukat

Rinnakkaiset silmukat (parallel loops) mallissa määritetään silmukka, jonka iteraatiot tapahtuvat rinnakkaisesti. Esimerkiksi C#-kielinen toteutus, jossa säikeen numerot tulostuvat rinnakkaisissa säikeissä konsoliin.

```
Parallel.For(0, 100, i => {
      Console.WriteLine(”Thread ” + i);
});
```

Kolmas parametri esimerkin Parallel.For -lausekkeessa on lambdalausekkeena esitelty (lambda expression) delegaatti. Monissa ohjelmointikielessä ohjelmoija on vastuussa silmukassa suoritettavan koodin yhtäaikaisuuden hallinnasta, ettei silmukassa muodostu kilpailutilannetta (kappale 2.3.2). [5]

2.3.4.3 Esittelyiden työstämiseen liittyvä aloitus

Monissa ohjelmointikielissä, esimerkiksi Ada, esittelyiden työstämiseen liittyvän aloitusmallia (launch-at-elaboration) käyttävän proseduurin P koodi näyttää parametrittomalta aliohjelmakutsulta. Tehtävällä T on oma aloitus ja lopetusosio, jonka sisällä tehtävä määritetään. T käynnistetään heti kun P suoritetaan. Jos P on rekursiivinen, silloin T instansseja voi olla monta yhtä aikaa. Proseduurin lopussa P odottaa luotujen T loppuun suorittamista ennen kuin P lopetetaan. [5]

```
procedure P is
task T is
...
end T;
begin -- P
...
end P;
```

2.3.4.4 Haarautumat ja liittymät

Haarautumat ja liittymät mallissa (fork-join) säie luodaan haarautumassa ja käynnistetään eksplisiittisesti. Liittymä odottaa aiemmin haaroitettujen säikeiden loppuun suorittamista. Esimerkiksi C#-kielessä haarautuma käynnistetään Start-metodilla ja liittymä suoritetaan Join-metodilla. [5]

```
var rend = new ImageRenderer();
var thread = new Thread(new ThreadStart(rend.Render));
thread.Start();
…
thread.Join();
…
class ImageRenderer {
      public void Render() => … // Thread code
}
```

2.3.4.5 Implisiittinen vastaaminen

Implisiittinen vastaaminen (implicit receipt) on samankaltainen kuin haarautuma, mutta aloittaa uuden säikeen toisessa muistialueella kuin aloittaja. Yleensä tämä tarkoittaa sitä, että kutsuja on yksi järjestelmä ja kutsuttava metodi sijaitsee toisessa järjestelmässä. Malli on samankaltainen kuin hajautetuissa järjestelmissä käytetty asiakas–palvelin arkkitehtuuri (client–server model), jolloin useilta asiakkailta tulleiden kutsujen perusteella luodaan automaattisesti uudet tehtävän suorittavat säikeet. Tehtävän suorittamisen jälkeen palvelin palauttaa asiakkaalle vastauksen. [5]

2.3.4.6 Aikaistettu kuittaus

Yleensä työsäikeen elinkaari loppuu, kun se palauttaa vastauksen pääsäikeelle. Aikaistetun kuittauksen (early reply) malli mahdollistaa vastauksen palauttamisen ja jatkaa suoritusta sen jälkeen esimerkiksi toteuttaa pääsäikeen pyytämä työ, palauttaa sen vastaus ja sen jälkeen päivittää työsäikeessä lokeja. Mallia käytetään yleensä asiakas–palvelin arkkitehtuurissa. Asiakas tekee palvelimelle työpyynnön ja saa kuittauksen välittömästi, kun työpyyntö on vastaanotettu. Palautettu kuittaus voi olla työpyynnön yksilöivä tunniste, jonka avulla voi palvelimelta tiedustella pitkäkestoisen työpyynnön tilaa. Palvelin jatkaa esimerkiksi tallentamista ja muita työpyyntöön liittyviä työtehtäviä. [5]

2.3.4.7 Liukuhihna

Liukuhihnamalli (pipelining) on erityinen säikeiden hallintamalli, jossa säikeestä toiseen välitetään dataelementtejä erilaisten käsittelyvaiheiden jälkeen. Säikeet $T_1, \dots, T_p$ on järjestetty ennalta määriteltyyn järjestykseen siten, että säie T_i vastaanottaa säikeen T_{i-1} lähdön ja tuottaa lähdön, joka syötetään seuraavalle säikeelle T_{i+1}, kun $i = 2, \dots, p-1$. Säie T_1 saa syötteensä toiselta ohjelman osalta ja säie T_p antaa lähtönsä toiselle ohjelman osalle. Tästä prosessointivaiheiden riippuvuudesta huolimatta liukuhihnan säikeet voivat toimia rinnakkain soveltamalla niiden käsittelyvaihetta eri tietoihin. Tämä voidaan toteuttaa siten, että tieto jaetaan useaksi dataelementtien virraksi, jotka ajetaan liukuhihnavaiheiden läpi peräkkäin. [3]

2.3.4.8 Tehtäväaltaat

Tehtäväallasmallissa (task pools) suoritettavat tehtävät tallennetaan tietovarastoon, josta ne voidaan hakea suoritettavaksi. Tehtävä sisältää ohjeistukset tehtävän toteuttamiseen, suoritettavat tiedot esimerkiksi laskutoimitus ja sen käyttämät tiedot. Laskutoimitukset määritellään yleensä funktiokutsuina. Tehtävien käsittelyyn käytetään kiinteää määrää säikeitä. Säikeet luodaan ohjelman alkaessa pääsäikeen toimesta ja ne päätetään vasta, kun kaikki tehtävät on käsitelty. Säikeille tehtäväallas on yhteinen tietovarasto, josta ne noutavat tehtäviä suoritettavaksi. Tehtävän käsittelyn aikana säie voi luoda uusia tehtäviä ja lisätä ne tehtäväaltaaseen. Tehtäväaltaan pääsyä on synkronoitava kilpailutilanteen välttämiseksi. Tehtäväallaspohjaisen rinnakkaisohjelman suoritus päättyy, kun

tehtäväallas on tyhjä ja jokainen säie on lopettanut viimeisen tehtävänsä käsittelyn. Tehtäväallasmalli on hyödyllinen sovelluksissa, joissa suoritettavien laskelmien määrä ei ole kiinteä ohjelman käynnistyessä. Lisäksi se takaa joustavan mallin, sillä tehtäviä voidaan luoda dynaamisesti missä tahansa vaiheessa ohjelman suorittamisen aikana. [3]

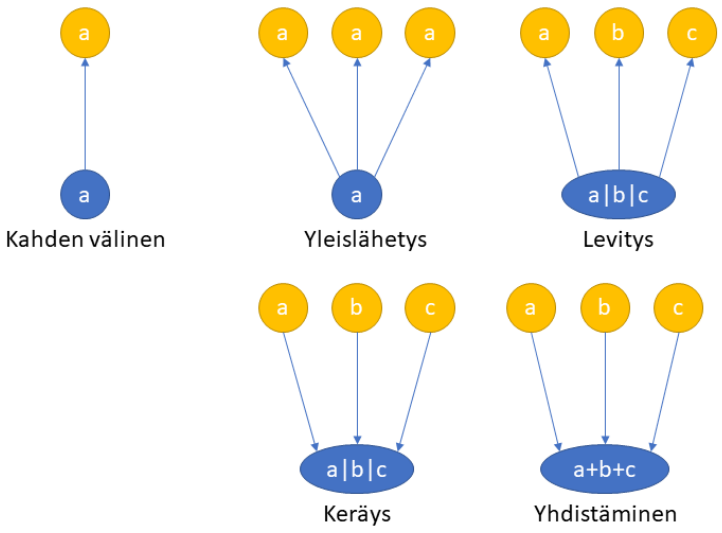
2.3.4.9 Tuottaja–kuluttaja

Tuottaja–kuluttaja mallissa (producer–consumer) tuottajasäikeet tuottavat tietoa käyttäen yhteistä tietomallia jonkun yhteisen tietopuskurin kautta, jota kuluttajasäikeet lukevat ja käyttävät sieltä saatua tietoa syötteenä. Tuottaja voi tallentaa tietoa puskuriin vain, jos se ei ole täynnä. Kuluttaja voi hakea tietoa puskurista, jos tämä ei ole tyhjä. Tämän takia synkronointia tarvitaan tiedon kulun varmistamiseksi tuottajien ja kuluttajien välillä. [3]

2.3.5 Kommunikointi ja synkronointi rinnakkaisuudessa

Triviaalisti rinnakkaistuvat algoritmit, kuten kahden vektorin yhteenlasku, voidaan rinnakkaistaa ilman kommunikointia ja synkronointia. Kommunikaatiota tarvitaan rinnakkaisuudessa, kun suorittimien välillä jaetaan tietoa. Kommunikaatio prosessien välillä voi rajoittaa rinnakkaisen ohjelman suorituskykyä johtuen kommunikaation aiheuttamasta viiveestä. Kommunikaatio imperatiivisessa ohjelmassa järjestetään yleensä jaetun muistin tai viestinvälitysmallin avulla. Jaetun muistin mallissa rinnakkaiset säikeet jakavat paikallisen tai hajautetun muistialueen samaan aikaan. Tällöin yksi säie voi kirjoittaa muistialueelle ja toinen säie voi lukea saman muistialueen. Viestinvälitysmallissa säikeillä ei ole yhteistä tilaa, vaan kommunikaatio prosessien välillä toteutetaan eksplisiittisesti. Viesti sisältää tiedon jostain kiinnostavasta tapahtumasta esimerkiksi ”laskenta valmis”. Viestinvälitysmalli on suorituskyvyllisesti huonompi ratkaisu verrattuna jaetun muistin malliin, sillä tieto kulkee yleensä verkkoliitäntöjen kautta. [3] [5]

Kommunikointi voi tapahtua kahdesta useaan tehtävän välillä (Kuva 3). Kahden välisessä viestinnässä (point to point) kaksi tehtävää kommunikoi keskenään, jolloin toinen tehtävistä on lähettäjä ja toinen vastaanottaja. Ryhmäkommunikoinnissa (collective) useat tehtävät osallistuvat kommunikointiin. Ryhmäkommunikointi voi tapahtua käyttäen yleislähetys- (broadcast), levitys- (scatter), keräys- (gather) tai yhdistämismallia (reduction). [3]



Kuva 3. Kommunikoinnin laajuus

Kommunikointi voi olla synkronista tai asynkronista. Synkroniseen kommunikointiin liittyy tyypillisesti kättely (handshaking), jolloin viestin vastaanottaja kuittaa viestin ja lähettäjä saa tiedon onnistuneesta siirrosta. Prosessien tulee odottaa, kunnes kommunikointi on saatettu loppuun. Asynkroniseen kommunikointiin ei liity kättelyä, jolloin viestin lähettäjä ei odota kuittausta. Tällöin prosessit eivät odota kommunikoinnin loppumista. Prosessissa laskenta ja kommunikointi voivat tapahtua samanaikaisesti. Tämä tapa parantaa useimmissa tilanteissa ohjelmakoodin tehokkuutta. [3] [5]

Kommunikointi helpottuu käyttäen synkronointitekniikoita. Synkronointi on mekanismi, joka mahdollistaa ohjelmassa eri säikeiden toimintojen suorituksen koordinoinnin. Sen avulla voidaan varmistua, että kaikki tehtävät ovat suorittaneet jonkin toiminnon. Synkronointi on viestinvälitysmallissa yleensä implisiittistä, toisin kuin jaetun muistin mallissa. Viestinvälitysmallissa viesti täytyy lähettää ennen kuin se voidaan vastaanottaa. Mikäli vastaanottaja yrittää lukea lähettämätöntä viestiä, jää se odottamaan lähettäjän viestiä. Jaetun muistin mallissa vastaanottava säie voi lukea muuttujan vanhan arvon ennen kuin lähettäjä on kirjoittanut sen, jollei tuona hetkenä tehdä jotain erityistä. Synkronointi jaetun muistin mallissa voidaan toteuttaa kiireinen odotustekniikalla (busy-waiting) ja odottamaan pysähtyvillä säikeillä (blocking). [3] [5]

Kiireinen odotustekniikassa säie tarkastelee silmukan sisällä jonkin muuttujan tilaa. Kun muuttujan ehto on tosi, toteutetaan ehtolauseen haaran sisältämä ohjelmakoodi. Tämän tekniikan avulla varmistutaan, että jokin toiminto on suoritettu esimerkiksi viestijono ei ole tyhjä tai jaettu muuttuja sisältää tietyn arvon. Odottamaan pysähtyvää säiettä käytetään siten, että odottava säie vapauttaa sen käyttämän suorittimen jollekin toiselle säikeelle. Ennen vapauttamista tallennetaan vapauttavan säikeen tilasta tietoa yhteiseen tietorakenteeseen. Vapautetun suorittimen käyttöönottava säie muuttaa myöhemmin yhteiseen tietorakenteeseen kirjoitetun tilatiedon ehtoa, jolloin alkuperäisen säikeen ajo käynnistyy. [3] [5]

Synkronoinnilla lukon tai semaforin avulla suojataan tyypillisesti jotakin resurssia. Semafori asettaa estetyt tehtävät jonoon FIFO-periaatteella, kun lukkoa käytettäessä estetyt tehtävät yrittävät toistuvasti saada lukon käyttöönsä. Semaforin avulla vältytään kiireinen odotustekniikan käytöltä esimerkiksi järjestelmässä, joissa on enemmän prosesseja kuin suorittimia. Synkronointiin osallistuvien tehtävien määrä voi vaihdella.

Ensimmäinen tehtävä ottaa haltuunsa lukon tai semaforin. Vain yksi tehtävä voi omistaa lukon tai semaforin kerrallaan. Muiden tehtävien tulee odottaa lukon tai semaforin vapautumista. [3] [5]

2.4 Vektorisointi

Vektorisointi on yleisesti tunnettu ohjelmakoodin suorituskykyä parantava prosessi, jossa silmukoissa yhtä arvoa kerrallaan suoritettavan koodin laskutoimitukset kootaan suorittimissa rinnakkain suoritettaviksi käskyiksi. Tämä voidaan suorittaa käyttämällä vektoreita. Vektori sisältää joukon elementtejä, kuten lukuja, totuusarvoja tai merkkijonoja. Vektoreita käytetään säilömään suuria tietomääriä, joiden avulla voidaan tallentaa useita toisiinsa liittyviä elementtejä. Vektoreille voidaan suorittaa monia erilaisia laskuoperaatioita, kuten yhteen- ja vähennyslasku, kerto- ja jakolasku. Vektorien väliset operaatiot suoritetaan elementeittäin. Kaikkien vektorin elementtien tulee olla saman tyyppisiä ja vektorien pitää olla elementeiltään saman mittaisia, jotta niihin voidaan suorittaa laskuoperaatioita. Vektorin indeksointi voidaan yksinkertaisimmillaan järjestää viittaamalla elementteihin niiden järjestysluvulla jonossa.

2.4.1 Profilointi ja vertailuanalyysi

Monet ohjelmakoodien kääntäjät vektorisoivat koodia automaattisesti osana niiden optimointiprosessia. Kääntäjien vektorisointiprosessi ei ole täydellinen, sillä kaikkea ohjelmakoodia ei voida koodin rakenteen takia vektorisoida tai sitä voi olla todella hankala vektorisoida. Lisäksi epätehokas keskusmuistin käyttö ja vähäinen välimuistin käyttö voi aiheuttaa negatiivisia vaikutuksia vektorisoinnin suorituskykyyn. Vektoritietoisella ohjelmoinnilla mahdollistetaan vektorisointi. Kehittäjän tulee ymmärtää mitkä koodin osat hyötyvät vektorisoinnista ja mikä estää vektorisoiduissa osioissa käskyjen mahdollisimman nopean suorittamisen. Ohjelman hitaiden osien (bottlenecks) tai eniten resursseja vaativien kohtien tunnistaminen (hotspots) on tärkein tehtävä koodin nopeuttamisessa. Tätä menetelmää kutsutaan profiloinniksi ja näitä kohtia voidaan tunnistaa profilointityökaluilla. Profilointityökalun avulla voidaan koodin suoritusaikaa ja suoritustehoa mitata valituista kohdista, tarkkailla muistin kulutusta ja selvittää siinä suoritettujen iteraatioiden määrät sekä osoittaa kohdat missä tehdään eniten töitä. Näiden metriikoiden avulla voidaan todentaa optimoinnin tehokkuus verraten

optimoinnin jälkeisiä metriikoita optimointia edeltäviin arvoihin. Kohta missä tehdään eniten töitä ei välttämättä yksinään ole kokonaisuuteen verrattuna ajankäytöllisesti suurin toimenpide. Kohdan kutsuminen lukuisissa iteraatioissa vaikuttaa suuresti ja tuolloin tuon kohdan vektorisointi voi vaikuttaa suorituskykyyn paljon. Yleisesti vektorisointi kannattaa suorittaa kohtiin, joista saadaan kokonaisuuden kannalta merkittävä hyöty. Lisäksi vektorisoinnin jälkeen, koodin optimointi on usein tarpeen. [4] [6] [7] [8]

Mikäli vektorisoitavaa koodia ei tunneta kunnolla, tuolloin kannattaa aloittaa sisäkkäisten silmukoiden tunnistamisella ja niiden iteraatioiden määrän laskemisella. Lisäksi silmukoille kannattaa suorittaa riippuvuus- (kappale 2.4.2) ja kompleksisuusanalyysi (kappale 2.4.3). Silmukoissa käytettävä koodissa ei saa olla syklisiä riippuvuuksia ja liian kompleksisella koodilla ei saavuteta tehokasta vektorisointia. Algoritmit ovat usein kompleksisia, mutta jo ehto- (if-else) ja valintarakenteiden (switch) käytöllä silmukan sisällä voidaan tuottaa koodia, jota ei voida vektorisoida. Yleisesti silmukoissa ei tulisi olla kompleksisien funktioiden kutsuja. Poikkeuksena sellaiset, jotka kääntäjä voi korvata sisäisillä vektorikäskyillä, kuten trigonometriset funktiot, neliöjuuri- ja neliöfunktiot. [4] [6] [7] [9]

2.4.2 Riippuvuusanalyysi

Ensimmäinen askel rinnakkaisen algoritmin kehittämisessä on hajottaa ongelma tehtäviksi, jotka voidaan suorittaa samanaikaisesti. Ongelma voidaan jakaa tehtäviksi monilla eri tavoilla ja tehtävien koot voivat vaihdella keskenään. Vektorisointi on yleisesti tunnettu keino nopeuttaa suuren datasetin silmukoita, joissa jokainen yksittäinen silmukan rungossa oleva käsky voidaan suorittaa kaikille iteraatioille samanaikaisesti ja silmukassa olevat lauseet voidaan suorittaa peräkkäin. Silmukan lausekkeille voidaan suorittaa riippuvuusanalyysi, jonka tavoitteena on saada toisista lausekkeista mahdollisimman riippumattomia lausekkeita. Tehtävien hajotus voidaan havainnollistaa suunnatun graafin avulla, jossa tehtäviä vastaavat solmut. Reunat osoittavat, että yhden tehtävän tulos tarvitaan seuraavan käsittelyyn.

2.4.2.1 Tietoriippuvuus

Tietoriippuvuusanalyysissä tarkastellaan riippuvuussuhteita käyttäen neljää ehtoa. **Lausekkeiden välillä ei ole riippuvuutta (1)**, kun jokainen silmukan iteraatio on itsenäinen ja iteraatiot voidaan suorittaa missä tahansa järjestyksessä ilman, että sillä on vaikutusta lopputulokseen. Tällainen tapaus voi olla lukeminen lukemisen jälkeen -operaatio, joka ei ole aito riippuvuus. Tällöin silmukka on rinnakkaistettavissa millä tahansa tavoin. [3, s. 98]

Lausekkeiden välillä on epäriippuvuus (2), kun myöhempi lauseke tuottaa arvon, joka hävittää aiemman lausekkeen käyttämän lähdeoperandin. Sama pätee myös toisin päin, kun aiempi lauseke tuottaa arvon, joka hävittää myöhemmän lausekkeen käyttämän lähdeoperandin. Tällöin puhutaan kirjoitus lukemiseen jälkeen -operaatiosta (write-after-read, anti-dependency). Tällainen ei ole yleensä turvallista muuttaa rinnakkaiseksi, mutta vektorisointi on turvallista, sillä iteraatiossa ylemmän indeksiarvon omaava iteraatio voi valmistua ennen pienemmän indeksiarvon omaavaa iteraatiota. Alla esimerkki lausekkeiden välisestä epäriippuvuudesta. [3, s. 98]

```
A = 1
B = A
C = B
```

Lausekkeiden välillä on suoritusriippuvuus (3), kun myöhempi lauseke käyttää aiemman lausekkeen tuottamaan arvoa (read-after-write, flow dependency). Käytännössä tämä ilmenee siten, että lausekkeiden suoritusjärjestys vaikuttaa lopputulokseen, jolloin vektorisointia ei voida suorittaa. Alla esimerkki lausekkeiden välisestä suoritusriippuvuudesta. [3, s. 98]

```
B = 1
A = B + 1
B = 3
```

Lausekkeiden välillä on ulostuloriippuvuus (4), kun myöhempi lauseke tuottaa arvon hävittäen aiemman lausekkeen käyttämän arvon. Käytännössä siis lausekkeet kirjoittavat samaan muistiosoitteeseen. Tällöin myös puhutaan kirjoitus kirjoittamisen jälkeen -operaatiosta (write-after-write, output dependency). Yleensä tällaista ei ole turvallista rinnakkaistaa. Antiriippuvuutta ulostuloriippuvuudella kutsutaan nimiriippuvuudeksi, jolloin riippuvuus voidaan poistaa uudelleennimeämällä muuttujat. Alla esimerkki nimiriippuvuudesta ja sen korjaaminen uudelleennimeämisellä. [3, s. 98]

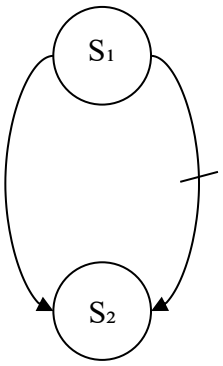
```
# Nimiriippuvuus
B = 1
A = B + 1
B = 3

# Uudelleennimetyt muuttujat
B2 = 1
A = B2 + 1
B = 3
```

Silmukan lausekkeiden riippuvuussuhteet tulee ymmärtää, sillä riippuvuussyklin omaavia silmukoita ei voida vektorisoida. Riippuvuussuhteita voidaan esittää suunnatulla graafilla.

```
Alkuperäinen silmukka:
For i = 1 To n:
   S1: a(i) = b(i+1) + c(i)
   S2: b(i) = a(i) + 1
EndFor

Vektoroitu silmukka:
S1: a(1:n) = b(2:n+1) + c(1:n)
S2: b(1:n) = a(1:n) + 1
```



Kuva 4. Silmukka, jossa on tietoriippuvuus ja epäriippuvuus lausekkeiden välillä

Silmukassa (Kuva 4) on tietoriippuvuus ja epäriippuvuus lauseiden S_1 ja S_2 välillä. Esimerkissä indeksointi aloitetaan luvulla 1. Lauseiden välillä ei ole sykliä, joten

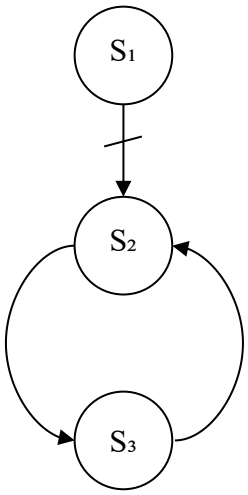
silmukka voidaan vektorisoida. Vektorisoidut lausekkeet voidaan suorittaa peräkkäin ja molemmat suorittavat operaation kaikille iteraatioille samanaikaisesti.

Alkuperäinen silmukka:

```
For i = 1 To n:
   S1: x(i) = a(i+1) + 1
   S2: a(i) = b(i-1) + c(i)
   S3: b(i) = a(i) + 1
EndFor
```

Osittain vektoroitu silmukka:

```
S1: x(1:n) = a(2:n+1) + 1
For i = 1 To n:
   S2: a(i) = b(i-1) + c(i)
   S3: b(i) = a(i) + 1
EndFor
```



Kuva 5. Silmukka, jossa on syklinen tietoriippuvuus lausekkeiden välillä

Silmukassa (Kuva 5) on syklinen tietoriippuvuus lausekkeiden S_2 ja S_3 välillä, sillä seuraava iteraatio käyttää edellisen laskemaa muuttujan b arvoa. Lause S_1 voidaan vektorisoida, sillä se on syklin ulkopuolella ja näin saadaan osittain vektorisoitu silmukka.

2.4.2.2 Hallintariippuvuus

Hallintariippuvuus ilmenee tilanteissa, joissa edellinen lauseke sallii seuraavan lausekkeen suorittamisen. Esimerkiksi lausekkeen S_2 suorittaminen on riippuvainen lausekkeen S_1 suorittamisesta. Lauseke S_2 suoritetaan vain, kun lausekkeen S_1 ehtolauseke on tosi. Hallintariippuvuus silmukan sisällä tuottaa tilanteen, jolloin silmukkaa ei voida vektorisoida ilman hallintariippuvuuden purkamista. [6] [7] [9]

```
S1: if (a == b)
S2:     a = a + b
S3: b = a + b
```

2.4.2.3 Resurssiriippuvuus

Resurssiriippuvuus esiintyy tapauksissa, joissa kaksi tehtävää käyttää samaa jaettua resurssia esimerkiksi tiedostoa, muistialuetta tai siirräntäresurssia. Resurssiriippuvuus voi esiintyä myös, kun tehtävien suhteet riippuvat tiedosta, johon viitataan epäsuorasti esimerkiksi a[c[i]], jossa c[i] tuottaa virheellisen indeksiviittauksen taulukoon a. Tällaista tilannetta kutsutaan myös termillä tuntematon riippuvuus. Tämän tyyppistä koodia ilmenee silmukoissa, jossa indeksiviittaus muuttuu jokaisessa iteraatiossa. Laskentaa suorittavat ohjelmat rakentuvat usein silmukoihin. Tämän takia on tärkeää kiinnittää huomiota silmukoiden suoritustehokkuuteen rinnakkaistamalla ne. Bernsteinin kolme ehtoa määrittävät milloin kaksi prosessia P_1 ja P_2 voidaan suorittaa turvallisesti rinnakkain. Ensimmäisessä ehdossa todetaan, että prosessi P_1 ei kirjoita paikkaan, jota prosessi P_2 lukee. Toinen ehto täydentää ensimmäistä siten, että prosessi P_2 ei kirjoita paikkaan, jota prosessi P_1 lukee. Kolmas määrittää kirjoittamisesta, että prosessi P_1 ei kirjoita samaan paikkaan, johon prosessi P_2 kirjoittaa. Nämä ehdot muodostavat perustan kaikille silmukan rinnakkaisalgoritmeille. Ehdot ovat hyvin rajoittavat, mikäli halutaan jakaa tietoa prosessien välillä. Tällöin käsittelyvuorot voidaan järjestää synkronoinnilla (kappale 2.3.5). [6] [7] [9]

2.4.3 Kompleksisuusanalyysi

Kompleksisuusanalyysissa arvioidaan koodissa käytetyn algoritmin tehokkuus ajan ja muistin käytön suhteen tuottaen abstrakti mittaustulos. Sen avulla voidaan arvioida algoritmin suorituskykyä ilman matemaattisia kaavoja ja sen perusteella pystytään valitsemaan tehtävään suorituskyvyltään riittävä algoritmi. Tehokkuutta kuvataan O-notaatiolla. Analyysin aikana selvitetään algoritmissa suoritettavien operaatioiden maksimimäärät esimerkiksi n ja m. O-notaatiosta jätetään lopulta vakiomuotoiset määrät pois. Oheisten kompleksisuusanalyysiesimerkkien katsotaan olevan melko tehokkaita.

// Luokkaa $O(1)$ laskentaoperaatiota

```
double y = x + 1.0;
```

// Luokkaa $O(2N) = O(N)$ laskentaoperaatiota

```
for(int i = 0; i < N; i++)
  y[i] = 2.0 * x[i] + 1.0;
```

// Luokkaa $O(N(1 + 2M)) = O(NM)$ laskentaoperaatiota

```
for(int i = 0; i < N; i++) {
  x[i] = x[i] + 3.0;
  for(int j = 0; j < M; j++)
    y[i] = A[i][j] * x[i] + 1.0;
}
```

// Luokkaa $O(N^2)$ laskentaoperaatiota

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

// Luokkaa $O(\log N)$ laskentaoperaatiota

```
for(int i = 1; i <= N; i = i * 2) {
  item[i] += 1;
}
```

2.5 Python ohjelmointikieli

Python on monipuolinen tulkattava olio-ohjelmointikieli, joka on moneen muuhun ohjelmointikieleen verrattuna selkeä ja tehokas. Kieli on saanut vaikutteita Perl-kielestä. Python-kielen käyttö on yleistynyt viime vuosina aikana data-analytiikan ja verkkopalveluiden ohjelmointikielenä. Suosio perustuu pitkälti sen käytön helppouteen, kielen joustavuuteen ja siihen saa paljon laajennuksia sekä kirjastoja. [10]

Silmukat voivat aiheuttaa Python-koodissa pullonkauloja, sillä silmukat, varsinkin sisäkkäiset silmukat, ovat hitaampia kuin C/C++ -silmukat. Tämä johtuu siitä, että Python on tulkittava kieli eli Python-kielinen ohjelma suoritetaan Python-tulkin avulla, joka ensin lukee Python-kielisen koodin ja kääntää koodin tavukoodiksi ennen sen ajoa. Oletetaan, että koodi sisältään osan, jossa iteroidaan taulukkoa. Python on dynaamisesti tyypittävä, mikä tarkoittaa, että sillä ei ole tietoa minkä tyyppisiä objekteja taulukossa on esim. kokonaisluku tai merkkijono. Tuo tieto on tallennettu taulukon alkioissa oleviin objekteihin ja Python tietää objektin tyypin vasta kun taulukon alkio otetaan käsittelyyn. Pythonissa tyyppi on siis olion ominaisuus, ei olioon viittaavan nimen ominaisuus. Python tekee jokaisessa operaatiossa tarkastuksia, kuten muuttujan tyyppitarkastuksen ja näkyvyysalueen selvittäminen ja virheellisten toimintojen tarkastamisen. C/C++ -kielessä on käytössä staattinen tyypitys eli taulukon sallitaan koostuvan vain yhdestä kääntäjän ennalta tietämästä tietotyypistä, joka mahdollistaa kääntäjän koodin optimoinnin. [10]

Python ei sisällä oletuksena toiminnallista syntaksia numeerisen datan analysointiin ja muuntamiseen, kuten R- tai Julia-ohjelmointikielet, vaan Python tarvitsee erikoiskirjastoja niiden suorittamiseen. Esimerkiksi NumPy-kirjastopaketin avulla Python-ekosysteemiin saadaan korkean suorituskyvyn tieteellisen laskennan ja analysoinnin toiminnallisuudet. Erilaisilla Python-kirjastojen yhdistelmillä voidaan korvata muilla ohjelmointikielillä tehtyjä toteutuksia, kuten Matlab-koodia voidaan korvata NumPy-, SciPy- ja MatploLib-yhdistelmällä. NumPy on perusta monille muille korkeamman tason Python-työkaluille, kuten Pandas ja scikit-learn. Jopa TensorFlow käyttää NumPy-taulukoita listoihin, vektoreihin ja matriiseihin kohdistuvissa lineaarialgebran operaatioissa Tensor-objektinsa ja tietokulkukaavionsa perustoteutuksissa. Useat NumPy-operaatiot ovat toteutettu C-kielellä, jolloin vältetään Python-silmukoinnin aiheuttamilta ongelmilta ja näin saavutetaan nykyaikaisten

analysointi ja koneoppimisen tehtävien vaatimat nopeusvaatimukset. Python-koodissa käytetyt NumPy-kirjaston kutsut ohjataan esikäännetylle C-koodille, joka takaa tehokkaan toiminnon, kuten taulukoiden käsittelyyn. NumPy sallii taulukoissa vain yhden tietotyypin käytön, joten taulukon elementit pakotetaan samanlaisiksi. NumPy tarjoaa myös C-API-rajapinnan, jonka avulla voi koodista tehdä entistä nopeampaa. Sen käyttö tosin vaatii syvällistä C-kielen tuntemusta ja tekee toteuttamisesta kompleksisempaa verrattuna Python-kielen käytön helppouteen. [11]

2.5.1 Prosessien ja säikeiden luonti

Prosessien ja säikeiden luonti voidaan suorittaa Python-ohjelmissa useilla eri tavoilla esimerkiksi käyttämällä threading-, multiprocessing- tai concurrent.futures-moduulien tarjoamia luokkia. Nämä moduulit tarjoavat korkeantason rajapinnan kutsuttavien toimintojen asynkroniseen suorittamiseen käyttäen joko säikeitä tai prosesseja. Moduulien tarjoamien luokkien käyttöönotto perustuu niiden avulla suoritettavien tehtävien mukaan ja käytössä olevien suoritinytimien määrän mukaan. Yleisenä ohjeistuksena pidetään, että säikeitä käytetään siirräntäoperaatioita sisältäviin tehtäviin ja prosesseja suoritinsidonnaisuutta vaativiin tehtäviin. Esimerkiksi verkkokutsuja suorittava tehtävä on siirräntäoperaatioita sisältävä tehtävä ja matemaattista laskentaa suorittava tehtävä on suoritinsidonnaisuutta sisältävä tehtävä. Prosesseja käytettäessä tulee suorittimessa olla useita ytimiä.

Tehtävä voidaan suorittaa uudessa säikeessä käyttäen threading-moduulin Thread-luokkaa antamalla suoritettava tehtävä target-argumenttina (seuraava esimerkki). Säie luodaan käyttäen haarautumat ja liittymät mallia (kappale 2.3.4.4) käynnistäen haarautuma start-funktiolla ja join-funktion kutsulla liittymä odottaa haaroitettujen säikeiden loppuun suorittamista. Esimerkissä pääsäikeestä käynnistetään toisessa säikeessä suoritettava tehtävä, joka tulostaa tervehdysviestin viiveen jälkeen. Pääsäikeessä oleva odotusviesti tulostuu haarautuman luonnin jälkeen ensin ja tervehdysviesti tulostuu vasta liittymäfunktion kutsun ja tehtävässä olevan viiveen jälkeen. [12]

```
from time import sleep
from threading import Thread

def task():
  sleep(1)
  print('Tervehdys toiseesta säikeestä')

thread = Thread(target=task)
thread.start()
…
print('Odottaa säikeen valmistumista')
…
thread.join()
```

Tehtävä voidaan suorittaa uudessa prosessissa käyttäen multiprocessing-moduulin Process-luokkaa. Prosessi luodaan Process-luokalla samoin kuin säikeet Thread-luokan avulla antamalla tehtävä target-argumenttina käynnistäen haarautuma start-funktiolla ja join-funktion kutsulla liittymä odottaa haaroitettujen prosessien loppuun suorittamista. Prosessien ohjelmoinnissa käyttäen multiprocessing-moduulia on tiettyjä sääntöjä, joita täytyy noudattaa välttyäkseen ongelmilta. Esimerkiksi suurten tietomäärien välittämistä prosessien välillä tulee välttää. Prosessien välisessä kommunikaatiossa suositetaan viestinvälitysmallia (kappale 2.3.5) jono tai liukuhihnamallia käyttäen matalamman tason synkronisointitapojen sijaan, kuten lukot ja semaforit. [13]

Thread- ja Process-luokan käyttö on perusteltavissa, kun suoritettavia tehtäviä on vähän. Lukuisien tehtävien suorittaminen näiden luokkien avulla muuttuu hankalaksi ja jokainen luotava ja tuhottava säie vaatii resursseja, kuten muistia ja laskentakapasiteettia. Suorituskyvyllisesti järkevämpää on luotujen säikeiden ja prosessien uudelleenkäyttö. Tämä voidaan toteuttaa säilömällä luotuja säikeitä tai prosesseja käyttäen allasmallia. [12] [13]

Python 3.2 -versioon lisätty concurrent.futures-moduuli tarjoaa korkeantason rajapinnan kutsuttavien toimintojen asynkroniseen suorittamiseen käyttäen joko säikeitä ThreadPoolExecutor-luokan tai prosesseja ProcessPoolExecutor-luokan avulla. Molemmat luokat toteuttavat abstraktin Executor-luokan rajapinnan. Molemmat luokat

käyttävät allasmallia tehtävien hallintaan (kappale 2.3.4.8). Tehtävien vastauksien hallinnassa käytetään Future-luokkaa, joka käärii asynkronisen kutsun vastauksen. Käytännössä se on lupaus siitä, että tehtävän suoritus loppuu myöhemmin. Future-luokka mahdollistaa tehtävän palauttaman vastauksen vastaanottamisen lisäksi muun muassa tehtävän tilan tarkastelun, tehtävän aikana tapahtuneen poikkeuksen hallinnan, takaisinkutsufunktion (callback) asettamisen ja tehtävän suorituksen peruuttamisen. Sen avulla voidaan synkronoida säikeet tai prosessit ja estää koodin suorituksen eteneminen, kunnes tehtävä on suoritettu tai sen vastaus on saatavilla. [14]

Executor-luokka sisältää submit-, map- ja shutdown-funktiot. Executor-luokan submit-funktion kutsu käynnistää tehtävän suorittamisen ja palauttaa Future-luokan välittömästi. Tehtävän vastaus saadaan tehtävän suorittamisen jälkeen result-funktion avulla. Executor-luokan map-funktion avulla suoritettava tehtävä voidaan käynnistää asynkronisesti jokaiselle elementille iteroitavassa objektissa, kuten lista. Asynkronisissa silmukoissa suositellaan käytettäväksi map-funktiota. ProcessPoolExecutor-luokkaa käytettäessä map-funktio kykenee palastelemaan iteroitavat elementit eriksi yksittäisten elementtien sijaan käyttäen chunksize-argumenttia. Parametrin oletuskoko on yksi, joten se on syytä asettaa suurta eräkokoa suoritettaessa. Pitkien iteraatioiden suorituskyky kasvaa huomattavasti, kun käytetään suurta eräkokoa. Executor-luokan shutdown-funktion kutsu vapauttaa Executor-luokan käyttämät resurssit. Funktion wait- ja cancel_future-argumentit määrittävät odotetaanko Future-luokkien valmistumista tai peruutetaanko luodut Future-luokat. [14]

Executor-luokan kanssa suositellaan käytettäväksi kontekstinhallintaa (context manager), joka toteutetaan with-lausekkeen avulla muodostaen koodin etenemiselle esteen tehtävien käynnistämisen ja niiden vastauksien keräämistä varten. Kontekstinhallinta huolehtii varattujen resurssien poistamisesta lausekkeen päädyttyä syystä tai toisesta kutsuen sisäisesti shutdown-funktiota. Esimerkissä käynnistetään yksi säie suorittamaan laskutoimitusta käyttäen submit-funktiota.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 123, 4567)
    print(future.result())
```

Laskutoimituksen vastaus saadaan tehtävän valmistuttua future-luokan result-funktiota käyttäen. Maksimityösäikeiden määrän oletusarvo määräytyy ThreadPoolExecutor-luokassa seuraavan algoritmin mukaan:

```
max_workers = min(32, os.cpu_count() + 4)
```

ja ProcessPoolExecutor-luokassa oletusarvona on suorittimien määrä. [14]

2.5.2 Vektorisointi

NumPy-kirjastopaketin avulla silmukoiden vektorisointi Python-koodissa nopeuttaa koodia yleensä huomattavasti. Oheinen esimerkki vektorisoi taulukon summan laskennan. Tuloksista havaitaan, että NumPy-kirjastopaketin avulla vektorisoitu laskenta on yli 60 kertaa nopeampi. Tämä johtuu siitä, että vektorisoinnissa käytettiin taulukon summan laskentaan NumPy-kirjastopaketin tarjoamaa sum-funktiota, joka on kirjoitettu ja esikäännettyä C-kieltä. Korkean tason kielten, kuten Python, Matlab ja R, vektorisointi perustuu matalamman tason kielellä kirjoitetun ja esikäännetyn koodin käyttöön. Tämän takia vektorisointi toimenpiteenä ei ole triviaali, jolla saavutettaisiin aina optimi tulos.

```
import numpy as np
from timeit import timeit

def sum():
    total = 0
    for i in np.arange(10000):
        total += i
    return total

def sum_n():
    return np.sum(np.arange(10000))

%timeit sum()
845 µs ± 9.91 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)

%timeit sum_n()
13.7 µs ± 105 ns per loop(mean ± std. dev. of 7 runs, 100000 loops
each)
```

Lisäksi itse ongelman vektorisointi voi olla vaikeampaa kuin yleensä koodin vektorisointi. Tämä tarkoittaa yleensä sitä, että joudutaan käyttämään uutta algoritmia tai keksimään uusi vanhan tilalle. Esimerkkinä vektorisoimaton alpha2u-funktio, joka kartoittaa ympyrän kehän pituuden (alpha) neliön kehän pituuteen (u) ja kartoituksen siirtymän (scale) yksikköneliöllä. Funktiossa on peräkkäin suoritettavia matemaattisia lausekkeita ja ehtolausekkeita, jotka muodostavat tietoriippuvuuden (kappale 2.4.2.1). Lausekkeiden välillä on epäriippuvuus ja lausekkeiden välillä ei ole sykliä. Vektorisoidut lausekkeet voidaan suorittaa peräkkäin. Täysin manuaalinen vektorisointi voi olla todella hankala.

```
import numpy as np

def alpha2u(alpha):
    c = np.cos(alpha)
    s = np.sin(alpha)
    scale = 1.0 / np.maximum(abs(c), abs(s))
    x = c * scale
    y = s * scale
    eps = 1.0e-8
    if abs(x - 1.0) < eps and 0 <= y:
        u = y
    elif abs(y - 1.0) < eps:
        u = 2 - x
    # x == -1
    elif abs(x + 1.0) < eps:
        u = 4 - y
    # y == -1
    elif abs(y + 1.0) < eps:
        u = 6 + x
    else:
        u = 8 + y
    return u, scale
```

Muuttamatta Python-koodia sen voi kääriä näyttämään vektorisoidulta käyttämällä NumPy-kirjastopaketin vectorize-metodia. Vektorisointitoiminto on ensisijaisesti tarkoitettu mukavuuden vuoksi. Toteutus on pohjimmiltaan for-silmukka, joten sillä ei

välttämättä saavuteta parempaa suorituskykyä. Peruskäyttö on hyvin yksikertainen. Vectorize-funktion parametriksi annetaan vektorisoitavan funktion referenssi. Tämän lisäksi voidaan määrittää ulostulotiedon tyyppi merkkijonona joko merkkijonona tai tietotyyppimääritteiden luettelona. Jokaiselle ulostulolle tulee olla yksi tietotyyppimääritelmä. Tässä tapauksessa ulostulotietotyypiksi määriteltiin kaksi liukuluku NumPy-taulukkoa sisältävä tuple-luokka. Näin tietotyyppiä ei selvitetä ensimmäistä kutsua suoritettaessa. [11]

```
types = (
      type(np.array([], dtype='f')),
      type(np.array([], dtype='f'))
      )
alpha2u_v = np.vectorize(alpha2u, otypes=types)
```

Generoidaan arvot (values), joita käytetään testattaessa vektorisoimatonta ja vektorisoitua funktiota. Aloitetaan ensin alkuperäisestä funktiosta, joka kestää noin 61 mikrosekuntia iteraatiota kohti.

```
values = np.arange(0, 180, 15) * np.pi/180
%%timeit
results1 = []
for i in values:
      results1.append(alpha2u(i))
```

```
61.1 µs ± 255 ns per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
```

Sen jälkeen mitataan vektorisoitu funktio, joka kestää noin 58 mikrosekuntia iteraatiota kohti.

```
%%timeit
results2 = alpha2u_v(values)
```

```
58.2 µs ± 392 ns per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
```

Ero suorituskyvyssä on hyvin pieni, joten tätä vektorisointitapaa voidaan pitää lähinnä koodin selkeyttäjänä. Koodi tosin on vektorisoituna paljon selkeämpää. Ennen kuin vektorointia aloitetaan suorittamaan manuaalisesti, on järkevää selvittää koodista pullon kaulat ja suorittaa optimointia pullonkauloihin, mikäli se on mahdollista ja järkevää.

2.5.3 Profilointi ja vertailuanalyysi

Python ohjelmia voidaan profiloida deterministisesti käyttäen cProfile- ja profile-toteutuksia käyttäen. Näistä kahdesta cProfile on C-kielellä toteutettu ohjelmistolaajennus, joka on suorituskyvyllisesti suositeltavaa käyttää pidempi kestoisten ohjelmien profiloinnissa. Mikäli on tarvetta laajentaa profiloijaa, silloin tehtävä saattaa olla helpompi käyttäen Python-pohjaista profile-toteutusta, mutta sen käyttö on luonnollisesti hitaampaa kuin cProfile-toteutuksen käyttö. Profilointi cProfile-toteutuksella on yksinkertaista. Esimerkissä profilointi käynnistetään main-funktiolle tulostaen lopulta profiloinnin tulokset standardiulostuloon (stdout). Profilointi voidaan käynnistää monilla muillakin tavoilla, mutta oheinen on yksinkertaisin tapa Python-koodina. [8]

```
>>> import cProfile
>>> cProfile.run('main()')

197 function calls (192 primitive calls) in 0.002 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

Tuloksissa ensimmäinen rivi kertoo, että 197 kutsua (function calls) valvottiin. Näistä kutsuista 192 oli suoria kutsuja (primitive calls). Tuloksissa näkyy sarakkeina tehtyjen kutsujen määrä (ncalls), kutsutussa funktiossa vietetty aika (tottime), kahden edellisen sarakkeen osamäärä (percall=tottime/ncalls), vietetty kumulatiivinen aika funktioissa ja alifunktioissa (cumtime), edellisen sarakkeen ja suorien kutsujen määrän osamäärä (percall=cumtime/primitive calls) ja kohteena olleen funktion tiedosto sekä rivinumero (filename:lineno(function)). Kun ensimmäisessä sarakkeessa (ncalls) on kaksi numeroa, tarkoittaa se rekursiivista kutsua. Toinen arvo on primitiivisten kutsujen määrä ja ensimmäinen kaikkien kutsujen määrä. [8]

Python-koodin profilointi voidaan käynnistää myös skriptinä kohdistaen se toiseen tiedostoon esimerkiksi seuraavasti. Ensimmäinen parametri m määrittää profiloinnissa käytettävän toteutuksen, arvoina joko cProfile tai profile. Parametri o määrittää profilointituloksien tallennustiedoston standardiulostulon sijaan. Lopuksi kerrotaan profiloinnin kohde, joka esimerkissä on run.py-tiedosto. [8]

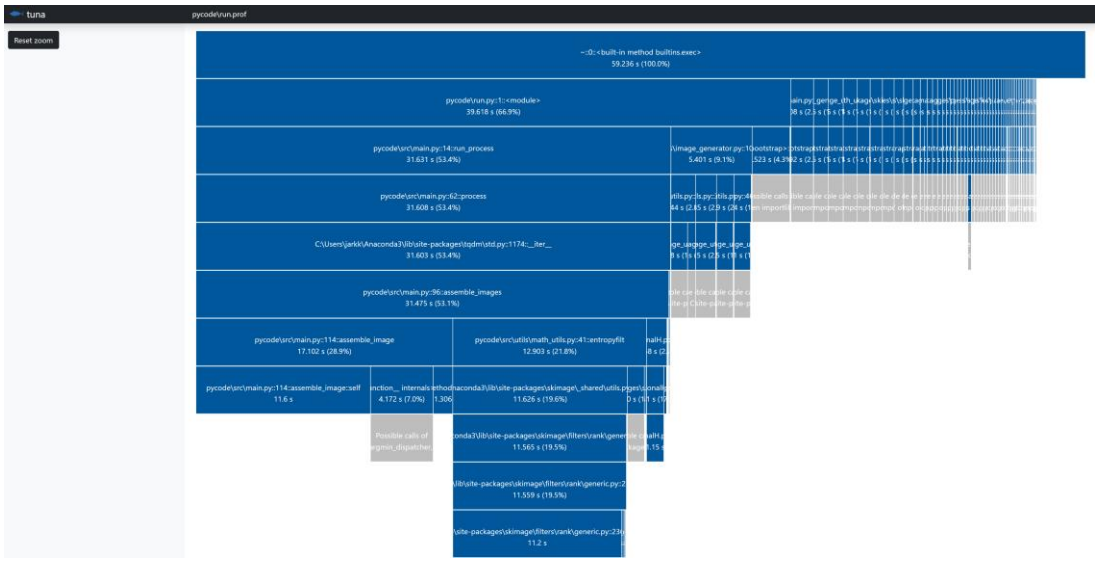
```
> python -m cProfile -o pycode\run.prof pycode\run.py
```

Profiloinnin tuloksia voidaan käsitellä useilla eri tavoilla. Python tarjoaa Stats-luokan, jonka avulla profilointituloksien tallennustiedosto voidaan lukea ja luettua tietoa voidaan käsitellä halutulla tavalla. Profilointituloksien esitystapaa ja tulostusta voidaan hallita luokan avulla ohjelmallisesti. [8]

Profilointitulokset on aiemmissa esimerkeissä tulostettu standardiulostuloon, mutta visualisointi voidaan järjestää graafisesti. Tällaisia työkaluja ovat mm. Tuna ja Snakeviz. Molempien työkalujen käyttö on samankaltainen. Työkalut asennetaan käyttäen pip-pakettien hallinnan avulla. Profilointituloksen esittäminen käynnistetään kutsumalla komentorivillä työkalua antamalla parametriksi profilointituloksien tallennustiedoston polku. [15] [16]

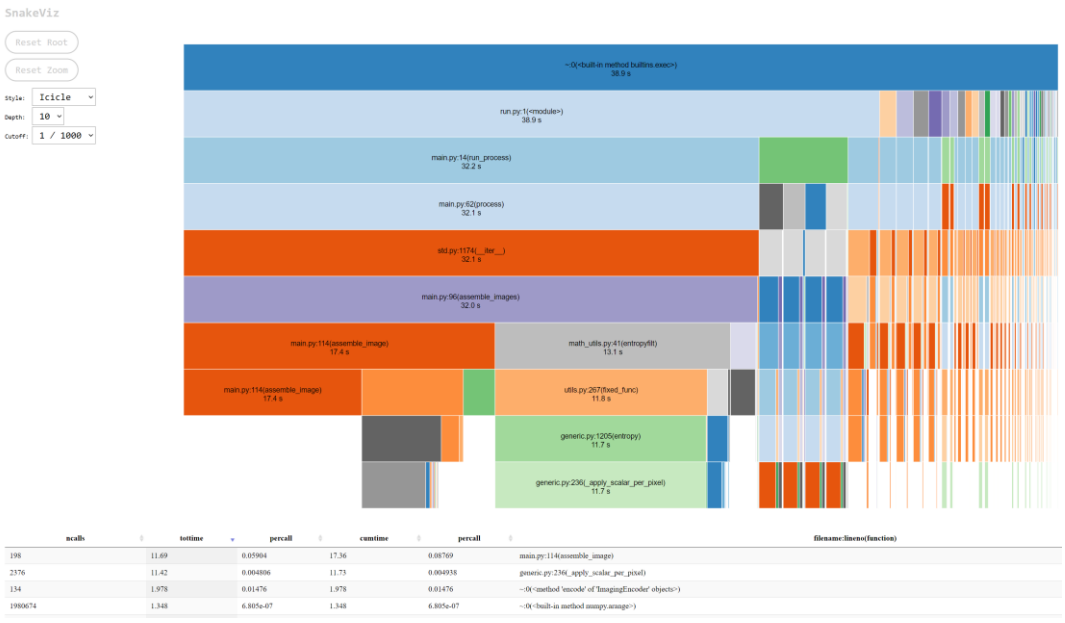
```
> tuna pycode\run.prof
> snakeviz pycode\run.prof
```

Tämän jälkeen visualisointi avattavissa selaimessa Tuna-työkalulla osoitteesta http://localhost:8000/ tai Snakeviz-työkalulla http://127.0.0.1:8080/snakeviz/-alkuisesta osoitteesta. Nämä kaksi työkalua ovat hyvin samankaltaiset, joskin Snakeviz-työkalussa (Kuva 6) on cProfile-työkalusta tuttu taulukko mukana ja visualisointiin pääsee vaikuttamaan interaktiivisesti enemmän kuin Tuna-työkalussa (Kuva 7).



Kuva 6. Profilointituloksien visualisointi Tuna-työkalulla

Molemmissa työkaluissa voi kaivautua kutsuissa syvemmälle klikkaamalla kutsuttua funktiota. Funktiokutsussa näkyy siellä käytetty aika ja missä tiedostossa millä rivillä funktio sijaitsee.



Kuva 7. Profilointituloksien visualisointi SnakeViz-työkalulla

Vertailuanalyysia Python-koodissa kannattaa suorittaa tehokkaan timeit-moduulin avulla ilman riippuvuuksia ulkopuolisiin kirjastoihin. Ohessa esimerkki sen yksinkertaisesta peruskäytöstä Python-koodin joukossa ja sen tuloksen standardiulostuloon tulostettuna. Timeit-moduulin avulla voi myös helposti tehdä oman dekoraattorin (decorator), jolloin usein toistuvaa ajastimen käyttöä voidaan yksinkertaistaa. [17]

```
import timeit
def functionA():
   print("Function A called")

def functionB():
   print("Function B called")

start_time = timeit.default_timer()
functionA()
print(timeit.default_timer() - start_time)

start_time = timeit.default_timer()
functionB()
print(timeit.default_timer() - start_time)


> Function A called
  6.419999408535659e-05
  Function B called
  4.169999738223851e-05
```

2.5.4 Indeksointi, viipalointi ja ikkunointi

Python-koodissa yksi tai moniulotteisien listojen alkioihin viitataan 0-pohjaisella indeksoinnilla (indexing). Tämä tarkoittaa sitä, että listan ensimmäisen alkion indeksi on 0 ja seuraavan indeksi on 1. Lisäksi on mahdollista käyttää negatiivisia indeksejä, joiden laskenta aloitetaan listan lopusta alkaen arvosta -1 (Taulukko 1). Listan alkioihin viittaaminen tai viipalointi (slicing) järjestetään siten, että haluttu indeksointi sijoitetaan listan muuttujan perään hakasulkuoperaattorin sisään. [10]

Taulukko 1. Listan indeksointi Pythonissa

Listan sisältö	**P**	**y**	**t**	**h**	**o**	**n**
Positiivinen indeksointi	0	1	2	3	4	5
Negatiivinen indeksointi	-6	-5	-4	-3	-2	-1

Listoja voidaan viipaloida aloitus- ja lopetusindeksin avulla. Viipalointi ei sisällä lopetusindeksiä, vaan laskee listaa siihen asti. Viipaloinnissa käytetään lisäksi kaksoispistenotaatiota erottamaan aloitus- ja lopetusindeksit sekä askeleen määrä. Viipaloinnin oletusarvoina aloitusindeksille on 0, lopetusindeksille listan pituus ja askeleen määränä 1. Jos aloitus- tai lopetusindeksiä ei viipaloinnissa anneta, käytetään oletusindeksiä ja viipalointi palauttaa oletusarvojen mukaisen tuloksen. NumPy-taulukoita voi indeksoida kuten Python-listoja hakasulkuoperaattorilla. [10]

```
>>> a = np.array([1, 2, 3, 5, 8, 13])
>>> a[0]
1
>>> a[-1]
13
>>> a[0:2]
array([1, 2])
>>> a[:2] # Sama kuin a[0:2], jolloin viipalointi aloitetaan alusta
array([1, 2])
>>> a[2:]
array([ 3,  5,  8, 13])
```

NumPy-taulukoita voidaan indeksoida myös bool- ja int-tyyppisillä taulukoilla. Bool-tyyppinen taulukko sisältää totuusarvoja, ja int-tyyppinen taulukko vastaavasti kokonaislukuja. Bool-tyyppisiä taulukoita käytetään poimimaan tietoja ehtojen mukaan käyttäen vertailuoperaattoreita. Vertailuoperaattoria käyttäen taulukosta poimitaan arvo, jonka kohdalla ehto on tosi ja muut jätetään poimimatta. Vertailuehtoja voidaan yhdistää bittioperaattoreilla (bitwise operator). [10]

```
>>> arr = np.array([100, 101, 102, 103, 104, 105, 106])
>>> inds = [0, 2, 4, 6]
>>>
>>> arr[inds] # Poimi indeksit 0, 2, 4, 6
array([100, 102, 104, 106])
>>>
>>> # Onko alkio suurempi kuin 102
>>> arr > 102
array([False, False, False,  True,  True,  True,  True])
>>>
>>> # Poimi alkiot, jotka suurempia kuin 102 ja pienempiä kuin 105
>>> arr[(arr > 102) & (arr < 105)]
array([103, 104, 105, 106])
```

Näiden lisäksi NumPy-kirjasto tarjoaa lukuisia tehokkaita indeksointiin, viipalointiin ja ikkunointiin tarkoitettuja toimintoja, kuten tiedon indeksointi-, vertailu-, lisäämis- ja iterointioperaatioita. [18]

```
>>> arr = np.array([0, 1, 2, 3])
>>> arr.nonzero()
(array([1, 2, 3], dtype=int64),)
>>> np.where(arr > 1)
(array([2, 3], dtype=int64),)
>>> arr2 = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(arr2):
...     print(index, x)
>>>
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

NumPy tarjoaa muutaman ikkunointifunktion, jolla moniulotteisesta taulukosta voi valita halutun kokoisen ikkunan erilaisin siirtymin. Tällaisia ovat esimerkiksi numpy.lib.stride_tricks-moduulin tarjoamat sliding_window_view- ja as_strided-funktiot. Näistä sliding_window_view-funktio luo annetusta taulukosta liukuvan ikkunanäkymän annetulla ikkunamuodolla. Tämä tunnetaan myös rullaavana tai liikkuvana ikkunana, jossa ikkuna liukuu taulukon kaikkien taulukon ulottuvuuksien poikki ja poimii taulukon osajoukot kaikista ikkunan kohdista. Toinen as_strided-funktio luo näkymän taulukkoon määritellyllä askeleella ja muodolla. Tämän funktion käyttöä ei suositella, sillä se voi aiheuttaa virheellisen käytön seurauksena virheellisiä muistipaikan osoittamisia, joka voi vioittaa tuloksia ja kaataa ohjelman. Virheellistä kirjoittamista voidaan estää writeable-parametrilla arvolla False, jolloin palautettava taulukko on vain luettavissa. [18]

```
>>> x = np.arange(6)
>>> x.shape
(6,)
>>> v = sliding_window_view(x, 3)
>>> v.shape
(4, 3)
>>> v
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

Ikkunointifunktioita tarjotaan myös muissa kirjastoissa. Kuvankäsittelyyn erikoistunut scikit-image-kirjasto tarjoaa algoritmien joukossa skimage.util-moduulissa view_as_windows- ja view_as_blocks-funktiot. Näistä view_as_windows-funktio luo rullaavan ikkunanäkymän syötetystä n-ulotteisesta taulukosta. Ikkunat ovat päällekkäisiä näkymiä syötetystä taulukosta ja vierekkäisiä ikkunoita on siirretty määritellyllä indeksimäärällä. Rullaavia näkymiä käytettäessä tulee olla varovainen. Mitä suurempi sisään syötetty taulukko on, sitä enemmän muistia kulutetaan. Rullaavien näkymien luonti on helppoa. Esimerkissä määritellään sisään syötettävä yksiulotteinen taulukko ja kolmen alkion levyinen ikkunakoko. [19]

```
>>> A = np.arange(10)
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> window_shape = (3,)
>>> B = view_as_windows(A, window_shape)
>>> B.shape
(8, 3)
>>> B
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6],
       [5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

Toinen view_as_block-funktio luo määritellyn kokoisia vierekkäisiä näkymiä sisään syötetystä taulukosta. Määritelty näkymäkoko tulee olla jaollinen sisään tulevan taulukon ulottuvuuden kanssa. Toiminnot siis poikkeavat hieman toisistaan. [19]

```
>>> import numpy as np
>>> from skimage.util.shape import view_as_blocks
>>> A = np.arange(4*4).reshape(4,4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> B = view_as_blocks(A, block_shape=(2, 2))
>>> B[0, 0]
array([[0, 1],
       [4, 5]])
>>> B[0, 1]
array([[2, 3],
       [6, 7]])
```

3 TOTEUTUS

Toteutusosiossa käydään läpi konseptitoteutuksen ohjelmointikielen vaihtoprosessia esittäen samalla käytännön koodifragmenttien esimerkkejä ja toteutuksen aikana ilmenneitä haasteita. Yksikkötestauksen avulla ohjataan kehittäminen laadullisesti parempaan suuntaan, joiden avulla toteutussyklin aikana saadaan välitön vaste muokatun koodin oikeellisuudesta ja toisaalta muiden osien muuttumattomuudesta. Ohjelmakoodin optimointi vaatii ohjelmiston profiloinnin ja riippuvuusanalyysin. Toteutusosiossa kuvataan konseptitoteutukseen suoritettuja optimointitoimia ja sen eri vaiheita reflektoiden havaittuja toteutuksen haasteita vasten taustatutkimusosuuden teoreettiseen tarkasteluun.

3.1 Ohjelmointikielen vaihto

Ohjelmointikielen vaihto aloitettiin suoralla Matlab–Python konversiolla, jossa jokainen Matlab-koodin lauseke muutettiin rivi riviltä Python-muotoon. Tällä pyrittiin saamaan toteutus mahdollisimman nopeasti samanlaiseksi kuin alkuperäinen, ilman optimointia tai koodin uudelleenkirjoittamista. Erilaisia laskenta-, sijoitus-, indeksointi- ja siivutusoperaatioita Matlab-koodissa oli yhteensä noin 140, kun Python-koodiin niitä päätyi lopulta noin 129. Haastavimmat muutokset olivat vektorien käsittelyoperaatiot, trigonometristen funktioiden konversiot ja kuvien luontiin liittyvät osuudet. Lisäksi entropia- ja histogrammitoteutuksissa oli haasteita, sillä Python-kirjastot käyttäytyivät hieman eri tavoin. Esimerkiksi NumPy-kirjaston histogram-funktio palauttaa laatikkojen oikean puoleiset reunapisteet, kun Matlab hist -funktio[1] palauttaa laatikkojen keskipisteet [20] [21]. Tämä saatiin lopulta helposti toteutettua käyttäen Python-koodissa skimage.exposure-moduulin histogram-funktiota, joka toimii samoin kuin Matlab-toteutus. [22]

Koodin muunnoksessa tapahtui edellä mainituissa kohdissa eniten virheitä. Virheet johtuivat pääsääntöisesti huolimattomuudesta, kuten liian suoraviivaisesta kahden kielen välisestä käännöksestä perustuen oletuksiin eikä tarkasti dokumentaatiota noudattaen, virheelliseen indeksointiin liittyen tai väärinymmärrettyihin matemaattisiin operaatioihin.

[1] Dokumentaation mukaan hist-funktion sijaan suositellaan käytettävän histogram-funktiota.

Virheet havaittiin, kun lopputulos ei ollut sama kuin alkuperäisessä koodissa tai ohjelman suoritus päätyi virheeseen. Virheiden silmämääräinen etsiminen oli työlästä ja ei tuottanut haluttuun lopputulokseen. Tässä vaiheessa Python-koodin lohkoja ryhdyttiin jakamaan sopivista kohdista metodeiksi, mahdollistaen toteutettuja lohkojen vertailu alkuperäiseen toteutukseen. Näitä uusia lohkottuja metodeja vasten toteutettiin yksikkötestit, joiden vastaanottamat syötteet ja odotetut vasteet luotiin tallentamalla ne väliulosottoina mat-tiedostoiksi Matlab-koodista. Lopulta luoduilla yksikkötesteillä kyettiin paikantamaan ohjelmointikielen vaihdossa tulleet virheet ja ne saatiin korjattua. Kehittäminen helpottui yksikkötestien myötä, sillä niistä saatiin välitön palaute toteutuksen oikeellisuudesta. Ohjelmien lopputulokset alkoivat lopulta näyttää samankaltaisilta ja saavutettiin ensimmäinen toimiva versio.

3.1.1 Yksikkötestien luominen ja testitiedon kerääminen

Alla olevassa koodissa on esimerkki, kun alkuperäiseen silmukkaan sijoitettiin väliulosotto, jossa getDirectionalH- ja entropyfilt-funktioiden syötteet ja ulostulot tallennetaan tiedostoon. Tiedostoon kirjoitettava tieto on sanakirjamuodossa (dictionary), jossa jokainen alkio koostuu avaimesta ja arvosta. Arvona voi olla yksittäisiä lukuja tai matriiseja, kuten esimerkissä alpha on kokonaisluku, delta on liukuluku ja loput ovat matriiseja.

```
for k=1:nAlphas
    alpha= alphas(k);
    [H,s]= getDirectionalH(alpha,delta);
    entr = entropyfilt(H);
    data(k).H= H; data(k).J= entr; data(k).s= s;
    filename = sprintf(get_directional_H_testdata%d.mat', k);
    save(filename, 'alpha', 'delta', 'z', 'zs', 'H', 's', 'entr');
end
```

Yksikkötestit luotiin hyödyntäen pytest-kirjastoa. Yksikkötestien alussa mat-tiedostot luettiin __load_parameters-funktiolla, josta saatiin testin syötteet ja vasteet. Luettavien tiedostojen nimet luetteloitiin erillisessä muuttujassa, josta pytest-kirjaston parametrize-funktio osaa poimia testitapaukset. Tiedostoista saatuja ulosottoja vasten Python-

toteutusta kyettiin vertaamaan Matlab-ajoon eri tapauksilla. Näin Python-toteutuksen voitiin todeta olevan mahdollisimman samanlainen kuin alkuperäinen. Suurin osa toteutuksista antoi suuruusluokaltaan samanlaisia tuloksia ja yksikkötestien tuloksien väitteissä (assert) sallittiin pieni ero käyttäen pytest-kirjaston approx-funktiota. Väitteiden sallima toleranssi valittiin tapauskohtaisesti huomioiden toteutuksen luonne ja toleranssin merkitys toteutuksessa.

```
testdata = [
      ('get_directional_H_testdata1.mat'),
      ...
]
@pytest.mark.parametrize("filename", testdata)
def test_get_directional_H(filename):
    alpha, delta, z, zs, H, s = __load_parameters(filename)

    actualH, actualS = get_directional_H(alpha, delta, z, zs)

    assert H == pytest.approx(actualH, abs=1e-15)
    assert s == pytest.approx(actualS, abs=1e-13)
```

Tämän kaltainen toleranssi johtuu kahden eri ohjelmointikielen liukulukujen pyörityseroista tai matemaattisten toimintojen toteutuseroavaisuuksista. Nollalla jakoa pyrittiin Python-toteutuksissa välttämään lisäämällä sisään syötettyihin matriiseihin hyvin pieni lisäys esimerkiksi $EPS = 1 \times 10^{-15}$, sillä nollalla jako pysäyttää Python-ohjelman suorituksen. Matlab-toteutuksessa nollalla jako ei (riippuu ohjelman asetuksista) lopeta ohjelman suoritusta, vaan palauttaa vastauksena IEEE:n (Institute of Electrical and Electronics Engineers) mukaisen aritmeettisen esityksen positiiviselle tai negatiiselle äärettömälle. Äärettömyys voi johtua nollalla jaosta ja ylivuodosta, jotka johtavat liian suuriin tuloksiin esitettäväksi tavanomaisina liukulukuarvoina. Esimerkiksi positiivisen äärettömän (Inf) tuottavat $1/0$, $1x10^{-1000}$, 2^{1000} ja e^{1000}. Negatiisen äärettömän (-Inf) tuottaa $\log 0$, kun taas ei-numeerisen arvon (NaN, Not a Number) tuottavat $Inf - Inf$ ja Inf/Inf.

Yksikkötestien väitteet pysyivät pääsääntöisesti pienen toleranssialueen sisällä, paitsi Python-toteutuksessa käytetty yksilöity entropiatoteutus antoi hieman eri arvoja kuin vastaava Matlab entropyfilt -funktio. Tuolla eroavaisuudella ei käytännössä ole merkitystä, sillä tärkeintä on saada samankaltainen pikselikohtainen entropiatieto samasta kohdasta molemmilla funktioilla.

3.1.2 Entropiafunktion toteutushaasteet

Matlab entropyfilt -funktio palauttaa paikallisen entropia-arvo ulostulotaulukon J harmaasävykuvasta I. Ulostulotaulukossa J jokainen alkio sisältää kuvan I kyseisen pikselin entropia-arvon. Arvo lasketaan kyseisen pikselin naapuripikseleistä 9x9-pikselin kokoiselta ikkuna-alueelta. Sisään syötetyn kuvan I reunoilla oleville pikselien kohdalla funktio käyttää ulostuloon J symmetristä täyttöä, jossa täytettävät pikselit heijastetaan sisään syötetyn kuvan I reunoilta. Naapuripikseleiden ikkuna-aluetta voi säätää nHood-argumentilla, jonka oletusarvo on true(9). Entropian laskeminen jokaiselle pikselille naapuripikselien mukaan onnistuu yhden rivin koodilla. [23]

```
J = entropyfilt(I,nhood)
```

Python-toteutuksessa käytetyllä yksilöidyllä entropiatoteutuksella pyrittiin mahdollisimman samankaltaiseen tulokseen. Toteutusvaiheessa kokeiltiin myös vaihtoehtoista tapaa, jossa Matlab entropiafunktion toiminnallisuus vietiin Python-paketiksi. Matlab-ajoympäristön asennustyökalun avulla asennettiin Library Compiler -sovellus, jonka avulla Matlab-toteutuksesta voidaan luoda Python-paketti [24]. Oheisella koodilla määriteltiin entropiatoteutus, joka sijaitsi m_entropyfilt.m-tiedostossa. Toteutus käärittiin m_entropyfilt-nimiseksi Python-paketiksi dokumentaation mukaisesti.

```
function J= m_entropyfilt(I)
    J= entropyfilt(I);
end
```

Python-toteutuksessa Matlab-ohjelmalla luodun Python-paketin käyttöönotto oli suoraviivasta. Matlab-ohjelmassa luotu m_entropyfilt-paketti ja Matlab Engine API -rajapinta tuotiin Python-koodiin import-määreillä sisään. Entropiapaketin alustus tehtiin funktion alussa. Sisään syötetty taulukko muunnettiin Matlab-muotoon, joka välitettiin entropiatoteutukselle. Lopuksi kutsutaan terminate-funktiota. Lopputulos ei kuitenkaan ollut tavoitteiden mukainen. Python paketin käytön myötä ohjelmiston käyttö muuttui epävakaaksi ja hitaaksi. Tämä korostui etenkin rinnakkaisessa toteutuksessa (kappale 3.2.2), jonka takia kokeilusta luovuttiin.

```
import m_entropyfilt
import matlab

# Alustus ennen entropiafunktion kutsua
m_entropyfilt_pkg = m_entropyfilt.initialize()
...

def entropyfilt(H):
    Hm = matlab.double(H, size=H.shape)
    return m_entropyfilt_pkg.m_entropyfilt(Hm)

...
# Lopetuskutsu, kun entropiafunktiota ei enää tarvita
m_entropyfilt_pkg.terminate()
```

Tutkimuksen aikana entropiatoteutus (seuraava koodi) pyrittiin järjestelmään valmiilla Python-kirjastoilla ja toteutus pohjautuu skimage.filters.rank-moduulin entropy-funktioon käyttäen skimage.morphology-moduulin square-funktiota määrittämään naapuripikselien ikkuna-alueen. Entropy-funktio vastaanottaa 8-bittisen harmaasävykuvan. Kuvan muunnos ja skaalaus oikeaan muotoon on järjestetty kustomoidulla minmaxscale-normalisointifunktiolla ja tämän jälkeen normalisoitu kuva muutetaan 8-bittiseksi skimage.util-moduulin img_as_ubyte-funktiolla. [25] [26] [27]

```
import numpy as np
from skimage.filters.rank import entropy
from skimage.morphology import square
from skimage import util

EPS = 1e-15

def minmaxscale(x):
    """
    Normalize given value [-1, 1]
    """
    # Avoid dividing by zero
    return 2 * (x - np.min(x)) / (np.max(x) - np.min(x) + EPS)- 1

def entropyfilt(I):
    """
    J = entropyfilt(I) returns the array J,
    where each output pixel contains the entropy value of
    the 9-by-9 neighborhood around the corresponding pixel
    in the input image I.
    """
    return entropy(util.img_as_ubyte(minmaxscale(I)), square(9))
```

3.2 Koodin optimointi

Ensimmäistä versiota Python-ohjelman toteutusta koestettiin Sievistä kesällä 2020 otettua fotogrammetrista kuvaa vasten. Koestuksen avulla pyrittiin keräämään referenssitietoa ohjelman suorituskyvystä, jotta suorituskykyä voidaan verrata optimoinnin jälkeen lähtötilanteeseen. Alkuperäisestä kuvasta koostettiin laskentaan neljä kuvaa kahdeksan pikselin ($k = 8$) välein aina kuvan reunoihin asti siten, että kuvan näytteen poiminta aloitetaan neljän pikselin ($l = k/2$) välein eri kohdista. Lähtökoodinaatit kaksiulotteiden kuvan näytteiden poimintaan ovat $[(0,0),(l,0),(0,l),(l,l)]$. Laskentaan välitetystä yksittäisestä kuvasta tulee $1/k^2$ kokoinen sisääntulokuvan ollessa kokoa 1.

Laskentaan välitetystä kuvasta luodaan yhdeksän eri versiota $I, I\oplus(\delta,0), I\oplus(\delta,\delta), \dots, I\oplus(\delta,-\delta)$ siirtämällä kuvassa käytettyjä pikseleitä vaaka- ja pystysuuntaisesti, jotka lopulta välitetään laskenta-algoritmille (Algoritmi 1). Siirtotoimenpide vaatii pikseleiden täytön kuvan reunoissa, sillä tässä tapauksessa kuvan ulkopuolelta ei ole pikselitietoa tarjolla. Tällöin pikselin korvaaminen toteutetaan toistamalla aiemman pikselin arvo. Esimerkiksi kuvan vasen reuna täytetään indeksiarvoin 0, 0, 1, 2, …, i ja oikea reuna 0, 1, …, j-1, j, j. Kuvien versioiden luonti on laskennallisesti kevyt, mutta kuluttaa muistia tuhlailevasti. Toisaalta kokonaisuutta tarkastellen toimenpiteen myötä laskentanopeus kasvaa huomattavasti, sillä laskentaa voidaan suorittaa vektorisoinnin avulla usealle kuvalle kerralla.

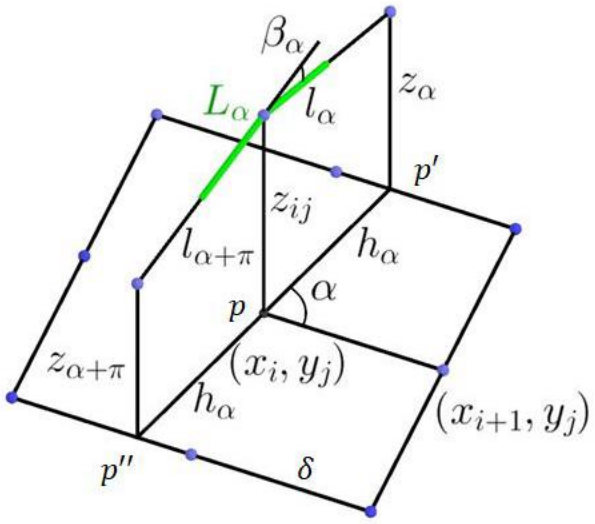
Algoritmi 1. Laske suunnattu kaarevuus ja kaltevuus [28]

$\alpha, \delta \rightarrow h_\alpha\ (vaakasuuntainen\ etäisyys)$
$\alpha \rightarrow kehäpisteen\ p'\ valinta\ (u)$
$Interpoloi\ z_\alpha = I_{prev}(1-u) + I_{next}u$
$Interpoloi\ z_{\alpha+\pi}$
$Laske\ h_\alpha, h_{\alpha+\pi}, l_\alpha, l_{\alpha+\pi}$
$Laske\ \beta_\alpha$
$L_\alpha = (l_\alpha + l_{\alpha+\pi})/2$

Suunnatun kaarevuuden ja kaltevuuden laskenta (Kuva 8) suoritetaan jokaisella kuvan versiolla ja lopulta laskennan tulokset koostetaan yhdeksi alkuperäisen kokoiseksi kuvaksi. Laskennassa kuvasta lasketaan jokaisesta pisteestä $p = (i, j)\delta$ suunnattu kaarevuus $\kappa_\alpha(p)$:

$$\kappa_\alpha(p) = \frac{\beta_\alpha(p)}{L_\alpha(p)}$$

ja kaltevuus annetun etäisyyden δ ja suunnan $\alpha \in A \subset [0, \pi] \subset \mathbb{R}$ mukaan. Suunnatun kaarevuustiedon perusteella lasketaan minimientropia, jonka avulla kuvasta löydetään haluttu kuvio. [28]



Kuva 8. Suunnatun kaarevuuden ja kaltevuuden laskeminen [28]

Ohjelmointikielen vaihdon jälkeen alkuperäinen Matlab-koodilla toteutettu ohjelma oli noin kaksi kertaa nopeampi kuin Python-koodilla toteutettu ohjelma (Taulukko 2). Ohjelmat veivät ajon aikana suurin piirtein saman verran keskusmuistia. Ajoa ei voitu suorittaa pienemmällä k arvoilla, sillä Matlab-ohjelmaa ajettaessa käytettävissä oleva keskusmuisti loppui ja ohjelman suoritus keskeytyi. Keskusmuistia käytetyssä tietokoneessa oli 16 gigatavua.

Taulukko 2. Matlab- ja Python-ohjelmien suorituskykyvertailu (ensimmäinen vaihe)

Ohjelmaversio ($k = 8$)	**Matlab**	**Python (v1)**
Keskusmuisti alussa (Gt)	7,6	6,5
Keskusmuisti ajossa (Gt)	10–11	8–10
Suoritusaika (s)	88	177

Python-ohjelman pääosioihin suoritettiin riippuvuusanalyysi (kappale 2.4.2), jonka avulla selvitettiin voiko osioita vektorisoida. Vektorisointi osoittautui monessa kohtaa hyvin kompleksiseksi. Ohjelma ylätasolla koostui viidestä silmukasta. Jokaisen silmukan sisällä suoritettiin indeksointia, siivutusta ja erilaisia matemaattisia operaatioita. Näiden silmukoiden sisällä lausekkeiden välillä oli suoritusriippuvuutta, hallintariippuvuutta ja resurssiriippuvuutta. Suoritusriippuvuus ilmenee siten, että seuraava lauseke oli riippuvainen edellisen lausekkeen tuloksesta (Seuraavan esimerkin rivit 10 ja 12). Hallintariippuvuus esiintyi monien ehtolausekkeiden muodossa, jotka muuttavat toimintalogiikkaa iteraatioiden välillä (ei esimerkissä). Resurssiriippuvuus esiintyi siten, että matemaattisten operaatioiden vastaukset tallennettiin laajemmalla näkyvyydellä oleviin taulukoihin (rivit 13–15). Tällöin tulee huolehtia, että vastaus tallennetaan alkuperäistä tietoa vastaavaan paikkaan. Muutoin lopputulos voi olla korruptoitunut tai osoitetaan väärään muistipaikkaan. Lisäksi koodissa tehdään viittauksia taulukon alkion ominaisuuksiin, joka vaikeuttaa vektorisointia entisestään.

```
1  def assemble_image(sz, nAlphas, data):
2      Js = np.empty(nAlphas)
3      H = np.empty(sz)
4      A = np.empty(sz).astype(int)
5      s = np.empty(sz)
6
7    for i in np.arange(sz[0]):
8        for j in np.arange(sz[1]):
9            for k in np.arange(nAlphas):
10                Js[k] = data[k].J[i, j]
11
12            index_k = np.argmin(Js)
```

```
13              H[i, j] = data[index_k].H[i, j]
14              A[i, j] = index_k
15              s[i, j] = data[index_k].s[i, j]
16
17      return H, A, s
```

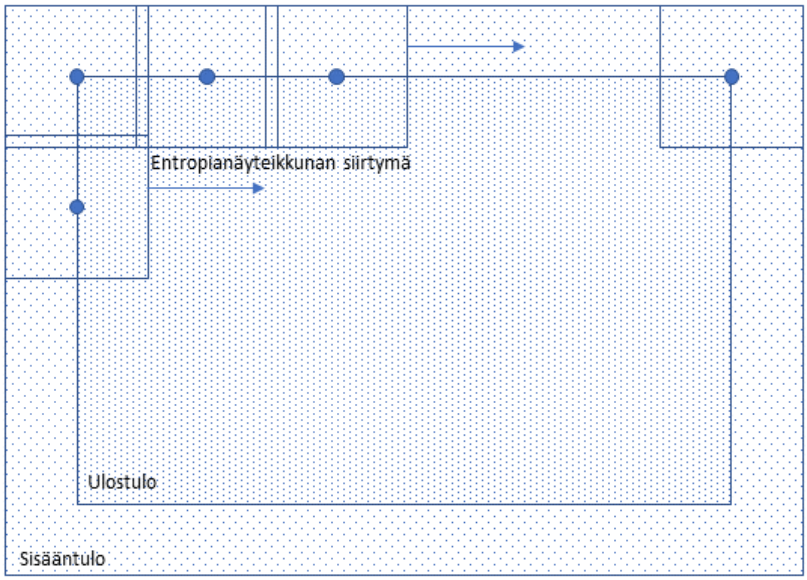
Ohjelmiston pullonkauloja etsittiin käyttäen cProfile-työkalua (kappale 2.5.3). Raportin mukaan koodin hitain osuus on edellistä esimerkkiä assemble_image-funktiota kutsuva assemble_images-funktio, jonka suoritusaika on noin 80 % koko ohjelman suorituksesta. Funktion tarkoitus on kerätä assemble_image-funktiolle sen tarvitsemat syötteet ja palauttaa sen vastaukset ylätasolle yhteen koostettavaksi. Funktion käyttämä aika jakaantuu kahtia, josta noin 53 % vietetään assemble_image-funktiossa ja loput ajasta entropyfilt-funktion suorittamisessa. Entropian laskentaa ja assemble_image-funktion suorituskykyä ei kyetty tämän tutkimuksen aikana parantamaan. Entropialaskenta yritettiin korvata varianssilaskennalla, mutta suorituskyvyllisesti operaatiot olivat samaa luokkaa. Todennäköisesti ainoa keino saada koodin suorituskykyä tehokkaammaksi on refaktoroida koko ohjelma vektoritietoinen ohjelmointi edellä.

Optimointia suoritettiin pääsääntöisesti turhien silmukoiden poistolla, rinnakkaistamisella, vektorisoinnilla ja muuttujien sekä metodien uudelleen järjestelyillä. Turhia silmukoita esiintyi lähinnä edistymispalkkitoiminnossa, jolloin standardiulostuloon tulostettiin prosentuaalinen lukema edistymisestä. Tämä korvattiin Python-toteutuksessa tqdm-kirjaston edistymispalkkitoiminnolla, sillä se on nopea ja yleisesti hyväksi havaittu toteutus.

Muuttujien uudelleen järjestämisiä toteutettiin kaikkialla yleisesti. Alkuperäisessä koodissa käytettiin muutamia globaaleja muuttujia. Python-toteutuksen lähtökohtana oli asettaa funktiot tilattomaksi ja niiden toimintalaajuus mahdollisimman pieneksi. Tämä auttaa osana yksikkötestattavuuteen. Ohjelmistokehitys suoritettiin koodin optimoinnin aikana yksikkötestaus edellä ja toteutuksen aikana saatiin vaste siitä, ollaanko etenemässä oikeaan suuntaan. Koodi dokumentoitiin koko ajan Python-dokumentaation hyväksi havaittuja tapoja käyttäen. Koodista on mahdollisuus tulostaa dokumentaatio ja ohjelmistokehitystyökalut osaavat korostaa funktion dokumentaation kursorin ollessa funktion päällä.

3.2.1 Ikkunointifunktion käyttöönotto

Silmukoiden sisällä taulukoiden siivuttamista vähennettiin käyttämällä ikkunointia skimage.util.shape-moduulin tarjoaman view_as_windows-funktion avulla. Entropianäyteikkunan kooksi asetettiin 100x100 ja askelien määräksi 96 pikseliä (Kuva 9). Ikkuna siirtyy kohti reunaa ja siirtyy seuraavalla riville reunan tullessa vastaan.



Kuva 9. Kuvan siivuttaminen entropianäyteikkunan avulla

Näin jokainen ikkuna saatiin aavistuksen edellisen päälle kaikissa ulottuvuuksissa ja lopputuloksessa ikkunoinnin aiheuttamat rajakohdat pehmentyvät. Oheinen koodi kuvaa view_as_windows-funktion käyttöä koodissa. Siivutettu kuva välitetään process_ProcessPoolExecutor-funktiolle, joka käynnistää prosessoinnin rinnakkaisesti. Funktio palauttaa lopputuloksen koostamisessa tarvittavat tiedot kahteen muuttujaan.

```
def run_process(img, delta=0.03292):
    window_shape = (100, 100)
    step = 96
    img_patches = view_as_windows(img, window_shape, step)
    ...
    inds, results = process_ProcessPoolExecutor(
            delta, img_patches, sz, max_cpu)
```

3.2.2 Rinnakkaistaminen ProcessPoolExecutor-luokan avulla

Aiemmin esitetyn assemble_image-funktion suoritus rinnakkaistettiin luomalla rinnakkaisia prosesseja concurrent.futures-moduulin ProcessPoolExecutor-luokan ja sen map-funktion avulla. Map-funktio asetettiin viipaloimaan sisään syötetty tieto määrään, joka on jaollinen suorittimien määrän kanssa (seuraava koodi). Näin silmukoita kyettiin ajamaan useita rinnakkain ja Python-ohjelma saatiin noin 2,2 kertaa nopeammaksi. Tätä voidaan kutsua triviaalisti rinnakkaistuvaksi algoritmiksi (kappale 2.3.1).

```
def process_ProcessPoolExecutor(delta, img_patches, sz, max_cpu):
    inds = get_xy_indices(sz[0], sz[1])
    args = [(img_patches[row, col]) for row, col in inds]
    chunksize = ceil(len(args)/max_cpu)
    with ProcessPoolExecutor(max_workers=max_cpu) as executor:
        results = list(tqdm(
            executor.map(assemble_images,
                         itertools.repeat(delta),
                         args,
                         chunksize=chunksize),
            total=len(args),
            desc='Processing directional curvatures'))
        return inds, results
```

Kehittämisessä käytetyssä tietokoneessa oli 12 ydintä, joiden avulla työtä voitiin suorittaa yhtä aikaa. Nopeutuksen perusteella laskettuna assemble_image-funktion osuus ohjelmasta on noin 60 %. Vaikka ytimiä olisi äärettömästi, ei sillä olisi kovinkaan suurta vaikutusta suorituskykyyn. Tämä voidaan todentaa Amdahlin lain avulla (kappale 2.3.3) ja laskennallisesti 60 % rinnakkaistuksen määrällä maksimiarvo nopeutukselle on noin 2,47.

3.3 Lopputuloksen koostaminen

Lopputulos koostetaan process_ProcessPoolExecutor-funktion palauttamien tietojen perusteella. Results-muuttuja sisältää kaarevuus-, entropia- ja kaltevuustiedot. Inds-muuttuja sisältää indeksitiedot, joilla viitataan kuvan pikseleihin mistä edelliset tiedot

ovat peräisin. Nuo kaksi muuttajaa pakataan yhdeksi, jotta niitä voidaan iteroida rinnakkain silmukan sisällä. Indeksien osoittamissa muistipaikoissa olevat tulokset tallennetaan tyhjiin taulukkoihin, joita käytetään yhdistämään niissä olevat pienemmät osat yhdeksi isoksi kuvaksi. NumPy-kirjaston transpose-funktiolla siirretään taulukoiden akseleita parametrina annetun indeksitaulukon mukaisesti, jonka jälkeen ne muotoillaan uudelleen alkuperäisen kuvan kokoiseksi käyttäen NumPy-kirjaston reshape-funktiota. Nämä syntyneet taulukot syötetään raportointi- ja kuvagenerointifunktioille.

```
inds, results = process_ProcessPoolExecutor(
      delta, img_patches, sz, max_cpu)
...
for ind, result in zip(inds, results):
      Hfinal[ind[0], ind[1]] = result[0]
      Afinal[ind[0], ind[1]] = result[1]
      sFinal[ind[0], ind[1]] = result[2]
inds = None
img_patches = None

# Convert patches back to image shape
Hs = Hfinal.transpose(0, 2, 1, 3).reshape(img_shape)
Hfinal = None
As = Afinal.transpose(0, 2, 1, 3).reshape(img_shape)
Afinal = None
Ss = sFinal.transpose(0, 2, 1, 3).reshape(img_shape)
sFinal = None

return Hs, As, Ss
```

4 TULOKSET

Vektorisoinnilla havaittiin tutkimuksen alussa merkittäviä suorituskyky parannuksia. Aivan ensimmäinen Matlab-toteutus oli naiiviversio (versio 0), joka käsitteli kaavojen laskenta-akselia pikseli kerrallaan. Tämän ohjelmaversion tehtävänä oli myöhempien versioiden oikeellisuuden tarkistaminen. Sen suorituskyky oli huomattavasti hitaampi, kuin myöhemmin toteutettu vektorisoitu versio (versio 1), jossa laskenta-akselia prosessoitiin kuva-alue kerrallaan. Ohjelmointikielen vaihto Matlab- ja Python-kielen välillä onnistui tavoitteiden mukaisesti. Toiminnallisuudet kyettiin toteuttamaan Python-ohjelmassa vektorisoidun Matlab-koodin kaltaiseksi käyttäen eri Python-kirjastoja. Python-ohjelma mahdollistaa toteutuksen tarjoamisen palvelumallisesti ja sen toiminnallisuuteen voidaan vaikuttaa parametrisoinnilla. Yksikkötestien avulla kyettiin varmistamaan ohjelmakoodin laadullisuus ja ettei ohjelmistokehityksen aikana tehdyt muutokset muuta ohjelman toimintaa.

Sivutavoitteena oli saada Python-ohjelmasta alkuperäistoteutusta nopeampi. Python-ohjelma oli optimoinnin jälkeen suorituskyvyltään hieman nopeampi kuin vektoroitu Matlab-toteutus. Saavutettu hyöty olikin ohjelmiston helpompi julkistettavuus ja integroitavuus muiden järjestelmien kanssa, sillä Matlab-ohjelmien integrointi on vaikeaa ja lisenssikäytäntöjen takia kallista. Suorituskyvyltään vielä nopeampi toteutus vaatii ohjelmakoodin uudelleenkirjoittamisen vektorisointitietoisesti. Toimenpiteet tällöin tulee kohdistaa suunnatun kaarevuuden ja kaltevuuden sekä entropian laskentaan, sillä profiloinnin mukaan nämä kohdat ovat ohjelmiston pullonkauloja.

Ohjelmaversioita koestettiin optimoinnin jälkeen siten, että kappaleessa 3.2 käytetystä fotogrammetrisesta kuvasta (14141×8517 pikseliä) otettiin huomattavasti pienempi osa käyttöön (3001×1807 pikseliä). Näin ohjelmien suorituksia voitiin verrata keskenään nopeammin ja ohjelmia voitiin käyttää pienemmällä k=1 arvolla. Taulukko 3 tuloksista havaitaan, että optimoitu rinnakkaistamista ja ikkunointia käyttävä Python-ohjelma suoriutuu laskennasta hieman Matlab-ohjelmaa nopeammin. Optimoitu ohjelma kuluttaa hieman enemmän muistia, kuin ensimmäinen versio. Suuruusluokassaan muistinkulutus on alkuperäistä toteutusta vastaavalla tasolla.

Taulukko 3. Matlab- ja Python-ohjelmien suorituskykyvertailu (optimoinnin jälkeen)

Ohjelmaversio ($k = 1$)	**Matlab**	**Python (v1)**	**Python (optimoitu)**
Keskusmuisti alussa (Gt)	8,3	6,6	6,6
Keskusmuisti ajossa (Gt)	8–9	7–7,4	7–8
Suoritusaika (s)	20	43	18

Kehittämisessä käytetyssä tietokoneessa käytössä ollut keskusmuisti ei riittänyt ajamaan Matlab-ohjelmaa alkuperäisellä fotogrammetrisella kuvalla. Kiinnostavaa olisi ollut ajaa Matlab-ohjelma tuolla kuvalla ja verrata ajoa Python-ohjelmaan. Tämä sillä yllättävää oli, että optimoitu Python-ohjelma suoriutui kuvasta ilman keskusmuistin loppumista. Lähtötilanteessa muistia oli käytetty 7,7/16 Gt ja ajon aikana muistin käyttö oli 10–15 Gt välillä. Ohjelman suoritus kesti 11 minuuttia 32 sekuntia. Jatkokehitysajatuksena kehittämisessä käytettävään tietokoneeseen on syytä varata ainakin 32 Gt keskusmuistia.

5 ARKKITEHTUURISET VAIHTOEHDOT

Tämän kappaleen aikana käydään läpi arkkitehtuurisia ratkaisuja ja vaihtoehtoja, joilla tutkimuksen kohdesovellus voidaan tarjota paremmin saavutettavana ja skaalautuvana esimerkiksi pilviympäristössä. Samalla käydään läpi keskeisiä käsitteitä ja eri pilvipalveluiden tarjoamia palveluita. Tulevaisuuden tavoitteena on laskea koko Suomen tasolla ilmassa tapahtuvan laserskannauksen (Airborne Laser Scanning, ALS) tai korkeusmallitiedon (Digital Elevation Model, DEM) avulla. Tämä tavoite asettaa arkkitehtuurille tarpeen käyttää skaalautuvaa suurteholaskentaa (High Performance Computing, HPC) ja tietojärviä valtavan aineistomäärän takia.

5.1 Massatieto

Massatieto (big data) on yleistynyt nimitys valtaville tietomäärille, joiden käsittelyssä ei voida soveltaa perinteisiä tiedonhallinnointitapoja. Tietoa voi tulla monista eri lähteistä nopeaan tahtiin ja suuria määriä. Tieto voi olla rakenteellista, jäsentämätöntä tai puolirakenteellista esimerkiksi tekstiä, kuvia, ääntä, loki-, sijainti-, sensori- ja klikkaustietoja. Tieto voi olla sellaista, että sitä ei voida analysoida sellaisenaan tai niiden tietomäärän käsittely on suorituskyvyllisesti haastavaa, kuten tutkimuksessa käsitellyt fotogrammetriset kuvat.

Tietoa voidaan kerätä tietolähteistä massamuistille. Tietomäärien kerääminen yhteen paikkaan vaatii valtavia massamuisteja. Samalla riski tiedon häviämisestä kasvaa, mikäli massamuistiin tallennettu tieto korruptoituu jostain syystä. Varmuuskopiomekanismin luominen tämänkaltaiseen massamuistiin voi muodostua taloudellisesti kalliiksi. Tieto voidaan vaihtoehtoisesti virtualisoida, jolloin eri tiedot haetaan vasta kun niitä tarvitaan. Tämänkaltaisesta lähestymisestä on hyötyä ajan suhteen muuttuvien tila- tai tilannetietojen kanssa, joiden muutoshistoriaa halutaan seurata. Virtuaalisessa käsittelykerroksessa tieto kerätään, kuvataan ja mallinnetaan lähdesovelluksen tai orkestrointityönkulun avulla, jonka jälkeen tieto julkaistaan tietovarastoon. Tämän jälkeen tietoa voidaan käsitellä analysointia ja raportointia varten. [29, s. 125-145]

5.2 Massatietojen eräkäsittely

Massatietojen eräkäsittely (batch processing) on yleinen skenaario käsitellä tietovarastoon kerätyt tiedot rinnakkaistyöllä (Kuva 10), joka voidaan käynnistää orkestrointityönkulun avulla. Eräkäsittely soveltuu erityisesti triviaalisti rinnakkaistuviin (kappale 2.3.1) työkuormiin. Käsittely sisältää useita iteratiivisia vaiheita. Rinnakkaistöiden valmistuttua tulokset ladataan analyyttiseen tietovarastoon, josta analytiikka- ja raportointikomponentit voivat tehdä kyselyjä. Esimerkiksi fotogrammetriset kuvat maastosta voidaan kerätä tietolähteistä tietovarastoon ja käsitellä niitä rinnakkaistöinä yön yli seuraavan päivän analytiikkaraporttien luomiseksi. Tämän kaltainen prosessi on toteutettavissa pilvilaskentaparadigmaa käyttäen suurteholaskentaa ilman oman laitteiston hankkimista palveluna pilvipalvelun tarjoajan ylläpitämänä esimerkiksi Microsoft Azure tai Amazon Web Services. [29, s. 125-145] [30, s. 135-139] [31]

Kuva 10. Massatiedon eräkäsittelyputki suurteholaskentaa käyttäen [31]

5.3 Pilvinatiivit mallit ja ratkaisut

Maastokuvien prosessointi voidaan järjestää pilviympäristöissä monella tavalla. Tulevaisuuden tavoitteena on laskea koko Suomi laserskannaus- tai korkeusmallitiedon avulla. Koko Suomen mittakaavassa tehtävä laskenta vaatii paljon kuvamateriaalia. Pitkäaikaista säilytystä varten kannattaa käyttöön ottaa tietojärvipalvelu esimerkiksi Azure Data Lake Storage Gen2 tai AWS Lake Formation. Tietojärvipalvelut sisältävät yleensä varmuuskopiointi- ja tietojen palautusominaisuudet. Lyhytaikaisessa säilytyksessä voidaan myös käyttää objektisäiliötä esimerkiksi Azure Blob Storage tai AWS S3. Tällöin kuvat voidaan tarvittaessa poistaa tietovarastosta eräkäsittelyn

valmistuttua vapauttaen tilaa uusille eräkäsittelyille. Eräkäsittely voidaan järjestää käyttäen eräkäsittelypalveluita esimerkiksi Azure Batch tai AWS Batch. Eräkäsittelypalvelujen käytöstä maksetaan vain, kun niissä ajetaan eräkäsittelytöitä. Eräkäsittely voidaan toteuttaa Python-koodina ja lisäksi sen ajoympäristö voidaan paketoida käynnistettäväksi Docker-sovelluskonttina. Sovelluskonttien levykuvat voidaan julkaista ja ottaa eräkäsittelyn käyttöön käyttämällä konttirekisteripalveluita esimerkiksi Azure ACR- tai Amazon ECR -palveluita.

Viestijonon avulla töiden eräkäsittelytöiden vastaanottaminen voidaan muuttaa skaalautuvaksi asettamalla viestijonon taakse useita kuluttajia, jotka purkavat viestijonoa tehtäviksi eräkäsittelyyn (Kuva 11). Tätä kutsutaan kilpailevien kuluttajien malliksi [32]. Kuluttajien määrää kuluttajapalveluiden altaassa voidaan muuttaa dynaamisesti viestijonossa olevien viestien määrän perusteella.

Kuva 11. Kilpailevien kuluttajien malli [32]

Microsoft Azure -ympäristössä eräkäsittelytöiden orkestrointi voidaan järjestää räätälöidysti käyttäen palvelimetonta Azure Durable Functions -tilafunktiopalvelua tai vähäkoodisena sovelluskehityksenä (low-code) käyttäen Azure Data Factory -palvelua. Jälkimmäistä vaihtoehtoa käytettäessä tarvitaan erillinen viestijonon purkutoiminto, joka käynnistää Azure Factory -putken. Tämä voidaan toteuttaa palvelimettomalla ja tilattomalla Azure Functions -palvelulla, joka tallentaa työpyynnön objektisäiliöön tai vaihtoehtoisesti käynnistämällä Azure Data Factory -putki ohjelmointirajapintakutsulla. Azure Data Factory -putken käynnistys on pääsääntöisesti mahdollista vain reagoimalla

tietovarastojen muutossyötteisiin ja ainakaan tällä viestijonon muutokset eivät ole tuettu tapa. [31]. Amazon-ympäristössä orkestrointi kannattaa toteuttaa käyttäen AWS Step Functions -palvelua, joka on vastaava palvelu kuin Azure Data Factory. Orkestrointityökulun hallintapalvelut tukevat MapReduce-paradigmaa esimerkiksi Azure Data Factory -palvelussa HDInsights MapReduce -toiminnallisuus tai AWS Step Functions -palvelussa map-toiminnallisuus [34] [35]. [36] [37] [38]

Suurteholaskennassa pilviympäristöissä käytetään usein MapReduce-ohjelmointiparadigmaa, joka mahdollistaa massiivisten tietomäärien rinnakkaiskäsittelyn. MapReduce koostuu kahdesta toiminnosta, kartoitus ja yhdistäminen. Kartoituksessa syötetieto hajotetaan useaksi pienemmäksi tietojoukoksi. Yhdistäminen käyttää kartoituksen tietojoukkoja, joka suorittaa yhteenvetotoiminnon luoden pienemmän yhdistetyn tuloksen. Avoimen lähdekoodin suosituin MapReduce-ohjelmointiparadigman toteutus on Apache Hadoop -viitekehys, joka pohjautuu isäntä-renkiarkkitehtuuriin. Isäntä solmu on vastuussa sovelluksen kartoitustehtävien ajoittamisesta renkisolmuissa, jotka pitävät sisällään paloja syötetiedosta. Renkisolmu ilmoittaa isäntäsolmulle, kun se voi vastaanottaa tehtävän hoitaakseen. Isäntä etsii jokaiselle rengille sopivan tehtävän ja kukin renki aloittaa vuorollaan tehtävän suorittamisen. Yhdistämistehtävä suoritetaan lopulta isäntäsolmussa. Apache Hadoop, Apache Spark ja monia muita Hadoop-viitekehyksen työkaluja tarjotaan pilviympäristöissä yhteen koostettuna palveluina esimerkiksi Azure HDInsights ja Amazon EMR. Tämä malli soveltuu hyvin eräkäsittelytöiden jakamiseen ja lopputuloksien yhdistämiseen, mikäli päädytään käyttämään jotakin orkestrointityökulun hallintapalvelua. [30, s. 30-31] [39]

5.4 Ratkaisuarkkitehtuuri käyttäen massatietojen eräkäsittelyä

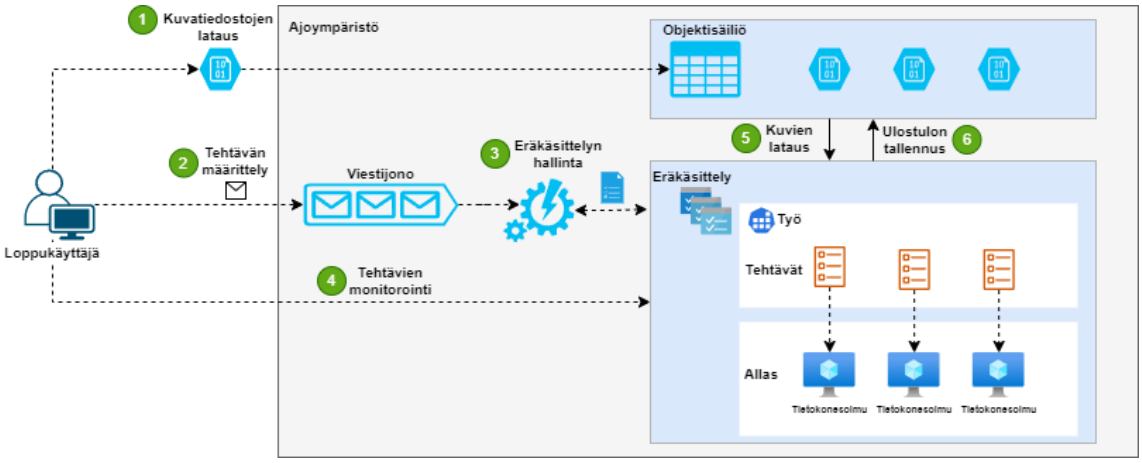
Tämä ratkaisuarkkitehtuuri ei ota kantaa käyttöliittymään liittyviin asioihin, vaan määrittää eräkäsittelyn prosessin tietojen ja tapahtumien kulun. Ratkaisuarkkitehtuuri on pilviagnostinen, joten toteutus on toteutettavissa pilviympäristöön tai omaan konesaliin. Pilvinatiiveilla ratkaisuilla toteutus on luonnollisesti vaivattomammin toteutettavissa saavuttaen samalla pilviympäristöjen saavutettavuus- ja skaalautuvuushyödyt, joten esitettävä ratkaisuehdotus keskittyy pilviympäristön hyödyntämiseen. Ratkaisua voi jatkokehittää tarpeiden mukaan sen löyhän riippuvuuden vuoksi. Esimerkiksi viestijono ja eräkäsittelyn hallinta on korvattavissa toisella orkestrointityökulun hallinnalla melko pienellä työllä.

Taulukko 4 listaa eri ratkaisuarkkitehtuurissa käytetyt komponentit vasten pilvipalvelun tarjoajien palveluita. Vastaavia tuotteita on saatavilla myös omassa konesalissa tarjottavaan palveluun, mutta niiden vertailu ja valinta perustuu enemmän valitun infrastruktuurin varaan ja siksi ei ole mielekästä listata samassa taulukossa. Pilvinatiivissa ratkaisuarkkitehtuurissa viestijonopalvelua käytetään yhdessä eräkäsittelyn hallinnan kanssa. Näin olemassa oleva koodipohja voidaan käyttää ilman suurempia muutoksia. Eräkäsittelyn hallinnassa käytetään räätälöityä palvelimetonta tilafunktiopalvelua, jossa ajetaan prosessin käynnistys ja ikkunointitoiminto (kappale 3.2.1). Triviaalisti rinnakkaistuva osa koodista (kappale 3.2.2) voidaan käynnistää eräkäsittelypalvelussa. Orkestrointityökulun hallintapalvelun käyttöönotto on hieman työläämpi, edellyttää konfigurointia ja uuden ohjelmakoodin toteuttamista. Tästä esimerkiksi Azure Data Factory -palvelun vähäkoodinen kehitys aiheuttaa prosessin luomisen alusta lähtien. Kuvien ja lopputuloksien tallentaminen järjestetään objektisäiliön avulla, joka on helposti vaihdettavissa tietojärvipalveluksi.

Taulukko 4. Pilvinatiivit palvelut massatietojen eräkäsittelyssä [40]

Osa-alue	**Microsoft Azure**	**Amazon Web Services**
Viestijono	Azure Service Bus Queue	Amazon SNS
Objektisäiliö	Azure Blob Storage	Amazon S3
Eräkäsittelyn hallinta	Azure Durable Functions	AWS Step Functions
Eräkäsittely	Azure Batch	AWS Batch

Eräkäsittelyprosessi saa alkunsa, kun asiakas lataa käsiteltävät kuvatiedostot objektisäiliöön (1). Asiakas lähettää viestijonoon ajourien kartoitustehtäväviestin (2), joka sisältää kartoitettavan alueen määrittelyn, siinä tarvittavien tiedostojen tiedot ja käsittelyssä muut tarvittavat parametrit. Eräkäsittelyn hallinta (3) on rekisteröity viestijonon kuluttajaksi. Eräkäsittelyn hallinta käyttää kilpailevien kuluttajien mallia (ks. 57), jolloin rinnakkaiset käsittelijät lukevat viestijonosta viestejä kilpaa luoden niiden perusteella eräkäsittelyn työt ja tehtävät. Käsittelijä huolehtii sisään tulevan kuvan ikkunoinnin jakaen sen erätöille ja lopulta koostaa erätöiden vastauksista lopputuloksen. Asiakas voi monitoroida (4) tehtävien suoritusta reaaliajassa. Erätyöt lataavat töissä käytettävät kuvat objektisäiliöstä (5) ja lopulta tallentavat töiden ulostulot objektisäiliöön (6). Asiakas voi ladata ulostulon objektisäiliöstä.



Kuva 12. Ratkaisuarkkitehtuuri käyttäen massatietojen eräkäsittelyä

6 YHTEENVETO

Tutkimuksen aikana käsiteltiin vektorisoinnin ja rinnakkaisuuden hyödyntämistä ja toteuttamista osana ohjelmaprojektin ohjelmistokielen vaihtoa. Ohjelmistojen suoritusnopeutta tehostetaan moderneissa käyttöjärjestelmissä suorittaen useita eri prioriteetin omaavia prosesseja yhtä aikaa. Yhdessä prosessissa voi olla yhdestä useaan säiettä, jotka käyttävät prosessin tarjoamaa yhteistä muistia. Prosessin tehtävänä on yhdistää sen säikeet loogisesti. Prosessien ja säikeiden suorituskykyä optimoidaan järjestelmässä toimivan vuorontajan avulla. Vuorontaja jakaa prosesseille ja säikeille vuorollaan optimaalisen ajan toimia niille asetettujen prioriteettien mukaan, jotta mikään prosessi tai säie ei nälkiinny tai lukkiudu. Tutkimuksessa käsiteltiin rinnakkaisuuden mukana tulevien ongelmia ja niiden ratkaisemiseen käytettäviä vakiintuneita ratkaisumalleja.

Ohjelma voidaan rinnakkaistaa jakamalla se rinnakkain ajettaviin osiin joko funktionaalisesti tai tietoperustaisesti. Tutkimuksen aikana tehdyssä toteutuksessa käytettiin tietoperustaista ositusta, jolloin osituksessa käytetty tieto jaettiin suoritinsidonnaisille tehtäville suoritettavaksi rinnakkaisiin prosesseihin. Rinnakkaisia operaatioita voidaan käynnistää suoritettavaksi monin eri hyväksi havaituin malleja. Syntaktiset ja semanttiset yksityiskohdat vaihtelevat eri ohjelmointikielien ja kirjastojen välillä, mutta useat noudattavat yleisimmin käytettyjä malleja. Tutkimuksen aikana tehdyn Python-toteutuksen kannalta kiinnostavimmat mallit olivat rinnakkaiset silmukat, haarautumat ja liittymät, tehtäväaltaat ja tuottaja–kuluttaja. Näistä toteutuksessa lopulta käytettiin rinnakkaiset silmukat mallia.

Ohjelmakoodin suoritusaikaa, suoritustehoa, muistinkulutusta ja eniten töitä vaativia kohtia tarkkailtiin käyttäen erilaisia profilointityökaluja. Näiden metriikoiden avulla kyettiin tunnistamaan ohjelman hitaat osat ja eniten resursseja vaativat kohdat, joita optimoimalla voitiin tehostaa ohjelman suoritusta. Profiloinnin ja riippuvuusanalyysin avulla selvitettiin vektorisoinnin edellytykset. Vektorisointi ei ollut triviaalista ja käytetyt algoritmit olivat haastavia tai jopa mahdottomia rinnakkaistaa vektorisoimalla. Algoritmissa suoritettujen tehtävien välillä oli riippuvuuksia, joita ei kyetty pienellä työllä korjaamaan. Nämä kohdat jäivät loogisesti sarjallisesti suoritettaviksi. Amdahlin lain mukaan ohjelman sarjallinen osa määrittää rinnakkaisuudella saavutettavan

suoritusajan parannuksen raja-arvon, vaikka operaatioita voitaisiin suorittaa rinnakkain ääretön määrä.

Python ohjelmointikielen avulla vektorisointi ja rinnakkaistaminen kyettiin toteuttamaan melko tehokkaasti käyttämällä C/C++ -kielellä toteutettuja kirjastoja. Ohjelmointikielen vaihtoprosessissa oli haasteita vektorien käsittelyoperaatioiden ja trigonometristen funktioiden muunnostyössä. Lisäksi kielten väliset eroavaisuudet, kuten liukulukujen eroavaisuudet ja nollalla jakamiseen liittyvät säännöt tulivat ottaa huomioon Python-toteutuksessa. Kuvien luontiin liittyvät osuudet, entropia- ja histogram-toteutuksien muunnostyöt osoittautuivat haasteellisiksi. Entropiatoteutus osoittautui myös pullonkaulaksi ja se yritettiin korvata varianssin minimoinnilla. Suorituskyvyllisesti operaatiot paljastuivat kuitenkin samantasoisiksi, joten toteutuksessa päädyttiin lopulta käyttämään alkuperäisen ohjelman mukaista entropialaskentaa. Parempaa suorituskykyä tavoiteltaessa tulee sarjallinen silmukassa sijaitseva ohjelmakoodin osa vektorisoida, jossa entropia-laskentaa suoritetaan.

Muunnostyön vaikeudet johtuivat pääsääntöisesti siitä, että matemaattiset operaatiot olivat haastavia ymmärtää. Matemaatikon ja ohjelmistokehittäjän yhteistyön tulee olla hyvin tiivistä kehittämisen aikana, jotta toteutuksessa päädytään optimiratkaisuun. Tutkimuksen aikana pidettiin yhdestä kahteen viikon välein virtuaalisia tapaamisia, joissa käytiin tutkimuksen aikana esiin tulleita kysymyksiä läpi. Niiden aikana matemaattiset esitykset konkretisoitiin mahdollisimman tarkasti. Tämä oli edellytys tehokkaalle ohjelmistokehitykselle, sillä vektoritietoinen ohjelmistokehitys vaatii algoritmien täydellistä ymmärtämistä. Yksikkötestien avulla kyettiin ohjelmakoodin toimivuus todentamaan. Lisäksi kehitystyön edetessä toteutuksien muuttumattomuus ja eheys kyettiin varmistamaan. Yksikkötestien käyttö ohjasi ohjelmistokehitystä segmentoimaan ongelmaa pienempiin yksiköihin, jolloin myös ongelman selvitys helpottui. Dokumentaation luomisen ja käytön tärkeys korostui ohjelmistokehitystyössä, sillä matemaattisten operaatioiden käyttäminen ja toiminnallisuuden päättely pelkän olettaman perustalla johti tutkimuksen alussa vääriin lopputuloksiin.

Ohjelmakoodin vektorisoinnilla pelkästään saavutettiin merkittäviä tuloksia, kun verrataan alkuperäiseen naiiviin pikseli kerrallaan suoritettavaan laskentaan. Kuva-alue kerrallaan laskenta yhdessä ikkunoinnin ja rinnakkaistamisen avulla saavutettiin tehokas

toteutus. Toteutus on melko helposti tarjottavissa ratkaisuarkkitehtuuriehdotuksen mukaisesti skaalautuvana ja korkeasti saavutettavana palveluna käyttäen massatietojen eräajomallia. Palvelu voidaan tarjota omasta konesalista, mutta pilvinatiiveilla malleilla ja ratkaisuilla toteutus voidaan suorittaa kustannustehokkaasti keskittyen toiminnallisuuden kehittämiseen yli infrastruktuurin kehittämisen. Pilvilaskentaparadigman hyödyntäminen on palvelun tulevaisuuden käyttönäkymien kannalta paras vaihtoehto. Työvälineet ja kehittäminen on ohjautunut viimeisen kahden kymmenen vuoden aikana kohti virtualisoitua ja palveluna ostettua infrastruktuuria. Kehitystiimejä ohjataan ja järjestetään nykyään pääsääntöisesti virtuaalisesti saavuttaen tehokkaan paikkariippumattoman työskentelyn, joka mahdollistaa helpommin osaavien kehittäjien löytämisen ohjelmistoprojektiin. Nämä kaikki yhdistettynä Python-kieliseen ohjelmaan ja yleisesti avoimen lähdekoodin käyttöön, palveluiden ja ohjelmien liitettävyys tosiin palveluihin ja ohjelmiin paranee.

LÄHTEET

[1] I. Haikala ja H.-M. Järvinen, Käyttöjärjestelmät, osa/vuosik. 2, Helsinki: Talentum, 2004.

[2] R. H. Arpaci-Dusseau ja A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2018.

[3] T. Rauber ja G. Rünger, Parallel Programming: For Multicore and Cluster Systems, Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010.

[4] G. Lanaro, Q. Nguyen ja S. Kasampalis, Advanced Python Programming : Build High Performance, Concurrent, and Multi-Threaded Apps with Python Using Proven Design Patterns, 1st edition toim., Packt Publishing Ltd, 2019.

[5] M. L. Scott, Programming Language Pragmatics, Elsevier, 2012.

[6] Intel Corporation, A Guide to Vectorization with Intel® C++ Compilers, 2010.

[7] Nvidia, ”CUDA C++ Best Practices Guide,” Nvidia, 11 Toukokuu 2022. [Online]. Saatavilla: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. [Haettu 29 Toukokuu 2022].

[8] Python Software Foundation, ”The Python Profilers,” Python Software Foundation, 29 Toukokuu 2022. [Online]. Saatavilla: https://docs.python.org/3/library/profile.html. [Haettu 29 Toukokuu 2022].

[9] Cornell University Center for Advanced Computing, ”Vectorization: Vectorizable Code,” Cornell University, 2022. [Online]. Saatavilla: https://cvw.cac.cornell.edu/vector/coding_vectorizable. [Haettu 29 Toukokuu 2022].

[10] M. Lutz, Learning Python, O’Reilly, 2013.

[11] ”NumPy Reference 1.22,” NumPy, 14 Tammikuu 2022. [Online]. Saatavilla: https://numpy.org/doc/stable/reference/index.html#reference. [Haettu 24 Huhtikuu 2022].

[12] Python Software Foundation, ”threading — Thread-based parallelism — Python 3.10.6 documentation,” Python Software Foundation, [Online]. Saatavilla: https://docs.python.org/3/library/threading.html. [Haettu 6 Elokuu 2022].

[13] Python Software Foundation, ”multiprocessing — Process-based parallelism — Python 3.10.6 documentation,” Python Software Foundation, [Online]. Saatavilla: https://docs.python.org/3/library/multiprocessing.html. [Haettu 6 Elokuu 2022].

[14] Python Software Foundation, ”concurrent.futures — Launching parallel tasks — Python 3.10.3 documentation,” Python Software Foundation, 19 Maaliskuu 2022. [Online]. Saatavilla: https://docs.python.org/3/library/concurrent.futures.html. [Haettu 19 Maaliskuu 2022].

[15] ”nschloe/tuna: Python profile viewer,” [Online]. Saatavilla: https://github.com/nschloe/tuna. [Haettu 15 Elokuu 2022].

[16] ”SnakeViz,” [Online]. Saatavilla: https://jiffyclub.github.io/snakeviz/. [Haettu 15 Elokuu 2022].

[17] Python Software Foundation, ”timeit — Measure execution time of small code snippets — Python 3.10.6 documentation,” Python Software Foundation, [Online]. Saatavilla: https://docs.python.org/3/library/timeit.html. [Haettu 15 Elokuu 2022].

[18] NumPy Developers, ”Indexing routines — NumPy v1.22 Manual,” NumPy, [Online]. Saatavilla: https://numpy.org/doc/stable/reference/arrays.indexing.html. [Haettu 16 Elokuu 2022].

[19] ”Module: util — skimage v0.20.0.dev0 docs,” scikit-image, [Online]. Saatavilla: https://scikit-image.org/docs/dev/api/skimage.util.html. [Haettu 16 Elokuu 2022].

[20] NumPy Developers, ”numpy.histogram — NumPy v1.23 Manual,” [Online]. Saatavilla: https://numpy.org/doc/stable/reference/generated/numpy.histogram.html. [Haettu 20 Elokuu 2022].

[21] The MathWorks, ”Histogram plot - MATLAB hist,” [Online]. Saatavilla: https://se.mathworks.com/help/matlab/ref/hist.html. [Haettu 20 Elokuu 2022].

[22] scikit-image, ”Module: exposure — skimage v0.20.0.dev0 docs,” [Online]. Saatavilla: https://scikit-image.org/docs/dev/api/skimage.exposure.html#histogram. [Haettu 20 Elokuu 2022].

[23] The MathWorks, Inc., ”Local entropy of grayscale image - MATLAB entropyfilt,” [Online]. Saatavilla: https://se.mathworks.com/help/images/ref/entropyfilt.html. [Haettu 18 Elokuu 2022].

[24] The MathWorks, ”Generate a Python Package and Build a Python Application,” [Online]. Saatavilla: https://se.mathworks.com/help/compiler_sdk/gs/create-a-python-application-with-matlab-code.html. [Haettu 20 Elokuu 2022].

[25] scikit-image, ”Module: filters.rank — skimage v0.19.2 docs,” [Online]. Saatavilla: https://scikit-image.org/docs/stable/api/skimage.filters.rank.html#skimage.filters.rank.entropy. [Haettu 18 Elokuu 2022].

[26] scikit-image, ”Module: morphology — skimage v0.19.2 docs,” [Online]. Saatavilla: https://scikit-image.org/docs/stable/api/skimage.morphology.html?highlight=morphology#skimage.morphology.square. [Haettu 18 Elokuu 2022].

[27] scikit-image, ”Module: util — skimage v0.19.2 docs,” [Online]. Saatavilla: https://scikit-image.org/docs/stable/api/skimage.util.html?highlight=util#skimage.util.img_as_ubyte. [Haettu 18 Elokuu 2022].

[28] P. Nevalainen, *Programming within/with/for AI DTEK0071 — Programming Paradigms in Practice 25.2.2021,* 2021.

[29] Z. Tejada, ”Big data architectures,” Microsoft, [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/. [Haettu 3 Syyskuu 2022].

[30] Microsoft, ”Competing Consumers pattern,” Microsoft, [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/architecture/patterns/competing-consumers. [Haettu 4 Syyskuu 2022].

[31] Microsoft, ”Pipelines and activities in Azure Data Factory and Azure Synapse Analytics,” Microsoft, 22 Tammikuu 2022. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/data-factory/concepts-pipelines-activities?tabs=data-factory. [Haettu 4 Syyskuu 2022].

[32] Microsoft, ”Invoke MapReduce Programs from Data Factory,” Microsoft, 25 Lokakuu 2021. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/data-factory/v1/data-factory-map-reduce. [Haettu 3 Syyskuu 2022].

[33] Amazon Web Services, ”Introducing larger state payloads for AWS Step Functions,” Amazon Web Services, 3 Syyskuu 2020. [Online]. Saatavilla: https://aws.amazon.com/blogs/compute/introducing-larger-state-payloads-for-aws-step-functions/. [Haettu 3 Syyskuu 2022].

[34] Amazon Web Services, ”AWS Batch,” Amazon Web Services, [Online]. Saatavilla: https://aws.amazon.com/batch/. [Haettu 3 Syyskuu 2022].

[35] Microsoft, ”Process large-scale datasets by using Data Factory and Batch,” Microsoft, 30 Elokuu 2022. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/data-factory/v1/data-factory-data-processing-using-batch. [Haettu 3 Syyskuu 2022].

[36] Microsoft, ”What is Azure Batch?,” Microsoft, 15 Joulukuu 2021. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/batch/batch-technical-overview. [Haettu 3 Syyskuu 2022].

[37] Microsoft, ”What is Apache Hadoop in Azure HDInsight?,” Microsoft, 31 Maaliskuu 2022. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/hdinsight/hadoop/apache-hadoop-introduction. [Haettu 3 Syyskuu 2022].

[38] Microsoft, ”AWS to Azure services comparison,” Microsoft, 8 Elokuu 2022. [Online]. Saatavilla: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services. [Haettu 3 Syyskuu 2022].

[39] J. Kołodziej, P. Florin ja D. Ciprian, Modeling and Simulation in HPC and Cloud Systems. Vol. 36., Springer International Publishing AG, 2018.

[40] S. Srinivasan, Guide to Big Data Applications. Vol. 26., Springer International Publishing AG, 2017.