Tampereen yliopisto

Topi Haavisto

MONIAJOISEN C++-YMPÄRISTÖN SUORITUSKYKYPULLONKAULOJEN ANALYSOINTI

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Kari Systä
Tarkastaja: Matti Rintala
Huhtikuu 2022

TIIVISTELMÄ

Topi Haavisto: Moniajoisen C++-ympäristön suorituskykypullonkaulojen analysointi
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Huhtikuu 2022

Tässä työssä käsitellään Nokia Solutions and Networks Oy:n DCAP-ohjelmiston Kerääjä-komponentin Parsija-pluginin suorituskykyä. Työn tavoitteena on etsiä ja mahdollisuuksien mukaan korjata pluginin suorituskykypullonkauloja samanaikaisuuteen liittyen. Aiemmin toteutetuissa suorituskykymittauksissa huomattiin pluginin suorituskyvyn skaalautuvuuden olevan muihin plugineihin verrattuna merkittävästi huonompaa prosessorien tai ytimien määrän kasvaessa.

Työssä käydään läpi samanaikaisuuden teoriaa, samanaikaisen koodin suoritustekijöitä, lukkojen käyttöä jaettua dataa käsiteltäessä sekä välimuistioperaatioiden merkitystä samanaikaisessa monisäikeisessä koodissa. Myös *perf*-komennon käyttöä suorituskykypullonkaulojen etsimiseen esitellään.

Tutkimuksessa keskityttiin eniten CPU-aikaa kuluttavaan funktioon, jossa kriittisen alueen suojaamiseksi käytetyn yksinkertaisen spinlockin suorituskykyä vertailtiin mittausten avulla muokatun spinlockin sekä mutexin suorituskykyihin. Muokatussa spinlockissa pyrittiin ottamaan huomioon välimuistioperaatioiden vaikutuksia, minkä seurauksena ilmeisin tulos olikin funktion noin 62 % alkuperäistä nopeampi suoritusaika juuri muokattua spinlockia käytettäessä. Pluginin kokonaissuorituskyvyssä ei kuitenkaan huomattu eroa, joten tutkimuksen tavoitteen voidaan katsoa täyttyneen vain osittain.

Avainsanat: Samanaikaisuus, suorituskyky, säie, C++, mutex, spinlock

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

ABSTRACT

Topi Haavisto: Analyzing the bottlenecks of a concurrent C++ environment
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
April 2022

This thesis deals with the performance of the Parser plugin of the Collector component in the DCAP software by Nokia Solutions and Networks Oy. The goal is to find and if possible, fix the performance bottlenecks related to concurrency in the plugin. Previous performance measurements have shown that compared to other plugins the performance scalability of the Parser plugin is considerably worse when additional processors or cores are added.

The thesis introduces some of the theory of concurrency, the factors of concurrent code, the use of locks when handling shared data and the effects of cache operations in multithreaded code. The use of the *perf* command to find performance bottlenecks is also talked about.

The investigation focused on the function where the CPU spent most of its time. The performance of the spinlock protecting the critical section in the function was compared to the performances measured with a modified spinlock and a mutex. The effects of the cache operations were attempted to take into consideration with the modified spinlock which yielded the most obvious result of the measurements: the execution time of the function with the modified spinlock was about 62 % faster than the original. The total performance of the plugin was not observably changed so the goal of this thesis can only be seen as partially achieved.

Keywords: Concurrency, performance, thread, C++, mutex, spinlock

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Tämä työ tehtiin Nokia Solutions and Networks Oy:lle. Haluan kiittää työnantajaa diplomityön tekomahdollisuudesta sekä työkavereitani, esimiestäni ja ohjaajia diplomityön aiheen muodostumisen auttamisessa ja työn edistämisen tukemisessa. Erityiset kiitokset kuuluvat professori Kari Syställe työn erinomaisesta ohjauksesta ja tärkeistä kommenteista.

Tampereella, 25.4.2022

Topi Haavisto

SISÄLLYSLUETTELO

1. JOHDANTO ... 1
2. TAUSTATIEDOT JA TEORIA ... 3
2.1 Lähtökohta ja ongelma ... 3
2.2 Samanaikaisuus ... 4
2.3 Samanaikaisen koodin suorituskykytekijöitä ... 6
2.3.1 Prosessorien määrä ... 6
2.3.2 Kilpailu datasta ja välimuistipoukkoilu ... 7
2.3.3 Väärä jakaminen ... 9
2.3.4 Datan läheisyys ... 10
2.3.5 Ylitilaaminen ja liialliset kontekstivaihdot ... 11
2.4 Skaalautuvuus ... 12
2.5 Lukot ... 12
2.5.1 Mutex ja spinlock ... 13
2.5.2 Deadlock eli lukkiutuminen ... 14
2.6 Muistirakenne ... 15
2.6.1 Uniform Memory Access (UMA) ... 15
2.6.2 Non-Uniform Memory Access (NUMA) ... 16
2.6.3 NUMA-tutkimus ... 16
2.7 Perf-profilointityökalu ... 17
3. TUTKIMUS ... 19
3.1 Tutkimusympäristö ... 19
3.2 Suorituskyvyn tarkastelumenetelmä ... 20
3.3 Profilointi ... 22
3.4 Lukkojen vertailu ... 24
3.4.1 Suoritusaikamittaus ... 26
3.4.2 Viivemittaus ... 29
3.4.3 Joutoaikamittaus ... 33
3.5 Tulosten arviointi ... 36
4. YHTEENVETO ... 39
LÄHTEET ... 40

LYHENTEET JA MERKINNÄT

4G	4. sukupolven langaton tiedonsiirtoteknologia
ANS.1	rajapintakuvauskieli (Abstract Syntax Notation One)
C++	ohjelmointikieli
CPU	suoritin, prosessori (Central Processing Unit)
DCAP	tutkittava ohjelmisto (Data Collection and Analytics Platform)
Deadlock	säikeiden lukkiutuminen
DRAM	hajasaantimuistin tyyppi (Dynamic Random Access Memory)
I/O	siirräntä, tiedonsiirto (input/output)
MESI	välimuistiprotokolla (Modified, Exclusive, Shared, Invalid)
Mutex	synkronointiprimitiivi (mutual exclusion)
NUMA	vaihtelevan hakuajan muistirakenne (Non-Uniform Nemory Access)
UMA	yhtenäisen hakuajan muistirakenne (Uniform Memory Access)
Spinlock	lukitusmenetelmä jaetun datan suojaamiseksi

.

1. JOHDANTO

Ensimmäisten tietokoneiden tietyssä ajassa suorittama työ voitiin määritellä suorittimen kellotaajuudella. Teknologian kehittyessä prosessoreista tuli yhä pienempiä, jolloin lämpötila ja muut fyysiset rajoitteet alkoivat rajoittamaan prosessorien kellotaajuutta. Ratkaisuna prosessorien suorituskyvyn kasvattamiselle tällöin muodostui useampien ytimien lisääminen prosessoriin, jolloin pystyttäisiin suorittamaan useampia tehtäviä sekunnissa kasvattamatta kellotaajuutta tai muuttamalla prosessorin kokoa ja lämpöominaisuuksia. Näin ongelmaksi jäi enää se, kuinka hyödyntää näitä ylimääräisiä ytimiä. [1]

Ytimien hyödyntämiseksi tarvitaan ohjelmistoa, joka kykenee suorittamaan tehtäviä samanaikaisesti [1]. Tällaisen ohjelmiston kehittäminen ei aina ole kuitenkaan helppoa, ja siihen voi liittyä monia suorituskykyongelmia tuottavia seikkoja.

Tässä työssä keskitytään Nokia Solutions and Networks Oy:n DCAP-ohjelmiston (Data Collection and Analytics Platform) Kerääjä-komponentin Parsija-pluginin suorituskyvyn tarkastelemiseen. Taustana tälle työlle toimii aiemmin suoritettu yrityksen sisäinen tutkimus, jossa mittausten tulosten perusteella pääteltiin, että kyseisen pluginin suorituskyky skaalautuu laitteistotehoa lisättäessä huomattavasti huonommin kuin Kerääjä-komponentin muut pluginit. Kyseessä on siis moniajoinen C++-ympäristö, jossa tällä hetkellä ei saada laitteistoa täysin hyödynnettyä, minkä vuoksi tämän työn tavoitteena on löytää ja mahdollisuuksien mukaan korjata samanaikaisuuteen liittyviä suorituskykypullonkauloja Parsija-pluginista.

Tutkimusmenetelmänä samanaikaisuuteen liittyvän teorian hankkimiseen käytettiin Tampereen yliopiston kirjaston Andor-hakupalvelulla löydettyjä julkaisuja sekä muita internetistä löytyviä lähteitä. Itse tutkimuksessa suorituskykypullonkaulojen etsimismenetelmänä käytettiin Linux-käyttöjärjestelmän omaa profilointityökalua ja suorituskyvyn mittausmenetelmänä toimi koodissa käytetyt kahden pisteen aikaeromittaukset.

Tutkimuksessa rajoituttiin profiloinnin tuloksena löydettyyn eniten suoritinaikaa kuluttavaan funktioon. Funktiossa olevan jaetun datan käsittelyä suojaavan lukitusmenetelmän suorituskykyä mitattiin ja vertailtiin muihin lukitusmenetelmiin, minkä tuloksena päädyttiin lopulta noin 62 % nopeutukseen funktion suoritusajassa. Parsija-pluginin kokonaissuorituskyvyssä ei tästä huolimatta huomattu eroa.

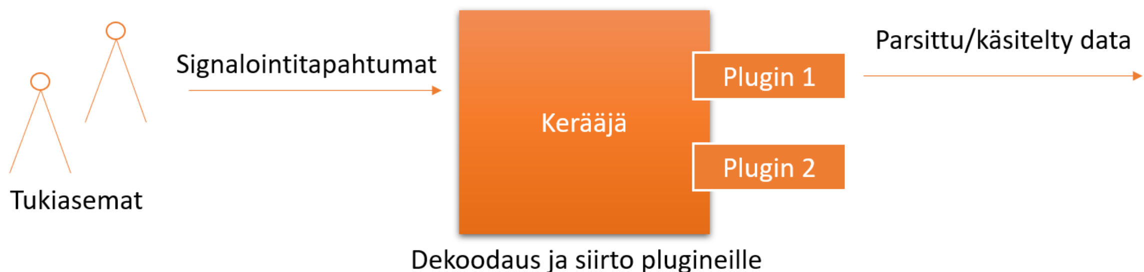
Aluksi luvussa 2 käydään läpi lähtökohdat ja ongelma sekä teoreettista taustaa samanaikaisuuteen, sen haasteisiin ja hallintaan liittyen. Tämän jälkeen luvussa 3 esitellään tutkimusmenetelmät ja aineisto sekä esitellään tutkimuksen tuloksia että pohditaan niiden merkitystä. Lopuksi luvussa 4 työn sisältö ja tutkimuksen tulokset tiivistetään yhteenvedossa.

2. TAUSTATIEDOT JA TEORIA

Tässä luvussa käydään läpi työn lähtökohta ja ongelma. Lisäksi esitellään työn kannalta tarpeellinen teoria tutkimuksessa esiintyvien yleisimpien suorituskykypullonkaulojen kannalta.

2.1 Lähtökohta ja ongelma

Työn lähtökohtana on DCAP-ohjelmiston Kerääjä-komponentin Parsija-pluginin suorituskyvyn tutkiminen ja mahdollinen parantaminen. DCAP kerää ja analysoi tukiasemilta tulevaa mobiilitietoliikenteen monitorointidataa, ja Kerääjä vastaa 4G-monitorointidatan vastaanottamisesta, dekoodaamisesta ja eteenpäin välittämisestä plugineille analysointia varten. Datan käsittely ja analysointi on jaettu plugineihin, koska niillä on toisistaan eroavia tehtäviä ja niistä ulos saatava käsitelty data lähetetään eri osoitteisiin. Kuvassa 1 on havainnollistettuna tutkittava komponentti osana koko järjestelmää sekä tiedon kulku ja käsittely kyseisessä järjestelmässä.



Kuva 1. Tutkittava komponentti osana koko järjestelmää ja tiedon kulku järjestelmässä.

Ongelmana on Parsija-pluginin suorituskyvyn huono skaalautuvuus suoritettavien säikeiden määrän kasvaessa muihin plugineihin verrattuna. Suorituskyky ei kasva oletetusti prosessoriytimiä lisäämällä, ja tutkimus pyrkiikin löytämään tämän ilmiön aiheuttavia mahdollisia ongelmakohtia.

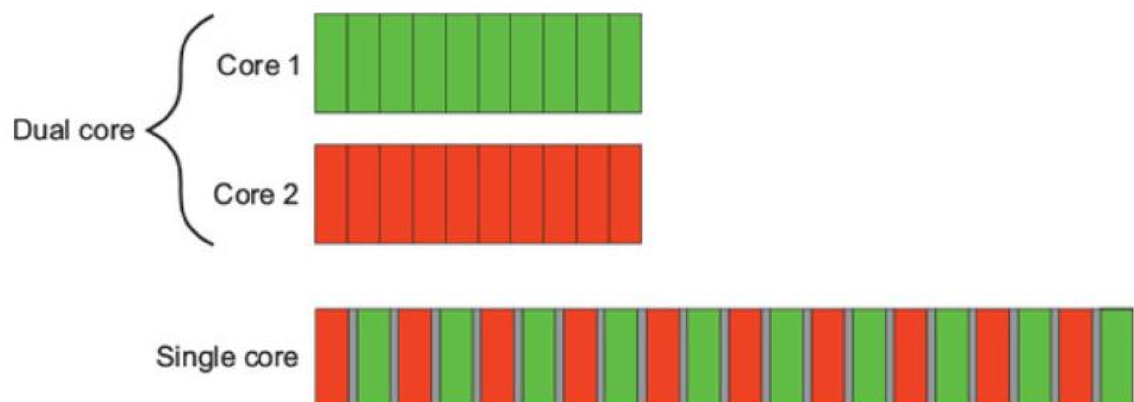
Idea Parsija-pluginin suorituskyvyn tutkimiselle ja optimoinnille lähti liikkeelle aiemmin suoritetusta Nokian sisäisestä NUMA-tutkimuksesta (Non-Uniform Memory Access), jossa huomattiin, että laitteisto ei ollut rajoittava tekijä. Tätä käsitellään lisää kohdassa ”NUMA-tutkimus”.

2.2 Samanaikaisuus

Yksinkertaisimmillaan kuvailtuna samanaikaisuus on kahden tai useamman tapahtuman tapahtumista samaan aikaan. Historiallisesti suurimmassa osassa tietokoneita on ollut vain yksi prosessori, jossa on vain yksi ydin. Tällainen kone ei pysty suorittamaan useaa tehtävää samanaikaisesti, mutta se kykenee silti vaihtelemaan monien tehtävien välillä useita kertoja sekunnissa. Tehtävien pala palata suorittaminen ja tehtävien välillä vaihtelu tapahtuu niin nopeasti, että tehtävien suorittaminen vaikuttaa samanaikaiselta. Tehtävien vaihtelun tapauksessa puhutaankin siis samanaikaisuudesta, sillä vaihtelun luoman samanaikaisuuden illuusion ansiosta käyttäjä ei pysty erottamaan, milloin tehtävän suoritus keskeytetään toisen tehtävän suorittamista varten. [2]

Monta prosessoria sisältäviä tietokoneita on käytetty palvelimissa ja korkean suoritustehon laskentatehtävissä jo vuosia, ja moniydinprosessoreita sisältävien tietokoneiden yleisyys kasvaa koko ajan. Toisin kuin yhden prosessorin ja ytimen järjestelmät, moniydinprosessorin tai useamman prosessorin sisältämät järjestelmät kykenevät todelliseen samanaikaisuuteen eli suorittamaan enemmän kuin yhtä tehtävää kerralla rinnakkain. Tätä kutsutaan laitteistosamanaikaisuudeksi. [2]

Ideaalisessa tilanteessa suoritettavia tehtäviä on korkeintaan yhtä paljon kuin tietokoneessa olevia suoritinytimiä. Kuvassa 2 on nähtävissä tilanteet, joissa on kaksi tehtävää jaettuna 10 samankokoiseen palaseen kaksiytimisen sekä yksiytimisen järjestelmän suoritettavana. [2]

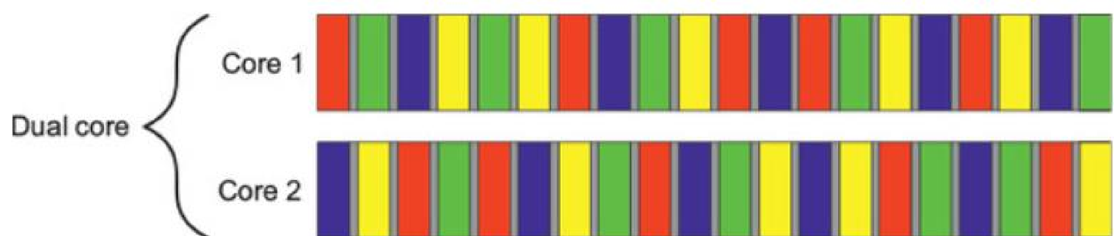


Kuva 2. Kahden samansuuruisen tehtävän suoritus kaksiytimisessä sekä yksiytimisessä järjestelmässä [2].

Kaksiytimisessä järjestelmässä kumpikin näistä tehtävistä voidaan suorittaa omissa ytimissään rinnakkaisesti. Yksiytiminen järjestelmä puolestaan käyttää tehtävien vaihtelua,

jolloin tehtävien palaset ovat lomittuneina keskenään. Lomituksen lisäksi palaset ovat hieman erillä toisistaan ohuiden harmaiden palkkien erottamana. Nämä harmaat palkit kuvastavat kontekstinvaihtoa, joka järjestelmän on suoritettava jokaisen tehtävänvaihdon yhteydessä. Kontekstivaihdot vievät aikaa, sillä käyttöjärjestelmän on tallennettava prosessorin tila ja ohjelmalaskuri nykyiselle tehtävälle, selvitettävä seuraava tehtävä sekä ladattava uudelleen seuraavan tehtävän tila prosessorille. Lisäksi prosessori voi joutua lataamaan seuraavaksi suoritukseen siirtyvän tehtävän käskyt ja datan välimuistiin, mikä voi estää prosessorin muiden käskyjen suorittamisen, aiheuttaen lisää viivettä. [2]

Vaikka samanaikaisuus laitteistossa on ilmeisintä moniydinprosessorin tai useamman prosessorin omaavissa järjestelmissä, jotkin prosessorit kykenevät suorittamaan useita säikeitä yhdellä ytimellä. Tärkeä tekijä on laitteistosäikeiden määrä, jolla määritellään niiden itsenäisten tehtävien määrä, jonka laitteisto pystyy aidosti suorittamaan samanaikaisesti. Vaikka järjestelmä kykenisi aitoon samanaikaisuuteen, on sillä yleensä enemmän tehtäviä kuin mitä se pystyisisi suorittamaan rinnakkain. Toisin sanoen, tehtävien vaihtelua käytetään myös aitoon samanaikaisuuteen kykenevissä järjestelmissä. Todellisuudessa suurimmassa osassa tietokoneita voi olla satoja säikeitä tai tehtäviä samanaikaisesti käynnissä, suorittamassa esimerkiksi taustaoperaatiota, vaikka tietokone olisi muuten käytännössä joutotilassa. Kuvassa 3 on esitettynä neljän eri tehtävän välillä suoritettavaa tehtävien vaihtelua kaksiydinprosessorin järjestelmässä. [2]



Kuva 3. Neljän tehtävän vaihtelu kaksiydinprosessorin järjestelmässä [2].

Kuvan tehtävät on jälleen paloiteltu ideaalisen tapauksen mukaisesti samankokoisiksi palasiksi. Käytännössä kuitenkin tehtävien suoritus ei jakaudu tasaisesti ja niiden aikatauluttaminen on epätasaista. [2]

2.3 Samanaikaisen koodin suorituskykytekijöitä

Koodin suorituskyvyn parantamisessa samanaikaisuutta hyödyntämällä on tärkeä tietää, mitkä tekijät vaikuttavat suorituskykyyn. Vaikka käytössä olisikin useampia säikeitä erillisiä tehtäviä varten, on varmistettava, että suorituskyky ei alene samanaikaisuuden käytön seurauksena. [2] Seuraavaksi suoritustekijöistä käydään läpi prosessorien määrä, datasta kilpaileminen ja välimuistipoukkoilu, väärä jakaminen, datan läheisyys sekä ylitilaaminen ja liialliset kontekstivaihdot.

2.3.1 Prosessorien määrä

Prosessorien määrä on ilmeisin iso ja tärkeä suorituskykytekijä monisäikeisessä ohjelmassa. Joissakin tilanteissa tiedetään ennalta, kuinka monta prosessoria tai ydintä kohdelaitteistossa on, jolloin tämä tieto voidaan ottaa huomioon ohjelmistoa suunniteltaessa. Ideaalisessa tilanteessa ohjelmistossa on saman verran säikeitä kuin sitä suorittavassa järjestelmässä on säikeitä. Suurimmassa osassa tapauksia tämä ei kuitenkaan toteudu, vaan monesti kehityksessä käytetty järjestelmä saattaa sisältää eri määrän ytimiä tai prosessoreja kuin kohdejärjestelmä. Samanaikaisen ohjelman käyttäytyminen ja suorituskyky saattavatkin vaihdella tällaisissa eri ympäristöissä, joten asia pitää ottaa huomioon kehityksessä ja mahdollisimman laaja testaus on suositeltavaa. [2]

Esimerkiksi 16-ytiminen prosessori pystyy samaan suoritustehoon kuin 4 neliydinprosessoria tai 16 yksiydinprosessoria, milloin jokaisessa tapauksessa pystytään suorittamaan yhteensä 16 säiettä samanaikaisesti. Jotta suoritustehosta saadaan kaikki irti, on sovelluksessa oltava vähintään 16 säiettä (olettaen, että järjestelmä ei suorita muita tehtäviä samaan aikaan). Muuten jätetään prosessointikykyä käyttämättä. Toisaalta voi käydä myös niin, että sovelluksessa on enemmän kuin 16 säiettä valmiina suoritukseen, jolloin prosessointiaikaa tuhlataan säikeiden välillä vaihteluun. Tätä tilannetta kutsutaan ylitilaamiseksi. [2]

Vaikka kaikki sovelluksessa pyörivät säikeet otetaan huomioon, muut järjestelmässä samaan aikaan suorituksessa olevat sovellukset vaikuttavat kuitenkin suorituskykyyn, olettaen tietysti, että suoritettava laskenta on mahdollista tehdä rinnakkaisesti. Prosessorien tai ytimien määrän kasvaessa kasvaa myös todennäköisyys ja suorituskykyvaikutus sille ongelmalle, että useat prosessorit yrittävät päästä käsiksi samaan dataan. [2]

2.3.2 Kilpailu datasta ja välimuistipoukkoilu

Prosessorissa jokaisella ytimellä on omat L1- ja L2-välimuistit sekä L3-välimuisti, joka on jaettu ytimien kesken. Dataa haetaan päämuistista tai L3-välimuistista alemman tason välimuisteille ja ytimen kannalta paras vaihtoehto on käyttää paikallista ja nopeinta L1-välimuistia tai vähintäänkin paikallista L2-välimuistia. Monisäikeisessä ohjelmassa on mahdollista, että useiden ytimien L1- ja L2-välimuisteista löytyy kopioita samasta datasta. Jos jokin ydin tekee muutoksia jaetussa muistissa olevaan dataan, välimuistin yhtenäisyyden säilyttämiseksi jokainen muutetusta datasta aiemmin tehty kopio invalidoidaan. Ytimien täytyy siis hakea uusi kopio datasta mahdollisesti jopa päämuistista asti, jos ne tarvitsevat kyseistä dataa myöhemmin. Nämä hakuoperaatiot muistissa tulevat sitä kalliimmiksi, mitä kauempaa dataa joudutaan hakemaan. [3]

Jos kahta säiettä suoritetaan samanaikaisesti eri prosessoreilla tai ytimillä ja molemmat säikeet lukevat samaa dataa, ongelmaa ei yleensä synny. Data tallennetaan tällöin molempien ytimien välimuistiin ja ytimet voivat jatkaa suoritustaan. Jos toinen säikeistä muokkaa dataa, tämän muutoksen täytyy kulkeutua myös toisen ytimen välimuistiin, mikä kuluttaa aikaa. Säikeiden operaatioiden luonteesta ja operaatioihin käytetyistä muistijärjestyksistä riippuen tämä muokkaus voi pysäyttää toisen ytimen suorituksen kokonaan siksi aikaa, kun muutos kulkeutuu muistilaitteiston läpi. Prosessorikäskyjen näkökulmasta tarkastellen tällainen odotus on erittäin hidas, jopa satojen yksittäisten käskyjen mittainen operaatio. Operaatioon kuluva aika tosin riippuu suurimmaksi osaksi laitteiston arkkitehtuurista. Tarkastellaan datasta kilpailemista ja välimuistipoukkoilua koodiesimerkin avulla, joka on esillä Kuvassa 4. [2]

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
    while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
    {
        do_something();
    }
}
```

Kuva 4. Koodipätkä, joka aiheuttaa datasta kilpailemista ja välimuistipoukkoilua useamman ytimen ja säikeen tapauksessa [2].

Muuttuja counter on globaali, joten kaikki funktiota processing_loop() kutsuvat säikeet muokkaavat samaa muuttujaa. Siksi jokaisen lisäyksen yhteydessä ytimen on varmistettava, että sillä on ajan tasalla oleva kopio muuttujasta counter välimuistissaan, minkä jälkeen ydin voi kasvattaa muuttujan arvoa ja julkaista sen muille ytimille. Operaatio fetch_add on luku – muokkaus – kirjoitusoperaatio, joten muuttujan viimeisin arvo täytyy hakea muistista. [2] Tämä on atominen operaatio, joka korvaa muuttujan arvon ensimmäisen parametrin ja nykyisen arvon summalla [4-5]. Atomisuus tarkoittaa, että muut säikeet eivät voi vaikuttaa muutettavaan arvoon sen luku- ja muokkaushetken välillä, mikä estää kilpailutilanteiden syntymisen (kilpailutilanteista lisää alaluvussa 2.5) [5-6]. Kun toinen säie suorittaa samaa koodia, täytyy muuttujan counter data siirtää ytimien ja niiden välimuistien välillä edestakaisin, jotta jokaisella ytimellä olisi viimeisin muuttujan arvo lisäystä tehdessä. Mikäli funktio do_something() on tarpeeksi lyhyt tai koodia on suorittamassa liian monta ydintä yhtä aikaa, ytimet voivat joutua odottamaan toisiaan. Tässä tapauksessa jokin ydin olisi valmiina kasvattamaan arvoa, mutta jokin toinen ydin on tekemässä sitä parhaillaan, jolloin muut joutuvat odottamaan muutoksen loppuun saamista ja sen kulkeutumista omille välimuisteilleen. Tällöin kilpailu datasta on kiivasta (high contention). Tapauksissa, joissa ytimet joutuvat vain harvoin odottamaan toisiaan, kilpailu on puolestaan vähäistä (low contention). [2]

Kuvan 4 silmukassa muuttujan counter arvoa välitetään useaan kertaan edestakaisin ytimien välimuistien välillä. Tätä kutsutaan välimuistipoukkoiluksi, ja sillä voi olla huomattava merkitys sovelluksen suorituskykyyn. Jos prosessorin suoritus seisahtuu välimuistisiirroksen takia, se ei voi tehdä mitään hyödyllistä työtä samaan aikaan, vaikka muita säikeitä olisi odottamassa vuoroaan päästäkseen suoritukseen. Tämä tietenkin on epäsuotuisaa koko sovelluksen kannalta. [2]

Samanlainen tilanne saadaan, jos säikeiden synkronointiin käytetään poissulkevaa mutex-lukkoa (mutual exclusion) atomisen operaation sijaan silmukassa (mutexista lisää alaluvussa 2.5.1), jolloin koodi on samankaltaista Kuvan 4 koodipätkän kanssa datan käsittelyn näkökulmasta. Mutexin lukitsemiseksi säikeen täytyy siirtää mutexin koostava koodi ytimelle ja muokata sitä. Kun säie on suorittanut tehtävänsä, se muokkaa mutexia uudelleen vapauttaakseen lukon, minkä jälkeen mutexin data pitää siirtää seuraavalle säikeelle, jotta se voidaan ottaa siellä haltuun. Siirtoaika lasketaan siis sen ajan päälle, mitä toinen säie on kuluttanut lukon vapautusta odotellessa. Jos dataa ja mutexia käyttää useampi kuin yksi säie, asiat huononevat edelleen prosessorien tai ytimien lisäyksen myötä kasvavan kiivaan kilpailutilanteen todennäköisyyden kasvamisena. Käytettäessä useita säikeitä saman datan prosessoimiseksi nopeammin säikeet kilpailevat keskenään

datasta ja siten myös mutexista. Mitä enemmän säikeitä on, sitä todennäköisempää on tilanne, että ne yrittävät saada mutexin haltuunsa tai muokata atomista muuttujaa samaan aikaan. [2]

Kilpailun vaikutukset ovat yleensä erilaiset mutexien ja atomisten operaatioiden välillä, sillä mutexin käyttäminen sarjallistaan säikeiden suorituksen käyttöjärjestelmän tasolla ja atomiset operaatiot prosessoritasolla. Kun säie odottaa mutexin vapautumista, käyttöjärjestelmä voi aikatauluttaa toisen säikeen suoritukseen, jos tällaisia suoritukseen pääsyä odottavia säikeitä on. Prosessoritasolla seisahdus taas estää kaikkien säikeiden suorittamisen prosessorilla. Tästä huolimatta mutexien tapauksessa kilpailu kuitenkin vaikuttaa suorituskykyyn niissä säikeissä, jotka kisaavat mutexin haltuunotosta. [2]

Välimuistipoukkoilua voidaan yrittää ehkäistä yrittämällä vähentää tilanteita, joissa kaksi tai useampi säie kilpailevat samasta muistipaikasta. Se ei välttämättä ole helppoa ja asiaa vaikeuttaa lisäksi se, että välimuistipoukkoilua saattaa tapahtua, vaikka tiettyä muistipaikkaa käyttäisi ainoastaan yksi säie. Tällöin on kyse väärästä jakamisesta. [2]

2.3.3 Väärä jakaminen

Prosessorien välimuistit käyttävät yleensä 32 tai 64 tavun kokoisia muistilohkoja yksittäisten muistipaikkojen sijasta. Lohkojen koot riippuvat prosessorin arkkitehtuurista. Koska välimuistilaitteisto käsittelee ainoastaan muistilohkon kokoisia paloja muistia, vierekkäisten osoitteiden data päätyy samaan muistilohkoon. Jos säie käsittelee esimerkiksi datasettiä, on tietorakenteen alkioiden sijaitsemisesta samassa muistilohkossa enemmän hyötyä suorituskyvyn kannalta kuin jos alkiot olisivat hajallaan useiden muistilohkojen alueella. Mikäli muistilohkossa on toisistaan riippumatonta dataa, jota useat säikeet käyttävät, voi siitä seurata huomattavia suorituskykyongelmia. [2]

Otetaan esimerkiksi taulukko, jossa on kokonaislukuja, ja joukko säikeitä, joista jokainen käyttää ja muokkaa ainoastaan yhtä omaa alkiotaan taulukosta toistuvasti. Kokonaislukujen pienen koon vuoksi useita taulukon alkioita mahtuu samaan välimuistilohkoon. Tämän seurauksena välimuistipoukkoilua alkaa tapahtua siitä huolimatta, että jokainen säie koskee ainoastaan yhteen omaan alkioonsa taulukossa. Välimuistilohkon omistajuus täytyy aina siirtää sille säikeelle, joka muokkaa omaa alkiotaan taulukossa, eli dataa siirretään yhden ytimen välimuistista toiselle toistuvasti. Välimuistilohko on jaettu säikeiden kesken, vaikka data ei olekaan, ja juuri tätä kutsutaan vääräksi jakamiseksi. Ratkaisuna väärälle jakamiselle on järjestää data niin, että saman säikeen käyttämät data-alkiot olisivat vierekkäin muistissa ja täten todennäköisesti samassa välimuistilohkossa, kun taas

useiden eri säikeiden käyttämät data-alkiot olisivat kaukana toisistaan muistissa eli todennäköisemmin eri välimuistilohkoissa. [2]

2.3.4 Datan läheisyys

Datan sijoittelulla voi olla vaikutusta myös yksinään toimivan säikeen suorituskykyyn sen lisäksi, että kahden tai useamman säikeen käsitellessä liian lähellä toisiaan muistissa olevia data-alkiota ilmenee väärää jakamista. Kyseessä on datan läheisyys. Jos yhden säikeen käyttämä data on hajallaan muistissa, on se todennäköisesti eri välimuistilohkoissa. Vastaavasti, jos säikeen käyttämä data on lähekkäin muistissa, on se todennäköisesti samassa välimuistilohkossa. Hajautuksen asteesta riippuen muistin käytön viive voi kasvaa ja suorituskyky laskea, jos hajallaan olevaa dataa käsitellessä joudutaan lataamaan muistista useampia lohkoja prosessorin välimuistiin. [2]

Lisäksi hajallaan olevan datan tapauksessa on todennäköisempää, että muistilohkossa on säikeelle turhaa tai tarpeetonta dataa. Pahimmillaan tarpeetonta dataa saattaa olla muistilohkossa enemmän kuin tarpeellista, mikä tuhlaa tärkeää välimuistitilaa ja kasvattaa välimuistihutien mahdollisuutta. Prosessori voi tällöin joutua hakemaan dataa uudelleen muistista välimuistiin, jos se joutui poistamaan kyseisen datan aiemmin tehdäkseen tilaa välimuistiin. [2]

Vaikka datan läheisyyden aiheuttamat suorituskykyvaikutukset käsittelevätkin vain yksisäikeistä koodia, ovat nämä vaikutukset relevantteja samanaikaisuuden kannalta kontekstivaihtojen takia. Jos järjestelmässä on enemmän säikeitä kuin suorittavia ytimiä, ytimet suorittavat useampaa kuin yhtä säiettä. Tämä kasvattaa välimuistin roolia, kun yritetään estää väärää jakamista varmistamalla, että eri säikeet käyttävät eri välimuistilohkoja. Tällöin taas on todennäköistä, että hajallaan olevan datan takia kontekstinvaihdon yhteydessä joudutaan välimuistiin lataamaan uudet muistilohkot, missä uuden säikeen tarvitsema data sijaitsee. C++17-standardi määrittelee vakion std::hardware_constructive_interference_size, joka määrittää välimuistilohkossa olevien taattujen peräkkäisten tavujen maksimimäärän. Jos tarvittava data saadaan mahdutettua tähän tavumäärään, välimuistihutien määrää saadaan mahdollisesti vähennettyä. [2]

Jos säikeitä on suorittavia ytimiä enemmän, käyttöjärjestelmä saattaa aikatauluttaa säikeen ensin yhdelle ytimelle yhdeksi aikajaksoksi ja sen jälkeen toiselle ytimelle toiseksi aikajaksoksi. Tällöin tietysti joudutaan jälleen siirtämään ensimmäisen ytimen välimuistista lohkoja toisen ytimen välimuistiin. Mitä enemmän muistilohkoja joudutaan siirtä-

män, sitä kauemmin siirrossa kestää. Käyttöjärjestelmät yrittävät välttää tällaista tilannetta, mutta välillä niitä tapahtuu, mikä vaikuttaa suorituskykyyn. Tehtävien vaihtoon liittyvät ongelmat ovat erityisen yleisiä tapauksissa, joissa useat säikeet ovat valmiina suoritukseen odotustilan sijasta, jolloin kyseessä on ylitilaus. [2]

2.3.5 Ylitilaaminen ja liialliset kontekstivaihdot

Monisäikeisissä järjestelmissä on tyypillistä, että säikeitä on enemmän kuin prosessoreita tai ytimiä, jollei kyseessä ole sitten erittäin rinnakkainen laitteisto. Säikeet odottavat usein ulkoisten I/O-operaatioiden (input/output) valmistumista, mutexien vapautumista, ehtomuuttujia ja niin edelleen, joten tästä ei yleensä koidu ongelmia. ”Ylimääräisten” säikeiden ansiosta prosessorit saadaan pidettynä kiireisenä, eivätkä ne jää niin sanotusti tyhjäkäynnille yhtä usein, kun säikeiden täytyy odottaa jotakin. [2]

Tämä ei ole kuitenkaan aina hyvä asia. Jos ylimääräisiä säikeitä on liikaa, suoritukseen valmiina olevia säikeitä on enemmän kuin vapaita prosessoreja tai ytimiä. Tällöin käyttöjärjestelmä alkaa vaihtamaan tehtäviä nopeaan tahtiin, jotta kaikki säikeet saavat reilun määrän suoritusaikaa. Itse tehtävien vaihdosta koituvan taakan lisäksi myös datan läheisyyden puuttumisesta johtuvat välimuistiongelmat saattavat kasautua. Ylitilausta voi tapahtua, kun jokin tehtävä luo lisää säikeitä toistuvasti ilman rajoitteita. Ylitilaus saattaa myös ilmetä tilanteissa, joissa tehtävätyypin mukaisesti erotettujen säikeiden luontainen määrä on prosessorien tai ytimien määrää suurempi ja tehtävä työ on CPU-painotteista (Central Processing Unit) I/O-painotteisuuden sijasta. [2]

Jos säikeitä luodaan liikaa datan jaottelun takia, työskentelevien säikeiden määrää voidaan rajoittaa. Jos ylitilaus taas johtuu työn luonnollisesta jakautumisesta, tilanteen helpottamiseksi ei voi juurikaan tehdä muuta kuin vaihtaa jakoperustetta, mikä todennäköisesti vaatisi enemmän tietoa kohdeympäristöstä ja olisi kannattavaa ainoastaan huonon suorituskyvyn demonstroitavissa olevan parannuksen saavuttamiseksi. [2]

Muutkin tekijät voivat vaikuttaa monisäikeisen koodin suorituskykyyn. Välimuistipoukkoilusta aiheutuva suorituskyvyn heikkeneminen voi vaihdella paljonkin esimerkiksi kahden yksiytimisen prosessorin järjestelmän ja yhden kaksiytimisen prosessorin järjestelmän välillä, vaikka prosessorit olisivatkin tyypeiltään ja kellonopeuksiltaan samanlaisia. Nämä kaksi tekijää kuitenkin ovat suurimpia tekijöitä, joilla on näkyvä vaikutus suorituskykyyn. [2]

2.4 Skaalautuvuus

Skaalautuvuudella pyritään siihen, että sovellus voi käyttää hyväkseen järjestelmässä olevia ylimääräisiä prosessoreja. Yhdessä ääripäässä yksisäikeinen sovellus on täysin skaalautumaton, eli sen suorituskyky ei kasvaisi, vaikka järjestelmään lisättäisiin prosessoreita. Toisessa ääripäässä on sovelluksia tai projekteja, jotka pystyvät hyödyntämään tuhansia prosessoreita. [2]

Skaalautuvuutta voidaan tarkastella jakamalla ohjelma sarjallisiin osioihin, joissa ainoastaan yksi säie suorittaa hyödyllistä työtä ja rinnakkaisiin osioihin, joissa kaikki vapaana olevat prosessorit tai ytimet suorittavat hyödyllistä työtä. Jos sarjallisten osioiden kokoa tai säikeiden odotuspotentiaalia pystytään pienentämään, saadaan suorituskyvyn hyötyjen potentiaalia kasvatettua järjestelmissä, joissa on enemmän prosessoreita tai ytimiä. Myös rinnakkaisten osien pitäminen jatkuvasti kiireisenä tarjoamalla systeemille enemmän dataa prosessoitavaksi kasvattaa suorituskykyhyötyä. [2]

Skaalautuvuudessa on kyse siis tehtävien tekemiseen kuluvan ajan pienentämisestä sekä tietyssä ajassa prosessoitavan datan määrän kasvattamisesta, kun prosessoreita tai ytimiä lisätään. Joskus nämä tarkoittavat samaa asiaa, sillä dataa voidaan prosessoida enemmän, jos tehtäviä suoritetaan nopeammin. [2]

2.5 Lukot

C++-ohjelmointikielessä on olemassa erilaisia lukitusmenetelmiä kriittisten alueiden suojaamiseksi. Ilman lukkoja voi usean säikeen tapauksessa esiintyä kilpailutilanteita.

Kilpailutilanne tapahtuu, kun kaksi säiettä käsittelevät jaettua muuttujaa samaan aikaan. Kummatkin säikeet lukevat saman arvon muuttujasta, minkä jälkeen säikeet suorittavat operaationsa muuttujan arvolle. Tällöin ne kilpailevat siitä, kumpi säikeistä saa kirjoitettua muutoksensa viimeisenä tähän jaettuun muuttujaan. Viimeisimpänä muutoksensa tehneen säikeen arvo talletetaan muuttujaan, sillä aiemmin muutoksensa tehneen säikeen muutos korvataan uudella. [7] Näin ollen kilpailutilanteet aiheuttavat ohjelman määrittelemätöntä käyttäytymistä [2].

Yleisin merkki kilpailutilanteesta onkin useiden säikeiden välillä jaettujen muuttujien arvaamattomat arvot, mikä seuraa säikeiden suoritusjärjestyksen ennustamattomuudesta. Joskus toinen säie voittaa, joskus taas toinen, ja välillä suoritus toimii virheettömästi. Lisäksi muuttujan arvo käyttäytyy oikein, jos säikeet suoritetaan erikseen. [7]

Kilpailutilanteen välttämiseksi on erilaisia keinoja. Voidaan yrittää esimerkiksi suunnitella datan rakenne niin, että muokkaukset tehdään atomisten operaatioiden sarjana. Tätä kutsutaan lukkovapaaksi ohjelmoinniksi ja se on hankalaa toteuttaa oikein. Tällä tasolla työskenneltäessä muistimallin vivahteet ja säikeiden eri arvoihin vaikuttamisen huomaaminen voivat olla hyvinkin monimutkaisia. [2] Onkin yksinkertaisempaa käyttää lukkoja kilpailutilanteiden estämiseksi. Perustavanlaatuisin ja yleisin mekanismi jaetun datan suojaamiseksi C++-kielessä on mutex [2]. Mutexin lisäksi käytetään myös muita lukkoja, esimerkiksi spinlockia.

2.5.1 Mutex ja spinlock

Mutex tai tarkemmin sanottuna std::mutex on synkronointiprimitiivi, jota voidaan käyttää suojaamaan jaettua dataa useampien säikeiden yhtäaikaiselta käsittelyltä eli signaloimaan poissulkevaa pääsyä kriittiselle alueelle [8-9]. Sama pätee myös spinlockiin, mutta näiden lukkotyyppien toimintatavoilla on eroavaisuuksia.

Jos säie yrittää lukita mutexia, joka on jo lukittu jonkin toisen säikeen toimesta, käyttöjärjestelmä siirtää lukitusta yrittävän säikeen nukkumaan, päästäen näin jonkin toisen säikeen suoritukseen. Nukkuva säie herää vasta, kun lukkoa hallussa pitänyt säie vapauttaa lukon. Kun säie yrittää lukita spinlockia, eikä onnistu siinä, se yrittää toistuvasti uudelleen spinlockin lukitsemista, kunnes lopulta onnistuu siinä. Näin se estää samalla toisen säikeen pääsyn suoritukseen. Käyttöjärjestelmä tosin pakottaa säikeen pois suorituksesta, kun säikeelle annettu suoritusaika ylittyy. [10]

Mutexien käytössä ongelmana on se, että säikeiden nukkumaan asettaminen ja herättäminen ovat kummatkin kalliita operaatiota, jotka tarvitsevat monta prosessorikäskyä ja vievät siis aikaa. Jos mutex on lukittuna vain hyvin pienen hetken, säikeen nukkumaan asettamiseen ja herättämiseen kuluva aika saattaa ylittää itse nukkumisajan tai jopa ajan, joka olisi tuhlattu yrittämällä jatkuvasti saada spinlock lukittua. Toisaalta spinlockin lukituksen yrittäminen kuluttaa jatkuvasti prosessointiaikaa, joten jos lukkoa ei vapauteta pitkään aikaan, olisi säie kannattavampaa asettaa nukkumaan. [10]

Spinlockien käyttämisestä yksiytimisen prosessorin järjestelmässä ei ole kannattavaa, koska spinlockin lukituksen jatkuva yrittäminen estää muiden säikeiden suorittamisen ainoalla mahdollisella ytimellä, täten estäen lukon vapautumisen. Tällöin on järkevämpää asettaa säie nukkumaan. Moniydinprosessorin omaavassa järjestelmässä, jossa lukkoja pidetään hallussa vain lyhyitä aikoja kerrallaan, säikeiden jatkuva nukkumaan asettaminen ja herättäminen voivat huonontaa suorituskykyä merkittävästi. Spinlockeja

käyttämällä tällaisessa järjestelmässä säikeillä on mahdollisuus hyödyntää koko niille suotu suoritusaikajakso ainoastaan pienen tukkimisajan jälkeen, mikä kasvattaisi suoritustehoa. [10]

Useimmiten ohjelmistokehittäjillä ei ole tietoa kohdejärjestelmän arkkitehtuurista eikä käyttöjärjestelmä voi tietää, kuinka moniytimiselle ympäristölle koodi on optimoitu, joten valintaa mutexin ja spinlockin välillä ei todennäköisesti voida etukäteen tehdä. Suurimmassa osassa moderneja järjestelmiä onkin käytössä hybridimutexit ja -spinlockit. [10]

Hybridimutex käyttäytyy moniydinjärjestelmässä aluksi kuten spinlock. Säiettä ei siis aseteta välittömästi nukkumaan, jos mutexin lukitseminen ei heti onnistu, sillä mutex voi vapautua nopeasti. Jos lukon vapautumisessa kestää, tietyn ajan kuluttua säie asetetaan nukkumaan. Hybridispinlock käyttäytyy aluksi samalla tavalla kuin normaali spinlock, mutta sillä on keinoja suoritusajan liiallisen tuhlaamisen välttämiseksi. Säiettä ei yleensä aseteta nukkumaan, sillä sen ei haluta tapahtuvan spinlockia käytettäessä. Säikeen suoritus voidaan kuitenkin pysäyttää luovuttamalla suoritus toiselle säikeelle, jolloin todennäköisyys lukon vapautumiselle kasvaa. Tällöin vältytään säikeen nukkumaan asettamisesta ja herättämisestä koituvista kustannuksista, mutta aikaa kuluu silti tehtävänvaihtoon. [10]

2.5.2 Deadlock eli lukkiutuminen

Lukkojen käytöstä saattaa seurata myös ongelmia, kuten deadlock eli lukkiutuminen. Tällainen tilanne syntyy, kun kaksi säiettä ottavat haltuun eri lukot samaan aikaan ja yrittävät sen jälkeen saada haltuunsa toisen säikeen jo omistamaa lukkoa. Tämän seurauksena kumpikin säie lopettaa suorituksensa ja odottaa, että toinen säie vapauttaa lukkonsa. Toisin sanoen, mitään ei tapahdu ja säikeet pysyvät lukkiutuneina. [7]

Lukkiutumisen tavallinen merkki on se, että ohjelma tai joukko säikeitä lakkaavat vastaamasta eli ne jumiutuvat [7]. Yleinen ohje lukkiutumisen välttämiselle on kahden synkronointiprimitiivin, kuten mutexin, lukitseminen aina samassa järjestyksessä, eli jos mutexin A lukitsee aina ennen mutexia B, ei lukkiutumista tapahdu. Joskus asia on näin yksinkertainen, jos mutexeja käytetään erillisiin tarkoituksiin. Toisinaan asia on monimutkaisempi, kuten esimerkiksi tilanteessa, jossa mutexit suojelevat kahta erillistä saman luokan instanssia operaatiossa, joka vaihtaa dataa näiden instanssien välillä. Jotta data vaihdetaan virheettä ilman, että siihen vaikuttavat samanaikaiset muutokset, on molemmat mutexit lukittava. Nyt valittu lukitsemisjärjestys – esimerkiksi ensimmäisenä parametrina annetun instanssin mutexin lukitseminen ensin ja toisena parametrina annetun

instanssin mutexin lukitseminen sen jälkeen – voi aiheuttaa hankaluuksia. Lukkiutuminen voi tapahtua, jos kaksi säiettä yrittävät suorittaa datanvaihto-operaation samojen instanssien välillä parametrit vastakkaisissa järjestyksissä. [2]

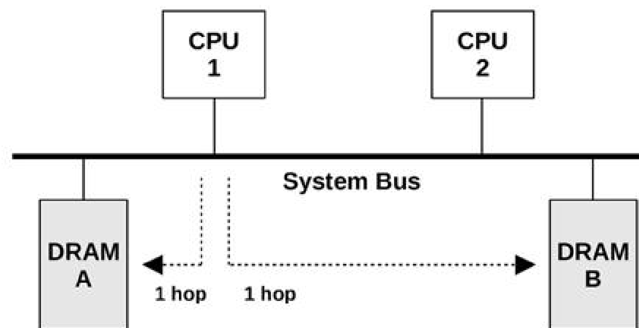
C++-standardikirjastosta löytyy tälle ongelmalle kuitenkin vastaus. Funktiolla std::lock on mahdollista lukita kaksi tai useampi mutex samaan aikaan ilman pelkoa lukkiutumisesta. [2]

2.6 Muistirakenne

Moniydinjärjestelmissä muisti voidaan jakaa jaettuun ja hajautettuun muistiin ja nykyään monissa järjestelmissä käytetään kumpaakin näistä muistityypeistä. Tämän työn alueessa keskitytään vain jaettuun muistiin. Jaetun muistin järjestelmät voidaan jakaa muistin hakuajan yhdenmukaisuuden perusteella kahteen kategoriaan: UMA (Uniform Memory Access) ja NUMA. [11]

2.6.1 Uniform Memory Access (UMA)

UMA-järjestelmissä jaetun muistin prosessoreilla on yhdenvertainen ja yhtenäinen pääsy muistiin [11]. Kuvassa 5 on näkyvillä kahden prosessorin muodostama UMA-järjestelmä.

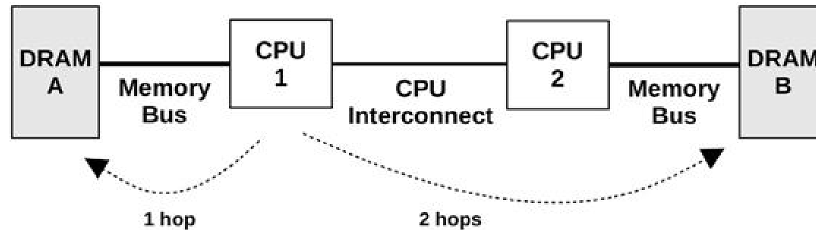


Kuva 5. Kahden prosessorin UMA-järjestelmä [12].

Prosessorien muistinkäyttö tapahtuu yhteisen muistiväylän kautta [12]. Näin CPU 1 - ja CPU 2 -prosessoreilla on sama, yhtenäinen viive DRAM A - ja DRAM B -muistien käsittelyssä.

2.6.2 Non-Uniform Memory Access (NUMA)

NUMA-järjestelmissä prosessoreiden päämuistin hakuaika vaihtelee muistin sijainnista prosessorin suhteen. Muistirakenteeseen kuuluu, että prosessorit ovat yhdistettyinä toisiinsa. [12] Kuvassa 6 on esitettynä kahden prosessorin muodostama NUMA-järjestelmä.



Kuva 6. Kahden prosessorin NUMA-järjestelmä [12].

CPU 1 voi käyttää DRAM A -muistia suoraan sen muistiväylän kautta eli DRAM A toimii tässä tapauksessa paikallisena muistina CPU 1:lle. DRAM B:n I/O-operaatioita CPU 1 joutuu tekemään prosessoreiden välisen väylän ja CPU 2:n kautta, mikä tarkoittaa kahta ”hyppyä” yhden sijaan. DRAM B:tä voidaankin kutsua CPU 1:n näkökulmasta etämuistiksi, jonka käyttämisellä on suurempi viive kuin paikallisella muistilla. [12]

Prosessoreihin yhdistettyjä muistilohkoja kutsutaan muistinoodeiksi tai pelkästään noodeiksi. Mikäli käyttöjärjestelmä on tietoinen tällaisesta muistinooditopologiasta, muistin varaus ja säikeiden vuoronnus on mahdollista pohjata muistin paikallisuuteen. Näin paikallista muistia saadaan suosittua mahdollisimman paljon suorituskyvyn parantamiseksi. [12]

2.6.3 NUMA-tutkimus

Nokian toteuttamassa NUMA-tutkimuksessa käytettiin palvelinta, jossa oli kaksi prosessorikantaa. Kun Kerääjää ja Parsija-pluginia ajettiin näillä kahdella kannalla, huomattiin yksittäisten prosessoriytimien ja säikeiden systeemikuorman osuuden olevan yli kaksi kertaa suurempi kuin sovelluskuorman. Korkean systeemikuorman takia alettiin epäillä, että säikeiden suoritus hyppeli eri prosessorikantojen NUMA-noodien välillä, aiheuttaen runsaasti koordinointityötä järjestelmälle. Kerääjän ajaminen rajoitettiin ainoastaan yhden prosessorikannan noodeihin, jolloin systeemikuorma tippui huomattavasti. Syötettävän liikenteen määrä kasvatettiin lähes kaksinkertaiseksi, mutta systeemikuorman osuus oli silti pienempi kuin alkuperäisessä tilanteessa kahta prosessorikantaa käytettäessä.

Tästä saavutuksesta huolimatta systeemikuorma näytti olevan silti vieläkin liian korkealla. Asiaa tutkiessa selvisi, että kuormittavin operaatio oli ”native_queued_spin_lock_slowpath”. Kyseessä oli siis säikeiden suorituksen synkronointiin liittyvää toimintaa. Jatkoselvittelyissä ajettiin testipluginia, joka suoritti ainoastaan ASN.1-dekoodausta eikä tarvinnut minkäänlaista synkronointia säikeiden välillä. Tällöin päästiin moninkertaiseen liikennemäärään systeemikuorman pysyessä silti alhaisena. Näin voidaan päätellä, että laitteisto ei ole rajoittava tekijä ja että ohjelmiston suorituskyvyn optimointimahdollisuuksien tutkintaa tulisi jatkaa.

2.7 Perf-profilointityökalu

Komento *perf* tai *perf_events* on Linux-käyttöjärjestelemästä löytyvä profilointityökalu. Sen avulla voidaan muun muassa tarkastella prosessorin suorituskykylaskureita ja tracepointteja, ja se kykenee kevyeen profilointiin käyttämättä itse paljon resursseja. Suorituskykylaskurit ovat prosessorin laitteistorekistereitä, jotka laskevat laitteistotapahtumia, kuten suoritettuja käskyjä, välimuistihuteja sekä väärin ennustettuja polkuja. Työkalun helppokäyttöinen rajapinta tarjoaa esimerkiksi seuraavat komennot:

- *perf stat*: tapahtumien määrien keräys
- *perf record*: tapahtumien tallentaminen myöhempää tarkastelua varten
- *perf report*: tapahtumien purkaminen prosessittain, funktioittain jne.
- *perf annotate*: assembly- tai lähdekoodin kommentointi tapahtumien määrillä
- *perf top*: nykyisten tapahtumamäärien tarkastelu suorana
- *perf bench*: erilaisten kernel-mikrosuorituskykytestien ajaminen

[13]

Kuvassa 7 on esillä osa komennon *perf top* tuottamasta listasta senhetkisistä tapahtumista ja niiden aiheuttamat kuormat, joista näkee, kuinka monta prosenttia ajastaan prosessori viettää kyseisen tapahtuman tai operaation parissa.

```
Samples: 21K of event 'cpu-clock:pppH', 100 Hz,
Event count (approx.): 8042766538 lost: 0/0 drop: 0/0

Overhead  Shared Object                      Symbol
  15.06%  [kernel]                           [k] __softirqentry_text_start
  10.42%  [kernel]                           [k] _raw_spin_unlock_irqrestore
   5.62%  [kernel]                           [k] finish_task_switch
   3.88%  libc-2.28.so                       [.] __memmove_avx_unaligned_erms
   3.24%  [kernel]                           [k] run_timer_softirq
   2.61%  [kernel]                           [k] rcu_core
   2.36%  [JIT] tid 2514                     [.] 0x00007f084003ae00
   1.97%  [JIT] tid 2514                     [.] 0x00007f082006edd3
   1.75%  [kernel]                           [k] update_sd_lb_stats.constprop.118
   1.23%  [kernel]                           [k] kmem_cache_free
   0.82%  [kernel]                           [k] rcu_work_rcufn
   0.79%  [kernel]                           [k] tick_nohz_idle_exit
   0.65%  [kernel]                           [k] do_idle
   0.62%  [kernel]                           [k] rcu_idle_exit
   0.57%  [kernel]                           [k] do_syscall_64
   0.57%  libc-2.28.so                       [.] __memcmp_avx2_movbe
   0.50%  [JIT] tid 2514                     [.] 0x00007f084003adee
   0.47%  libc-2.28.so                       [.] __memset_avx2_erms
   0.45%  libz.so.1.2.11                     [.] adler32_z
   0.45%  [kernel]                           [k] rcu_gp_kthread
   0.44%  libpthread-2.28.so                 [.] __pthread_mutex_lock
   0.38%  libglib-2.0.so.0.5600.4            [.] g_mutex_unlock
   0.38%  [kernel]                           [k] rcu_nocb_unlock_irqrestore
   0.38%  [JIT] tid 2514                     [.] 0x00007f082006ef79
   0.37%  [kernel]                           [k] nft_do_chain
   0.36%  [kernel]                           [k] load_balance
   0.35%  [kernel]                           [k] refill_obj_stock
   0.35%  libc-2.28.so                       [.] malloc
   0.35%  [kernel]                           [k] __pv_queued_spin_lock_slowpath
   0.33%  [JIT] tid 2514                     [.] 0x00007f084003b067
   0.33%  libz.so.1.2.11                     [.] 0x00000000000033e0
   0.31%  [kernel]                           [k] run_rebalance_domains
   0.30%  [JIT] tid 2514                     [.] 0x00007f084003a4ba
   0.30%  Xvnc                               [.] rfb::ZRLEEncoder::writePixels
   0.29%  [kernel]                           [k] wake_up_process
   0.29%  [kernel]                           [k] find_next_and_bit
   0.28%  libglib-2.0.so.0.5600.4            [.] g_slice_alloc
   0.26%  [kernel]                           [k] __fget
   0.26%  libpthread-2.28.so                 [.] __libc_recvmsg
   0.26%  libz.so.1.2.11                     [.] 0x00000000000034df
   0.24%  libz.so.1.2.11                     [.] 0x000000000000459c
   0.24%  libc-2.28.so                       [.] __libc_enable_asynccancel
```

Kuva 7. Osa *perf top* -komennon tuottamasta listasta, jossa näkyy nykyisten tapahtumien tuottama suhteellinen kuorma.

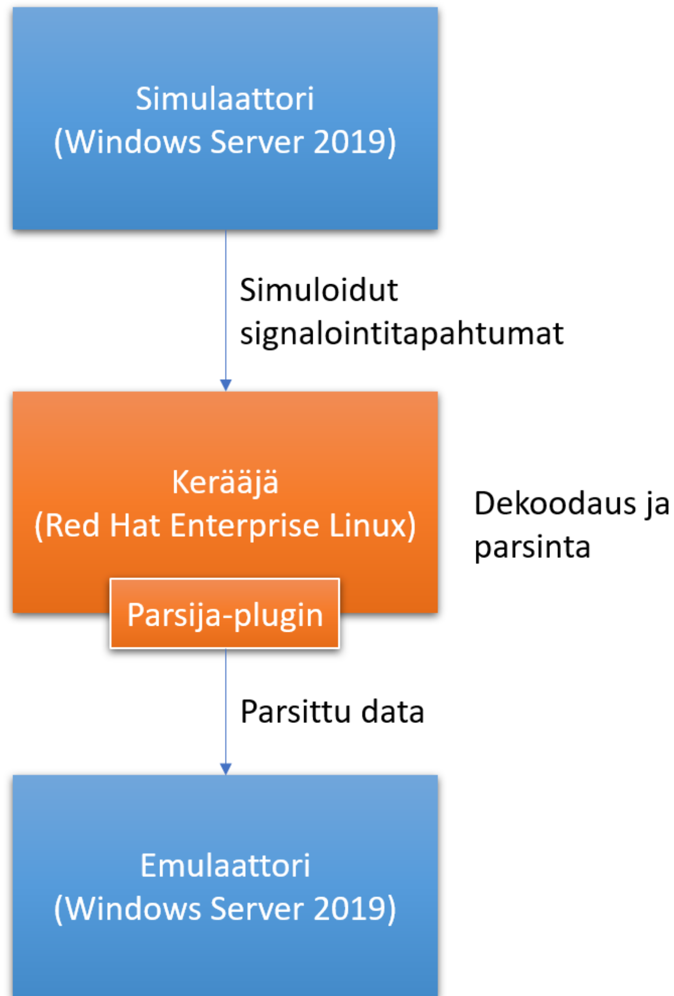
Komentoihin voidaan lisätä myös prosessin ID (PID), jolloin komento on esimerkiksi muotoa *perf top -p PID*. Tällöin päästään tutkimaan tarkemmin kyseisen prosessin sisäisiä tapahtumia ja niiden aiheuttamia kuormia. [14]

3. TUTKIMUS

Tutkimuksen tavoitteena on löytää Kerääjä-ohjelmistokomponentin Parsija-pluginin koodista samanaikaisuuteen liittyviä suorituskykypullonkauloja, joiden vuoksi pluginin suorituskykyskaalautuvuus on Kerääjän muiden pluginien skaalautuvuutta huomattavasti huonompi. Ongelmakohtien löytämisellä ja korjaamisella toivotaan olevan pluginin kokonaissuorituskykyä parantava vaikutus. Suorituskykypullonkaulojen etsimiseen käytetään Linux-käyttöjärjestelmän omaa *perf*-profilointityökalua. Löytyvien pullonkaulojen ja niiden korjausvaihtoehtojen suorituskykyä tutkitaan koodiin toteutettavilla kahden pisteen aikaeromittausten avulla.

3.1 Tutkimusympäristö

Työn tutkimusympäristönä toimii virtuaalikoneella pyörivä Red Hat Enterprise Linux versio 8.4 (Ootpa), jossa Kerääjää ajetaan. Prosessoriytimiä on 16 ja keskusmuistia löytyy 16 Gt. Käytössä olevien prosessorien määrää voidaan säätää Kerääjän konfiguraatiotiedostosta. Prosessorien korkean lukumäärän pitäisi mahdollistaa rinnakkaisuuteen liittyvien suorituskykyongelmien havaitsemista ja analysointia. Linux-virtuaalikoneen lisäksi tutkimusympäristöön kuuluu vielä kaksi muuta virtuaalikonetta. Tutkimusympäristöä havainnollistetaan Kuvassa 8.

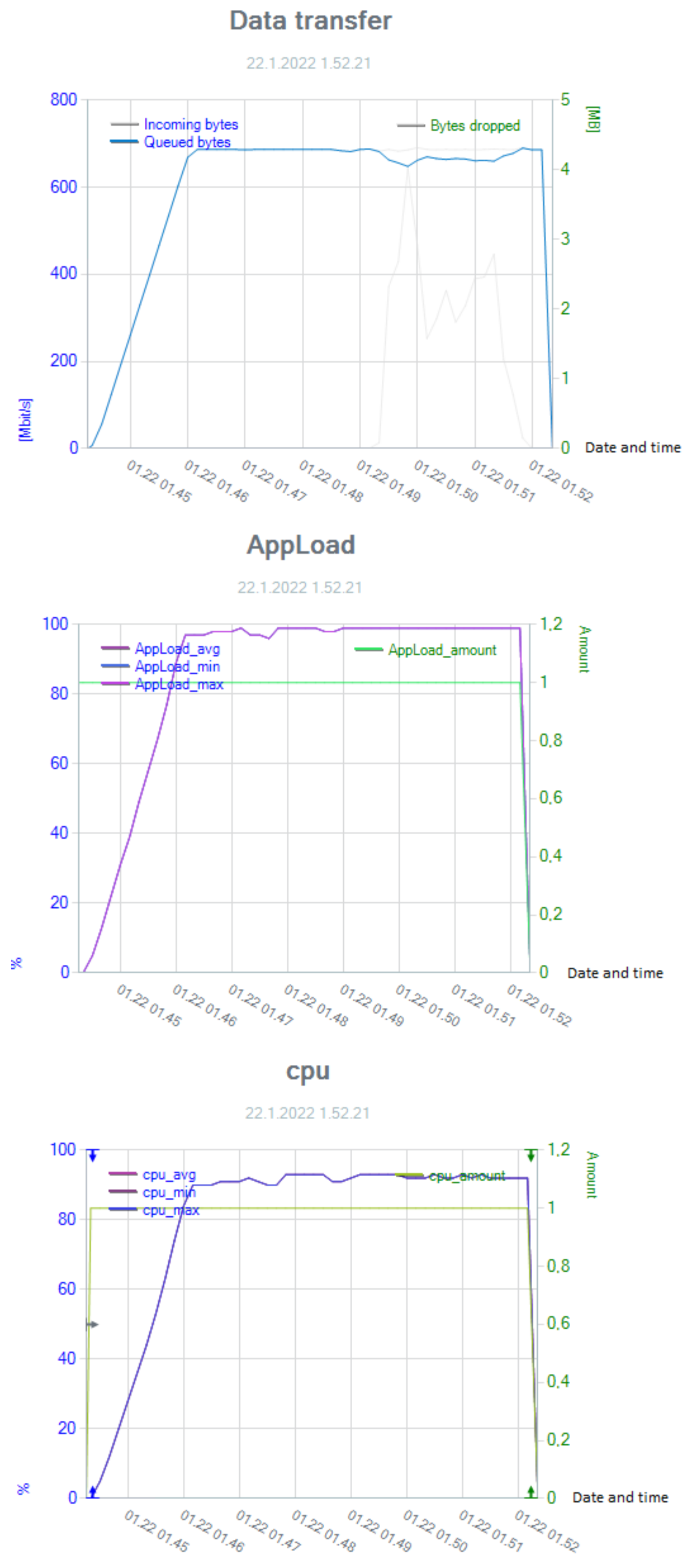


Kuva 8. Tutkimusympäristön koostavat virtuaalikoneet ja niiden tehtävät.

Tukiasemilta Kerääjälle sisään tulevaa liikennettä simuloidaan Windows Server 2019 -virtuaalikoneella pyörivällä simulaattorilla, koska simulaattori on Windows-pohjainen sovellus. Näiden koneiden välinen yhteys on tarpeeksi nopea, jotta saavutetaan riittävä lähetysnopeus Kerääjän kuormittamiselle. Lisäksi Parsija-pluginin ulostulon vastaanottavaa komponenttia emuloivaa sovellusta ajetaan toisella Windows Server 2019 -virtualikoneella, sillä emulaattoriohjelma on myös Windows-pohjainen.

3.2 Suorituskyvyn tarkastelumenetelmä

Parsija-pluginin kokonaissuoritustehon kasvaminen olisi ideaali lopputulos tutkimukselle. Pluginin suorituskykyä sekä esimerkiksi sovelluksen ja prosessorin kuormitusasteita voidaan tarkastella Nokian omalla sisäisellä sovelluksella, joka saa rajapinnan kautta statistiikkatietoja Kerääjältä. Kuvassa 9 on esitettynä kuvaajat Kerääjän ja Parsija-pluginin käsittelemälle datamäärälle, sovelluskuormalle sekä prosessorin kuormitusasteelle.



Kuva 9. Ylhäältä alas: Kerääjän ja Parsija-pluginin käsittelemä datamäärä, sovelluskuorma ja prosessorin kuormitusaste.

Kuvaajista nähdään, että vastaanottopuskurit täyttyvät ja dataa jätetään käsittelemättä ("bytes dropped"), kun sovelluskuorma kasvaa 100 prosenttiin. Kuvaajien tallennushetkellä Kerääjälle syötettiin enemmän simuloitua liikennettä kuin sen Parsija-plugin pystyy

käsittelemään. Prosessorin kuormitusasteesta voidaan päätellä, että prosessori pystyisi käsittelemään vielä enemmänkin dataa, mutta sovelluksen käsittelyraja tulee aikaisemmin vastaan.

3.3 Profilointi

Tutkimusta suoritetaan profiloinnin avulla. Tähän käytetään Linux-käyttöjärjestelmästä löytyvää *perf*-profilointityökalua, jolla voidaan etsiä paljon prosessointiaikaa kuluttavia tapahtumia tai operaatioita. Ensin käynnistetään Kerääjä, jonka jälkeen sille aletaan syöttää simuloitua dataa simulaattorilla. Tämän jälkeen kuormaa tarkastellaan komennolla *perf top -p $(pgrep Kerääjä)*, jolla saadaan esille juuri Kerääjän sisäiset tapahtumat ja niiden kuormat. Toisena vaihtoehtona on käyttää komentoa *perf record -p $(pgrep Kerääjä)*, jolloin kuorman jakautuminen tallennetaan perf.data-tiedostoon, jota voidaan tarkastella myöhemmin komennolla *perf report*. Kuvassa 10 on nähtävissä Kerääjän ja ajossa olevan Parsija-pluginin tapahtumia profiloituina ennen koodimuutoksia. Kerääjän ajamiseen käytetään 16 ydintä ja simuloitua liikennettä lähetetään enemmän kuin Kerääjä pystyy käsittelemään ilman vastaanottopuskurien täyttymistä.

```
Samples: 202K of event 'cpu-clock:pppH', 100 Hz,
Event count (approx.): 893415765587 lost: 0/0 drop: 0/0

Overhead  Shared Object               Symbol
   5.05%  libcasnrts.so               [.] asnBitFrameGetnFirstBits
   3.47%  [kernel]                    [k] __pv_queued_spin_lock_slowpath
   3.03%  libcasnrts.so               [.] pp_pushPath
   2.79%  libcasnrts.so               [.] pp_popPath
   2.75%  libc-2.28.so                [.] __memmove_avx_unaligned_erms
   2.71%  libParsija.so               [.] Funktio: HaePuhelu
   2.64%  libc-2.28.so                [.] __strlen_avx2
   2.19%  libpthread-2.28.so          [.] __pthread_mutex_lock
   1.94%  [kernel]                    [k] _raw_spin_unlock_irqrestore
   1.87%  libParsija.so               [.] Funktio: ParsiRaportti
   1.76%  libpthread-2.28.so          [.] __lll_lock_wait
   1.72%  libpthread-2.28.so          [.] __lll_unlock_wake
   1.66%  [kernel]                    [k] do_syscall_64
   1.49%  [kernel]                    [k] finish_task_switch
   1.31%  libpthread-2.28.so          [.] __pthread_mutex_unlock_usercnt
   1.28%  libc-2.28.so                [.] _int_malloc
   1.20%  libc-2.28.so                [.] malloc
   1.12%  libParsija.so               [.] boost::detail::sp_counted_base::release
   1.10%  libParsija.so               [.] Util::IntervalStatistics::Add
   1.01%  libcasnrts.so               [.] asnBitFrameGetnFirstBytes
   0.95%  libcasnrts.so               [.] walloc
   0.93%  [vdso]                      [.] __vdso_clock_gettime
   0.90%  ld-2.28.so                  [.] __tls_get_addr
   0.86%  libParsija.so               [.] Funktio: HaePuheluListasta
   0.85%  libParsija.so               [.] Funktio: PäivitäPuhelu
   0.84%  libc-2.28.so                [.] _int_free
   0.74%  libcasnrts.so               [.] pp_pushElem
   0.71%  libc-2.28.so                [.] __strcpy_avx2
   0.59%  [kernel]                    [k] futex_wake
   0.57%  libParsija.so               [.] Funktio: HoidaParsinta
   0.54%  libc-2.28.so                [.] cfree@GLIBC_2.2.5
   0.54%  libcasnrts.so               [.] pd_constrWNAlLe64K
   0.52%  [kernel]                    [k] __raw_spin_unlock_irq
   0.50%  [kernel]                    [k] futex_wait_setup
```

Kuva 10. Kerääjän ja Parsija-pluginin tapahtumia ennen koodimuutoksia.

Listan kärjessä näkyvä ASN.1-käsittely on aikaa vievää, mutta sitä ei ole mahdollista nopeuttaa. Listassa näkyviä tapahtumien nimiä on muokattu, jotta esimerkiksi funktioiden oikeat nimet eivät paljastuisi. Symbol-sarakkeessa näkyvät ”Funktio”-alkuiset tapahtumat ovat funktioita, jotka viittaavat tiettyihin kohtiin koodissa ja joita voidaan tarkastella suorituskyvyn kannalta. Tässä tutkimuksessa keskitytään funktioon HaePuhelu ja siinä kutsuttavaan funktioon HaePuheluListasta, sillä muista funktioista ei löydy ainakaan yhtä helposti samanaikaisuudesta johtuvia mahdollisia suorituskykypullonkauloja. Lisäksi funktio HaePuhelu on eniten CPU-aikaa kuluttava tapahtuma listaan erikseen merkityistä funktioista.

3.4 Lukkojen vertailu

Näkyvimmäksi mahdolliseksi keinoksi samanaikaisuuteen liittyvän suoritustehon parantamiseksi funktiossa HaePuhelu osoittautuu synkronointiprimitiivin lukituksen käytön tarkastelu. Jos lukko on käyttötarkoitukseen nähden huonosti valittu, voi se aiheuttaa merkittävänkin pullonkaulan funktion suoritusteholle ja tätä myötä myös mahdollisesti koko tarkasteltavan pluginin suoritustehon skaalautuvuudelle useampien säikeiden tapauksessa. Kuten alaluvussa 2.4 mainittiin, skaalautuvuutta voidaan parantaa pienentämällä sarjallisten osien kokoa sekä säikeiden potentiaalia joutua odottamaan. Lukkojen käsittelystä aiheutuvaa aikaa pienentämällä ja kriittisen alueen suoritusta nopeuttamalla voidaan siis parantaa suoritustehoa. Aiemmin alaluvussa 2.6.3 käsitellyssä NUMA-tutkimuksessakin havaittiin, että mittauksissa raskaimmaksi operaatioksi nousi ”native_queued_spin_lock_slowpath”, mikä viittaisi lukkojen käytöstä johtuvaan hidastumiseen.

Funktiossa on käytössä luku-kirjoituslukkoyhdistelmän lukuoperaatioita varten tarvittava lukulukko, joka käyttää yksinkertaista spinlockia lukulukkolaskurimuuttujan arvon muokkaamiseksi. Kutsuttavassa funktiossa HaePuheluListasta on käytössä rakenteeltaan samanlainen yksinkertainen spinlock kuin sitä kutsuvan funktion lukulukossa, ja tällaisen spinlockin rakenne on näkyvillä Kuvassa 11.

```
1   void Lock()
2   {
3       while (m_lock.test_and_set(std::memory_order_acquire))
4       {
5           std::this_thread::yield();
6       }
7   }
8
9   void Unlock()
10  {
11      m_lock.clear(std::memory_order_release);
12  }
```

Kuva 11. Yksinkertaisen spinlockin rakenne, jossa muuttuja m_lock on tyypiltään std::atomic_flag.

Kuvan spinlockissa lukitus tapahtuu kutsumalla test_and_set-funktiota, joka palauttaa arvon ”false”, mikäli mikään muu säie ei ole sitä vielä kutsunut, ja arvon ”true”, jos jokin toinen säie on jo kutsunut sitä. Jos säie ei saa lukkoa haltuunsa, se lähettää käyttöjärjestelmälle ilmoituksen oman suorituksensa aikatauluttamisesta uudelleen myöhemmin

kutsumalla std::this_thread_yield-funktiota. Lukon vapautus tapahtuu asettamalla atomisesti std::atomic_flag-tyyppisen lipun m_lock arvoksi ”false” kutsumalla lipulle clear-funktiota.

Lukkojen vertailussa ja mittailussa mukaillaan Malte Skarupken blogissa [15] esiteltyjä menetelmiä sekä mahdollisia lukkovaihtoehtoja, sillä blogissa käsitellään samanlaista yksinkertaista spinlockia. Blogissa tarkastellaan, miten lukon käyttö aiheuttaa suorituskykypullonkaulan ja miten asiaa voidaan korjata muokkaamalla lukkoa tai vaihtamalla lukko toiseen lukkotyyppiin.

Kuvan 11 lukkoa voitaisiin parantaa välimuistilohkojen omistajuuksien kannalta. Muistin oikeellisuuden varmistamiseksi monien ytimien kirjoittaessa yhteiseen muistiin käytetään protokollia. Nykyiset prosessorit käyttävät jotakin versiota MESI-protokollasta (Modified, Exclusive, Shared, Invalid), jossa ”invalid”-tila pakotetaan muille ytimille ennen kuin välimuistilohkoon voidaan kirjoittaa. Jos siis useampi ydin yrittää saada spinlockin haltuunsa, niin ne kilpailevat keskenään välimuistilohkon omistajuudesta, pakottaen muita ytimiä ”invalid”-tilaan. Tästä seuraa turhaa koordinointityötä ja muistin siirtoa ytimien välimuistien välillä, mikä puolestaan hidastaa koko järjestelmää. [15] Korvataan yksinkertainen spinlock hieman muokatulla versiolla ja tarkastellaan, nopeutuuko funktion suoritus. Muokatun spinlockin rakenne on näkyvillä Kuvassa 12.

```
1  void Lock()
2  {
3      while (true)
4      {
5          bool isLocked = lock.load(std::memory_order_relaxed);
6          if (!isLocked && lock.compare_exchange_weak(
7              isLocked, true, std::memory_order_acquire))
8          {
9              break;
10         }
11
12         _mm_pause();
13     }
14 }
15
16 void Unlock()
17 {
18     lock.store(false, std::memory_order_release);
19 }
```

Kuva 12. Muokatun spinlockin rakenne, jossa muuttuja lock on tyypiltään std::atomic<bool>.

Tässä muokatussa spinlockissa otetaan huomioon välimuistilohkojen omistajuudesta aiheutuva kiista säikeiden kesken. Muisti ladataan ennen kuin siihen yritetään tehdä muutoksia koodirivillä 5. MESI-protokollan mukaisesti välimuistilohko voi täten olla jaetussa tilassa kaikissa prosessointiytimissä ilman kommunikaatiotarvetta ytimien välillä ennen kuin data oikeasti vaihtuu välimuistilohkossa. [15]

Kolmantena lukkona vertailussa käytetään tavallista mutexia. Näin saadaan mielenkiintoinen tarkastelunäkökulma yleisimmän C++:n lukitusmenetelmän pärjäämisestä spinlockeille tämän tutkimuksen puitteissa.

Lukkojen suorituskyvyn vaikutusta voidaan arvioida mittaamalla kokonaissuoritusaikaa, joka kuluu lukon ensimmäisestä varausyrityksestä lukon vapauttamiseen asti. Lisäksi voidaan mitata pelkästään viivettä, joka kuluu lukon ensimmäisestä varausyrityksestä lukon haltuun saamiseen asti. Yhtenä mahdollisuutena on myös mitata joutoaikaa eli niin sanottua säikeen tyhjäkäyntiä lukon luovuttamisen ja seuraavaksi suoritukseen pääsevän säikeen lukon haltuun ottamisen välillä. [15] Hitaimman ajan tallentamalla jokaisesta mittauksesta saadaan käsitystä huonoimman tapauksen tilanteesta.

Luotettavien tulosten takaamiseksi mittaukset toistettiin kolme kertaa ja suorituskeston keskiarvon laskennassa otettiin mukaan 500 000 000 toistoa tai erotusaikaa, koska lähetettävän datan simulaattorilla kestää hetki päästä täyteen vauhtiin. Kun simuloitavien tukiasemien määrä simulaattorissa lisääntyy, lähetetään viestejä enemmän, milloin tarkastelussa olevia funktioita kutsutaan useammin. Siksi toistokertojen määrän on hyvä olla korkea, jotta keskiarvo tasaantuu oikeaan arvoonsa. Samalla vähennetään myös kuorman ja säikeiden lisääntymisestä mahdollisesti johtuvien vuorontajan aikataulutusmuutosten vaikutusta. Vuorontaja hallitsee säikeiden suoritusta, joten aikataulutuksella on vaikutusta lukkojen myöntämiseen säikeille ja näin ollen suorituskyvylle. Kun saadaan enemmän toistoja oikealla kuormalla ja lukkojen varausyritystahdilla, niin saadaan paremmin todellista tilannetta kuvaava tulos aikaiseksi.

3.4.1 Suoritusaikamittaus

Suoritusajan mittaus voidaan toteuttaa ottamalla järjestelmäaika ennen lukon haltuun ottamista sekä lukon vapauttamisen jälkeen ja laskemalla aikojen erotus. Tällaisen mittauksen havainnollistava koodi on esitettynä Kuvassa 13.

```
1   auto startTime = std::chrono::high_resolution_clock::now();
2
3   Lock();
4   // Critical section
5   Unlock();
6
7   auto endTime = std::chrono::high_resolution_clock::now();
8   std::chrono::duration<double> duration = endTime - startTime;
9
10  sum += duration.count();
11  ++counter;
12  average = sum / counter;
13
14  if (duration.count() > longestDuration.count() || counter == 1)
15  {
16      longestDuration = duration;
17  }
```

Kuva 13. Suoritusaikamittauksessa käytetty koodi.

Suoritusajan keskiarvon ja hitaimman ajan mittaustulokset yksinkertaista spinlockia käytettäessä funktiolle HaePuhelu ovat esitettynä Taulukossa 1. Taulukossa on mukana kolmen mittauskerran tulokset ja jokaisen mittauskerran kolme hitainta aikaa.

Taulukko 1. Funktion HaePuhelu kokonaissuoritusaikojen keskiarvot ja hitaimmat ajat kolmelta mittauskerralta yksinkertaista spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	1,10734 µs	1,0237 µs	1,06658 µs
1. hitain aika	27,5165 ms	28,1653 ms	23,1108 ms
2. hitain aika	12,9311 ms	16,8707 ms	19,4076 ms
3. hitain aika	6,7505 ms	16,2699 ms	14,7452 ms

Kolmen mittauksen keskiarvoiseksi suoritusajaksi muodostuu taulukon tulosten perusteella kolmen numeron tarkkuuteen pyöristettynä 1,07 µs. Mittauskertojen hitaimpien aikojen keskiarvoksi puolestaan tulee 26,3 ms. Tarkastellaan seuraavaksi muokatulla spinlockilla funktiolle HaePuhelu saatuja mittaustuloksia, jotka näkyvät Taulukossa 2.

Taulukko 2. Funktion HaePuhelu kokonaissuoritusaikojen keskiarvot ja hitaimmat ajat kolmelta mittauskerralta muokattua spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	422,133 ns	382,146 ns	411,639 ns
1. hitain aika	12,9905 ms	9,05237 ms	3,69212 ms
2. hitain aika	4,08839 ms	2,6925 ms	2,12594 ms
3. hitain aika	2,70487 ms	0,620235 ms	2,06773 ms

Muokatulla spinlockilla mittausten keskiarvoiseksi suoritusajaksi muodostuu 405 ns ja hitaimpien aikojen keskiarvoksi 8,58 ms. Tarkastellaan lopuksi vielä Taulukosta 3 funktion HaePuhelu suoritusaikoja, kun lukon tilalle on vaihdettu mutex.

Taulukko 3. Funktion HaePuhelu kokonaissuoritusaikojen keskiarvot ja hitaimmat ajat kolmelta mittauskerralta mutexia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	1,2425 µs	1,20101 µs	1,14825 µs
1. hitain aika	14,0976 ms	11,7928 ms	17,7016 ms
2. hitain aika	9,36333 ms	9,06802 ms	13,9555 ms
3. hitain aika	8,25823 ms	7,63897 ms	8,49304 ms

Mutexia käyttämällä mittausten keskiarvoiseksi suoritusajaksi saadaan 1,20 µs, mikä on melko lähellä yksinkertaisen spinlockin tulosta. Hitaimpien aikojen keskiarvo on 14,5 ms.

Funktion HaePuhelu kokonaissuoritusajan mittausten tuloksista huomataan, että keskiarvoisista suoritusajoista nopein saavutetaan muokatulla spinlockilla ja hitain mutexilla. Muokatulla spinlockilla keskiarvoinen suoritusaika on noin 665 ns nopeampi kuin alkuperäisesti koodissa käytetyllä yksinkertaisella spinlockilla. Spinlockia vaihtamalla funktion suoritusnopeus on siis noin 62 % nopeampi kuin aikaisemmin.

Huonoimpien tilanteiden osalta yksinkertaisella spinlockilla näyttäisi olevan näistä kolmesta lukosta hitain aika ja muokatulla spinlockilla nopein. Tämä voisi viitata aiemmin mainittuun suorittavien säikeiden muistilohkojen omistajuuskiistelystä johtuvaan koordinointityöhön ytimien välimuistien välillä yksinkertaisen spinlockin tapauksessa.

3.4.2 Viivemittaus

Viiveajan mittaus voidaan suorittaa ottamalla järjestelmäaika juuri ennen lukon haltuunoton yritystä ja heti lukon saamisen jälkeen ja laskemalla aikojen erotus. Viiveaikaa mittaamalla saadaan kuvaa siitä, miten esimerkiksi käyttöjärjestelmän vuorontaja vaikuttaa lukon haltuunottonopeuteen. Saadaan siis enemmän tietoa kokonaissuoritusajan lisäksi, mihin muuhun aikaa kuluu järjestelmässä kriittisten alueiden yhteydessä. Tällainen mittaus on esillä Kuvan 14 koodissa.

```
1   auto startTime = std::chrono::high_resolution_clock::now();
2
3   Lock();
4
5   auto endTime = std::chrono::high_resolution_clock::now();
6   std::chrono::duration<double> duration = endTime - startTime;
7
8   sum += duration.count();
9   ++counter;
10  average = sum / counter;
11
12  if (duration.count() > longestDuration.count() || counter == 1)
13  {
14      longestDuration = duration;
15  }
16
17  // Critical section
18
19  Unlock();
```

Kuva 14. Viiveaikojen mittauksessa käytetty koodi.

Viiveen keskiarvon ja hitaimman ajan mittaustulokset yksinkertaista spinlockia käyttäessä funktiolle HaePuhelu ovat esitettynä Taulukossa 4. Taulukossa on mukana kolmen mittauskerran viiveiden tulokset sekä jokaisen mittauskerran kolme suurinta viiveaikaa.

Taulukko 4. Funktion HaePuhelu viiveaikojen keskiarvot ja suurimmat viiveet kolmelta mittauskerralta yksinkertaista spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	239,46 ns	235,928 ns	236,725 ns
1. suurin viive	3,0421 ms	3,77642 ms	5,37009 ms
2. suurin viive	2,94094 ms	3,73189 ms	1,62215 ms
3. suurin viive	1,46223 ms	2,93752 ms	1,57063 ms

Kolmen mittauksen keskiarvoiseksi viiveeksi saadaan taulukon tulosten perusteella kolmen numeron tarkkuuteen pyöristettynä 237 ns. Suurimpien viiveaikojen keskiarvoksi tulee 4,06 ms. Seuraavaksi siirrytään muokatulla spinlockilla saatuihin viiveaikoihin, jotka näkyvät Taulukossa 5.

Taulukko 5. Funktion HaePuhelu viiveaikojen keskiarvot ja suurimmat viiveet kolmelta mittauskerralta muokattua spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	312,141 ns	321,366 ns	294,917 ns
1. suurin viive	10,967 ms	6,93713 ms	3,04235 ms
2. suurin viive	10,9224 ms	2,03403 ms	2,1267 ms
3. suurin viive	1,83191 ms	1,77319 ms	2,05633 ms

Muokatulla spinlockilla mittausten keskiarvoinen viive on taulukon mukaan 309 ns ja suurimpien viiveaikojen keskiarvo 6,98 ms. Tarkastellaan vielä mutexia käyttämällä saatuja viiveaikoja Taulukossa 6.

Taulukko 6. Funktion HaePuhelu viiveaikojen keskiarvot ja suurimmat viiveet kolmelta mittauskerralta mutexia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	395,208 ns	381,628 ns	386,368 ns
1. suurin viive	3,46167 ms	4,00348 ms	1,68646 ms
2. suurin viive	2,05367 ms	0,522564 ms	1,30832 ms
3. suurin viive	1,98927 ms	0,490125 ms	0,837703 ms

Mutexin kanssa mittausten keskiarvoiseksi viiveeksi saadaan laskettua 388 ns. Suurimpien viiveaikojen keskiarvoksi tulee 3,05 ms.

Funktion HaePuhelu viiveaikojen mittaustuloksista nähdään, että keskiarvoisesti pienin viive lukon haltuunotolla on yksinkertaisella spinlockilla ja suurin viive mutexilla. Muokatun spinlockin keskiarvoisen viiveen mittaustulos on hieman yllättävä, sillä paremman suoritusaikatuloksen perusteella olisi voinut päätellä, että myös lukon haltuunoton viive olisi pienempi. Tämä tulos on myös ristiriidassa Skarupken suorittaman mittailun kanssa [15]. Ero voi johtua monista syistä, kuten eri prosessoreista ja ytimien sekä säikeiden määrästä tai käyttöjärjestelmän ja vuorontajan eroista. Voidaan myös ajatella, että muokatun spinlockin tilanteessa välimuistilohkon datan lataaminen ennen muutosten tekemistä siihen kuluttaa enemmän aikaa lukon haltuunoton kannalta, mutta pidemmällä aikavälillä eli suoritusaikatasolla kulutetaan lopulta yhteensä vähemmän aikaa.

Huonoimmissa tilanteissa mutexilla näyttäisi olevan pienimmät viiveet ja muokatulla spinlockilla suurimmat. Mutexin tapauksessa vuorontaja ilmeisesti pitää paremmin huolen reiluudesta, eivätkä säikeet joudu odottamaan lukkoa huonoimmassa tapauksessakaan yhtä kauan kuin muilla lukoilla. Toisaalta reiluuden ylläpitäminen saattaa olla syynä keskiarvoisen viiveen suurempaan kestoon.

3.4.3 Joutoaikamittaus

Joutoajan mittaus voidaan tehdä ottamalla järjestelmäaika juuri ennen lukon vapauttamista ja heti lukon saamisen jälkeen ja laskemalla aikojen erotus. Tällä mittauksella siis mitataan aikaa, jolloin lukko ei ole käytössä varausten välissä. Myös joutoaikaa mittaamalla saadaan enemmän tietoa vuorontajan vaikutuksesta lukkojen haltuunottoon. Vaikka suoritusaika olisi nopea, on mahdollista, että lukkoa joudutaan odottamaan pitkä aika. Joutoaikamittauksen suorittava koodi on nähtävillä Kuvassa 15. Joutoajan tarkka mittaus on hankalaa, mutta näin sen mittaamista voidaan ainakin yrittää [15].

```
1  static std::chrono::_V2::system_clock::time_point startTime;
2  static bool first = true;
3
4  Lock();
5
6  if (first)
7  {
8      first = false;
9  }
10 else
11 {
12     auto endTime = std::chrono::high_resolution_clock::now();
13     std::chrono::duration<double> duration = endTime - startTime;
14
15     sum += duration.count();
16     ++counter;
17     average = sum / counter;
18
19     if (duration.count() > longestDuration.count() || counter == 1)
20     {
21         longestDuration = duration;
22     }
23 }
24
25 // Critical section
26
27 startTime = std::chrono::high_resolution_clock::now();
28
29 Unlock();
```

Kuva 15. Joutoaikamittauksen suorittava koodi.

Koska ajan mittaaminen aloitetaan vasta juuri ennen lukon vapautusta, tarvitaan ensimmäisen funktion suorituskerran huomaamiseksi tarkistus. Näin ensimmäinen joutoajan mittaus saadaan otettua ylös vasta sitten, kun järjestelmäaika on jo tallennettu edellisellä suorituskerralla. Taulukossa 7 nähdään kolmen mittauksen keskiarvoiset joutoaikojen pituudet sekä huonoimpien tilanteiden suurimmat joutoajat yksinkertaiselle spinlockille.

Taulukko 7. Funktion HaePuhelu joutoaikojen keskiarvot ja suurimmat joutoajat kolmelta mittauskerralta yksinkertaista spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	811,236 ns	815,48 ns	817,024 ns
1. pisin joutoaika	81,7223 ms	79,1298 ms	78,2883 ms
2. pisin joutoaika	1,2377 ms	0,488141 ms	0,934577 ms
3. pisin joutoaika	0,947532 ms	0,201044 ms	0,732121 ms

Kolmen mittauksen keskiarvoiseksi joutoajaksi saadaan taulukon tulosten perusteella kolmen numeron tarkkuuteen pyöristettynä 815 ns. Suurimpien joutoaikojen keskiarvo on 79,7 ms. Tarkastellaan nyt muokatulla spinlockilla saatuja joutoaikoja Taulukossa 8.

Taulukko 8. Funktion HaePuhelu joutoaikojen keskiarvot ja suurimmat joutoajat kolmelta mittauskerralta muokattua spinlockia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	807,561 ns	812,02 ns	809,515 ns
1. pisin joutoaika	81,1887 ms	82,6562 ms	93,1661 ms
2. pisin joutoaika	76,353 ms	0,486698 ms	92,737 ms
3. pisin joutoaika	73,2043 ms	0,201752 ms	59,8578 ms

Muokatulla spinlockilla mittausten keskiarvoinen joutoaika on taulukon mukaan 810 ns ja suurimpien joutoaikojen keskiarvo 85,7 ms. Mutexia käyttämällä saadut keskimääräiset sekä suurimmat joutoajat ovat esitettynä Taulukossa 9.

Taulukko 9. Funktion HaePuhelu joutoaikojen keskiarvot ja suurimmat joutoajat kolmelta mittauskerralta mutexia käyttäen.

	1. mittaus	2. mittaus	3. mittaus
Keskiarvo	827,135 ns	808,743 ns	816,767 ns
1. pisin joutoaika	134,493 ms	81,3651 ms	92,2659 ms
2. pisin joutoaika	76,7575 ms	0,48133 ms	60,1811 ms
3. pisin joutoaika	0,729591 ms	0,202027 ms	0,499361 ms

Taulukosta laskemalla saadaan mutexin joutoaikojen keskiarvoksi 818 ns. Suurimpien joutoaikojen keskiarvoksi tulee 103 ms.

Mittaustuloksista huomataan, että keskiarvoiset joutoajat kaikilla kolmella eri lukolla ovat lähes samat. Lukon valinnalla ei tässä tapauksessa näyttäisi olevan juurikaan merkitystä säikeiden suorituksen aikatauluttamisessa vuorontajan toimesta. Viiveaikamittauksen tavoin myös joutoaikamittauksen tulos on ristiriidassa Skarupken suorittaman mittauksen kanssa [15].

Pisimpien joutoaikojen tuloksista on mainittava, että kaikki pisimmät ajat mitattiin noin ensimmäisten 200 funktion toistokerran aikana. Syynä tähän voisi olla se, että simulaattorin tuottama liikennemäärä on mittauksen alkuvaiheessa vielä alhainen, jolloin lähetettävien viestien käsittelyyn tarvittavaa funktiota – ja tätä kautta siinä käytettävää lukkoa – kutsutaan harvemmin. Tällöin joutoajat pitenevät, sillä mittauksessahan mitattiin lukon vapauttamisesta sen uuteen haltuunottoon kuluvaa aikaa. Ei siis ole täyttä varmuutta,

ovatko pisimmät joutoajan mittaustulokset luotettavia ja että vastaavatko ne arvoja oikeassa tilanteessa, jossa liikenteen määrä on huomattavasti suurempi kuin mittauksen alussa.

3.5 Tulosten arviointi

Kootaan funktion HaePuhelu suoritusajan, viiveen ja joutoajan kolmen mittauksen keskiarvoiset tulokset jokaiselle lukolle vielä yhteen taulukkoon helppoa tarkastelua varten. Kootut tulokset ovat näkyvillä Taulukossa 10.

Taulukko 10. Funktion HaePuhelu jokaisen lukon kolmen mittauskerran keskiarvoiset mittaustulokset suoritusajalle, viiveelle sekä joutoajalle.

	Suoritusaika	Viive	Joutoaika
Yksinkertainen spinlock	1,07 µs	237 ns	815 ns
Muokattu spinlock	405 ns	309 ns	810 ns
Mutex	1,20 µs	388 ns	818 ns

Kuten taulukosta nähdään, merkittävin ero lukkojen keskiarvoisten mittaustulosten välillä havaitaan funktion suoritusaikaa mitattaessa. Parsija-pluginin käyttötarkoituksen huomioiden viive- ja joutoajoilla ei ole niin suurta merkitystä, joten kaiken kaikkiaan pelkän funktion suoritusajan parantuminen on positiivinen lopputulos. Funktion suoritusaika parani muokattua spinlockia käyttämällä noin 665 ns eli 62 % alkuperäisesti käytettyyn yksinkertaiseen spinlockiin verrattuna. Pluginin kokonaissuorituskyvyssä ei kuitenkaan huomata selvää nopeutusta lukon päivityksen johdosta, kun tarkastellaan Kuvassa 9 esiteltyä kuvaajaa pluginin käsittelemästä datamäärästä. Syynä suorituskyvyn muuttumattomuuteen kokonaistasolla on todennäköisesti se, että *perf*-työkalulla CPU:n ajankäyttöä tutkittaessa (Kuva 10) funktiossa HaePuhelu vietettiin vain 2,71 % ajasta. Tällöin funktion suoritusajan noin 62 % nopeutumisella on ymmärrettävästi suhteellisen pieni vaikutus pluginin kokonaissuorituskykyyn.

Funktion HaePuheluListasta mittaustulokset tuottivat erittäin pieniä eroja eri lukkojen välillä sekä keskiarvoisten aikojen että huonoimpien aikojen osalta. Erojen puuttuminen saattaa johtua funktion lyhyydestä ja nopeasta suoritusajasta. Funktion suoritusajan, viiveen ja joutoajan kolmen mittauksen keskiarvoiset tulokset jokaiselle lukolle ovat esitettynä Taulukossa 11 samaan tapaan kuin vastaavat tulokset funktiolle HaePuhelu Taulukossa 10. Aikojen yhtäläisyyksien johdosta mittausten tarkempi tarkastelu ei ole mielekästä.

Taulukko 11. Funktion HaePuheluListasta jokaisen lukon kolmen mittauskerran keskiarvoiset mittaustulokset suoritusajalle, viiveelle sekä joutoajalle.

	Suoritusaika	Viive	Joutoaika
Yksinkertainen spinlock	149 ns	84 ns	908 ns
Muokattu spinlock	147 ns	84 ns	907 ns
Mutex	175 ns	90 ns	910 ns

Keskimääräiset suoritusajat ja viiveet ovat todella pienet. Suurehkot joutoajat kertovat vähäisemmästä lukon haltuunoton haastamisesta säikeiden kesken. Tämä saattaa olla syynä sille, että mutex on molempia spinlockeja hitaampi, joskin kyseessä ovat vain muutaman tai muutaman kymmenen nanosekunnin erot. Säikeen nukkumaanmenon ja kontekstinvaihdon kustannukset mutexin tapauksessa voivat olla liian suuret tilanteissa, joissa kriittinen alue suoritetaan nopeasti ja lukon haltuunoton haastaminen on vähäistä.

Mittaustulosten perusteella voidaan todeta, että mutexin käyttäminen ei joka tilanteessa ole paras vaihtoehto, vaikka se olisikin yleisin lukitusmenetelmä C++-ohjelmissa. Lukitusmenetelmä on suotavaa miettiä tapauskohtaisesti, ja eri vaihtoehtojen suorituskykyä kannattaa mitata, mikäli suorituskyky on tärkeä tekijä ohjelmistossa. Yleistävänä sääntönä voidaan kuitenkin sanoa, että mutexia tulisi käyttää, jos lukon kilpailuasteesta tai yleensäkään kohdeympäristön arkkitehtuurista ei ole selvyyttä. Spinlock on mutexia tapauskohtaisempi ja sen käyttö saattaa parantaa suorituskykyä tilanteissa, joissa lukon

haltuunotosta ei ole juurikaan kilpailua. Mittauksista ilmenee myös se, että välimuistin roolia ei pidä aliarvioida jaetun muistin käsittelyssä eikä monisäikeisessä koodissa yleisellä tasollakaan. Turhien välimuistioperaatioiden määrää tulisikin pyrkiä minimoimaan mahdollisimman paljon.

Työn tavoitteen voidaan katsoa osittain täyttyneen, sillä tutkimuksessa löydettiin samanaikaisuuteen liittyvä ongelmakohta. Toisaalta tutkittavan pluginin kokonaissuorituskyky ei kasvanut huomattavalla tasolla, eikä *perf*-komennolla saatu lista CPU-ajan jakautumisesta tuonut ilmi mitään niin suurta suorituskykypullonkaulaa, mitä aiemmin suoritetun NUMA-tutkimuksen tulosten perusteella olisi voitu odottaa. Voi olla, että ongelman esiintyminen ja sen helpompi löytäminen tarvitsevat vielä enemmän samaan aikaan suoritettavia prosessoreja tai ytimiä, kuin tässä tutkimuksessa käytetyssä ympäristössä oli käytössä. Tulosten perusteella voidaan päätellä, että työssä löydetty suorituskyvyn ongelmakohta ei ainakaan yksinään ole vastuussa Parsija-pluginin suorituskyvyn skaalautuvuusongelmista.

4. YHTEENVETO

Tässä työssä käsiteltiin Nokia Solutions and Networks Oy:n DCAP-ohjelmiston Kerääjä-komponentin Parsija-pluginin suorituskykyä. Tavoitteena oli etsiä ja mahdollisuuksien mukaan parantaa pluginin suorituskykypullonkauloja samanaikaisuuteen liittyen, sillä aiemmin toteutetussa yrityksen sisäisessä tutkimuksessa todettiin, että pluginin suorituskyky ei skaalautunut muiden pluginien tapaan prosessorien tai ytimien määrän kasvaessa.

Työssä käytiin läpi muun muassa samanaikaisuuden teoriaa, samanaikaisen koodin suoritustekijöitä, lukkojen käyttöä jaettua dataa käsiteltäessä sekä välimuistioperaatioiden merkitystä monisäikeisessä koodissa. Lisäksi esiteltiin myös *perf*-komennon käyttöä suorituskykypullonkaulojen etsimiseen.

Tutkimusvaiheessa keskityttiin eniten CPU-aikaa vievään funktioon, jossa kriittisen alueen suojaamiseksi käytetyn yksinkertaisen spinlockin suorituskykyä vertailtiin muokatun spinlockin sekä mutexin suorituskykyihin. Muokatussa spinlockissa pyrittiin ottamaan huomioon välimuistioperaatioita, mikä näytti tuottavan tulosta, sillä huomattavin mittaustulos oli funktion noin 62 % alkuperäistä nopeampi suoritusaika juuri muokattua spinlockia käytettäessä. Pluginin kokonaissuorituskyvyssä ei kuitenkaan huomattu eroa, joten tutkimuksen tavoitteen voidaan katsoa täyttyneen vain osittain.

LÄHTEET

[1] “Concurrency and Application Design.” Saatavissa (viitattu 24.04.2022): https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ConcurrencyandApplicationDesign/ConcurrencyandApplicationDesign.html

[2] A. Williams, *C++ Concurrency in Action*. New York: Manning Publications Co. LLC, 2019.

[3] “Cornell Virtual Workshop: Multi-Core Cache Sharing.” Saatavissa (viitattu 24.04.2022): https://cvw.cac.cornell.edu/codeopt/multicore

[4] “std::atomic<T>::fetch_add - cppreference.com.” Saatavissa (viitattu 24.04.2022): https://en.cppreference.com/w/cpp/atomic/atomic/fetch_add

[5] “atomic::fetch_add - C++ Reference.” Saatavissa (viitattu 24.04.2022): https://www.cplusplus.com/reference/atomic/atomic/fetch_add/

[6] “memory_order - C++ Reference.” Saatavissa (viitattu 24.04.2022): https://www.cplusplus.com/reference/atomic/memory_order/

[7] “Race conditions and deadlocks - Visual Basic | Microsoft Docs.” Saatavissa (viitattu 21.04.2022): https://docs.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/race-conditions-deadlocks

[8] “std::mutex - cppreference.com.” Saatavissa (viitattu 12.03.2022): https://en.cppreference.com/w/cpp/thread/mutex

[9] “mutex - C++ Reference.” Saatavissa (viitattu 20.03.2022): https://www.cplusplus.com/reference/mutex/mutex/

[10] “synchronization - When should one use a spinlock instead of mutex? - Stack Overflow.” Saatavissa (viitattu 22.04.2022): https://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex

[11] Bryon. Moyer, “Real world multicore embedded systems a practical approach : expert guide.” Newnes, Oxford, 2013.

[12] Brendan Gregg, *Systems Performance, 2nd Edition*. Pearson, 2020.

[13] “Perf Wiki.” Saatavissa (viitattu 11.10.2021): https://perf.wiki.kernel.org/index.php/Main_Page

[14] Brendan Gregg, “Perf examples.” Saatavissa (viitattu 18.10.2021): https://www.brendangregg.com/perf.html

[15] M. Skarupke, “Measuring Mutexes, Spinlocks and how Bad the Linux Scheduler Really is | Probably Dance.” Saatavissa (viitattu 13.12.2021): https://probablydance.com/2019/12/30/measuring-mutexes-spinlocks-and-how-bad-the-linux-scheduler-really-is/